ATSUI Reference

Carbon > Text & Fonts

2007-06-28

Adobe, Acrobat, and PostScript are trademarks or registered trademarks of Adobe Systems Incorporated in the U.S. and/or other countries.

Contents

Figures

ATSUI Reference

Framework:	ApplicationServices/ApplicationServices.h
Declared in	ATSLayoutTypes.h ATSUnicodeDirectAccess.h ATSUnicodeDrawing.h ATSUnicodeFlattening.h ATSUnicodeFonts.h ATSUnicodeGlyphs.h ATSUnicodeObjects.h ATSUnicodeTypes.h

Overview

Apple Type Services for Unicode Imaging (ATSUI) enables the rendering of Unicode-encoded text with advanced typographic features. It automatically handles many of the complexities inherent in text layout, including the correct rendering of text in bidirectional and vertical script systems.

ATSUI may be useful to developers who are writing new text editors or word processing applications that render Unicode-encoded text. You can also use ATSUI if you want to modify your existing application to support Unicode text rendering.

This document describes the ATSUI application programming interface (API) through version 2.4. If you are a font designer or want more information about fonts, see the Apple font site: http://developer.apple.com/fonts/

Functions by Task

Creating and Initializing Style Objects

`ATSUCreateStyle` (page 40)
    Creates an opaque style object containing only default style attributes, font features, and font variations.

`ATSUCreateAndCopyStyle` (page 38)
    Creates a copy of a style object.

`ATSUCompareStyles` (page 31)
    Compares the attribute values of two style objects.

`ATSUClearStyle` (page 30)
    Restores default values to a style object.

`ATSUStyleIsEmpty` (page 134)
: Indicates whether a style object contains only default values.

`ATSUSetStyleRefCon` (page 129)
: Sets application-specific data for a style object.

`ATSUGetStyleRefCon` (page 88)
: Obtains application-specific data for a style object.

`ATSUDisposeStyle` (page 49)
: Disposes of the memory associated with a style object.

Manipulating Style Attributes

`ATSUSetAttributes` (page 119)
: Sets style attribute values in a style object.

`ATSUCopyAttributes` (page 32)
: Copies all style attribute settings from a source style object to a destination style object.

`ATSUOverwriteAttributes` (page 113)
: Copies to a destination style object the nondefault style attribute settings of a source style object.

`ATSUUnderwriteAttributes` (page 138)
: Copies to a destination style object only those nondefault style attribute settings of a source style object that are at default settings in the destination object.

`ATSUGetAllAttributes` (page 59)
: Obtains an array of style attribute tags and value sizes for a style object.

`ATSUGetAttribute` (page 65)
: Obtains a style attribute value for a style object.

`ATSUGetContinuousAttributes` (page 66)
: Obtains the style attribute values that are continuous over a given text range.

`ATSUClearAttributes` (page 24)
: Restores default values to the specified style attributes of a style object.

Manipulating Font Features

`ATSUSetFontFeatures` (page 120)
: Sets font features in a style object.

`ATSUGetAllFontFeatures` (page 60)
: Obtains the font features of a style object that are not at default settings.

`ATSUGetFontFeature` (page 66)
: Obtains the font feature corresponding to an index into an array of font features for a style object.

`ATSUClearFontFeatures` (page 25)
: Restores default settings to the specified font features of a style object.

`ATSUGetFontFeatureTypes` (page 70)
: Obtains the available feature types of a font.

`ATSUCountFontFeatureTypes` (page 35)
: Obtains the number of available feature types in a font.

`ATSUGetFontFeatureSelectors` (page 68)
: Obtains the available feature selectors for a given feature type in a font.

`ATSUCountFontFeatureSelectors` (page 34)
: Obtains the number of available feature selectors for a given feature type in a font.

Manipulating Font Variations

`ATSUSetVariations` (page 133)
: Sets font variation axes and values in a style object.

`ATSUGetAllFontVariations` (page 61)
: Obtains a style object's font variation values that are not at default settings.

`ATSUGetFontVariationValue` (page 74)
: Obtains the current value for a font variation axis in a style object.

`ATSUClearFontVariations` (page 26)
: Restores default values to the specified font variation axes of a style object.

`ATSUGetIndFontVariation` (page 81)
: Obtains a variation axis and its value range for a font.

`ATSUCountFontVariations` (page 37)
: Obtains the number of defined variation axes in a font.

`ATSUGetFontInstance` (page 72)
: Obtains the font variation axis values for a font instance.

`ATSUCountFontInstances` (page 35)
: Obtains the number of defined font instances in a font.

Creating and Initializing Text Layout Objects

`ATSUCreateTextLayout` (page 41)
: Creates an opaque text layout object containing only default text layout attributes.

`ATSUCreateTextLayoutWithTextPtr` (page 42)
: Creates an opaque text layout object containing default text layout attributes as well as associated text and text styles.

`ATSUCreateAndCopyTextLayout` (page 39)
: Creates a copy of a text layout object.

`ATSUSetTextPointerLocation` (page 131)
: Associates text with a text layout object or updates previously associated text.

`ATSUGetTextLocation` (page 91)
: Obtains information about the text associated with a text layout object.

`ATSUSetRunStyle` (page 127)
: Defines a style run by associating style information with a run of text.

`ATSUGetRunStyle` (page 86)
: Obtains style run information for a character offset in a run of text.

`ATSUSetTextLayoutRefCon` (page 130)
: Sets application-specific data for a text layout object.

`ATSUGetTextLayoutRefCon` (page 91)
: Obtains application-specific data for a text layout object.

`ATSUDisposeTextLayout` (page 49)
: Disposes of the memory associated with a text layout object.

Manipulating Text Layout Attributes

`ATSUSetLayoutControls` (page 122)
: Sets layout control attribute values in a text layout object.

`ATSUCopyLayoutControls` (page 33)
: Copies all layout control attribute settings from a source text layout object to a destination text layout object.

`ATSUGetAllLayoutControls` (page 62)
: Obtains an array of layout control attribute tags and value sizes for a text layout object.

`ATSUGetLayoutControl` (page 82)
: Obtains a layout control attribute value for a text layout object.

`ATSUClearLayoutControls` (page 28)
: Restores default values to the specified layout control attributes of a text layout object.

Manipulating Line Attributes

`ATSUSetLineControls` (page 124)
: Sets layout control attribute values for a single line in a text layout object.

`ATSUCopyLineControls` (page 33)
: Copies line control attribute settings from a line in a source text layout object to a line in a destination text layout object.

`ATSUGetAllLineControls` (page 63)
: Obtains an array of line control attribute tags and value sizes for a line in a text layout object.

`ATSUGetLineControl` (page 83)
: Obtains a line control attribute value for a line in a text layout object.

`ATSUClearLineControls` (page 29)
: Restores default values to the specified line control attributes of a line in a text layout object.

Manipulating Line Breaks

`ATSUBreakLine` (page 22)
: Calculates and, optionally, sets a soft line break in a range of text.

`ATSUBatchBreakLines` (page 20)
: Calculates soft line breaks for the text associated with a text layout object.

`ATSUSetSoftLineBreak` (page 128)
: Sets a soft line break that you specify.

`ATSUGetSoftLineBreaks` (page 87)
: Obtains soft line breaks in a range of text.

`ATSUClearSoftLineBreaks` (page 30)
: Removes soft line breaks from a range of text.

Substituting Fonts

`ATSUMatchFontsToText` (page 106)
: Examines a text range for characters that cannot be drawn with the current font and suggests a substitute font, if necessary.

`ATSUSetTransientFontMatching` (page 132)
: Turns automatic font substitution on or off for a text layout object.

`ATSUGetTransientFontMatching` (page 92)
: Obtains whether ATSUI automatically performs font substitution for a text layout object.

`ATSUCreateFontFallbacks` (page 39)
: Creates an opaque object that can be set to contain a font list and a font-search method.

`ATSUSetObjFontFallbacks` (page 126)
: Assigns a font list and a font-search method to a font fallback object.

`ATSUGetObjFontFallbacks` (page 85)
: Obtains the font list and font-search method associated with a font fallback object.

`ATSUDisposeFontFallbacks` (page 48)
: Disposes of the memory associated with a font fallback object.

Identifying Fonts

`ATSUGetFontIDs` (page 71)
: Obtains a list of all the ATSUI-compatible fonts installed on the user's system.

`ATSUFontCount` (page 57)
: Obtains the number of ATSUI-compatible fonts installed on a user's system.

`ATSUFindFontName` (page 53)
: Obtains a name string and index value for the first font in a name table that matches the specified ATSUI font ID, name code, platform, script, and/or language.

`ATSUFindFontFromName` (page 52)
: Obtains an ATSUI font ID for the first entry in a name table that matches the specified name string, name code, platform, script, and/or language.

`ATSUGetIndFontName` (page 78)
: Obtains a name string, name code, platform, script, and language for the font that matches an ATSUI font ID and name table index value.

`ATSUCountFontNames` (page 36)
: Obtains the number of font names that correspond to a given ATSUI font ID.

`ATSUGetIndFontTracking` (page 80)
: Obtains the name code and tracking value for the font tracking that matches an ASTUI font ID, glyph orientation, and tracking table index.

`ATSUCountFontTracking` (page 37)
: Obtains the number of entries in the font tracking table that correspond to a given ATSUI font ID and glyph orientation.

`ATSUGetFontFeatureNameCode` (page 67)
: Obtains the name code for a font's feature type or selector that matches an ASTUI font ID, feature type, and feature selector.

`ATSUGetFontVariationNameCode` (page 74)
: Obtains the name code for the font variation that matches an ASTUI font ID and font variation axis.

`ATSUGetFontInstanceNameCode` (page 73)
: Obtains the name code for the font instance that matches an ASTUI font ID and font instance index value.

Drawing and Highlighting Text

`ATSUDrawText` (page 50)
: Renders a range of text at a specified location in a QuickDraw graphics port or Quartz graphics context.

`ATSUHighlightText` (page 101)
: Renders a highlighted range of text at a specified location in a QuickDraw graphics port or Quartz graphics context.

`ATSUUnhighlightText` (page 140)
: Renders a previously highlighted range of text in an unhighlighted state.

`ATSUSetHighlightingMethod` (page 121)
: Sets the method ATSUI uses to highlight and unhighlight text for a text layout object.

`ATSUGetTextHighlight` (page 90)
: Obtains the highlight region for a range of text.

`ATSUHighlightInactiveText` (page 100)
: Highlights previously selected text using an alpha value of 0.5.

`ATSUClearLayoutCache` (page 27)
: Clears the layout cache of a line or an entire text layout object.

Supporting User Interaction With Onscreen Text

`ATSUTextInserted` (page 136)
: Informs ATSUI of the location and length of a text insertion.

`ATSUTextDeleted` (page 135)
: Informs ATSUI of the location and length of a text deletion.

`ATSUTextMoved` (page 137)
: Informs ATSUI of the new memory location of relocated text.

`ATSUPositionToOffset` (page 115)
: Obtains the memory offset for the glyph edge nearest a mouse-down event.

`ATSUOffsetToPosition` (page 111)
: Obtains the caret position(s) corresponding to a memory offset.

`ATSUNextCursorPosition` (page 109)
: Obtains the memory offset for the insertion point that follows the current insertion point in storage order, as determined by a move of the specified length.

`ATSUPreviousCursorPosition` (page 117)
: Obtains the memory offset for the insertion point that precedes the current insertion point in storage order, as determined by a move of the specified length.

`ATSURightwardCursorPosition` (page 118)
: Obtains the memory offset for the insertion point to the right of the high caret position, as determined by a move of the specified length at a line direction boundary.

`ATSULeftwardCursorPosition` (page 103)
: Obtains the memory offset for the insertion point to the left of the high caret position, as determined by a move of the specified length at a line direction boundary.

`ATSUPositionToCursorOffset` (page 113)
: Obtains the memory offset for the glyph edge nearest a mouse-down event, after a move of the specified length.

`ATSUOffsetToCursorPosition` (page 110)
: Obtains the caret position(s) corresponding to a memory offset, after a move of the specified length.

Obtaining Text Metrics

`ATSUMeasureTextImage` (page 107)
: Obtains the image bounding rectangle for a line of text after final layout.

`ATSUGetUnjustifiedBounds` (page 93)
: Obtains the typographic bounding rectangle for a line of text prior to final layout.

`ATSUGetGlyphBounds` (page 75)
: Obtains the typographic bounds of a line of glyphs after final layout.

`ATSUCalculateBaselineDeltas` (page 23)
: Obtains the optimal baseline positions for glyphs in a style run.

`ATSUGlyphGetIdealMetrics` (page 97)
: Obtains resolution-independent font metric information for glyphs associated with a given style object.

`ATSUGlyphGetScreenMetrics` (page 99)
: Obtains device-adjusted font metric information for glyphs associated with a given style object.

`ATSUGetNativeCurveType` (page 84)
: Obtains the type of outline path used for glyphs associated with a given style object.

`ATSUGlyphGetCurvePaths` (page 96)
: Obtains the outline paths for a glyph associated with a given style object.

`ATSUGlyphGetCubicPaths` (page 95)
: Obtains the cubic outline paths for a glyph.

`ATSUGlyphGetQuadraticPaths` (page 98)
: Obtains the quadratic outline paths for a glyph.

Working With Tabs

`ATSUSetTabArray` (page 129)
: Sets a tab ruler for a text layout object.

`ATSUGetTabArray` (page 89)
: Retrieves the tab ruler associated with a text layout object.

Accessing Glyph Data

`ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45)
: Obtains the glyph data specified by a direct-data selector and for a specific line of text.

`ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46)
: Obtains a copy of the glyph data specified by a direct-data selector and for a specific line of text in a text layout object.

`ATSUDirectReleaseLayoutDataArrayPtr` (page 48)
: Releases a pointer to a direct-data array.

`ATSUDirectAddStyleSettingRef` (page 44)
: Looks up, and if necessary, adds a style setting to a line of text.

Flattening and Parsing Style Data

`ATSUFlattenStyleRunsToStream` (page 55)
: Flattens ATSUI style-run data so that it can be saved to disk or passed (through the pasteboard) to another application.

`ATSUUnflattenStyleRunsFromStream` (page 138)
: Unflattens previously-flattened ATSUI style run data so that it can be read from disk or accepted (through the pasteboard) from another application.

Creating, Calling, and Deleting Universal Procedure Pointers

`NewATSUDirectLayoutOperationOverrideUPP` (page 155)
: Creates a new universal procedure pointer (UPP) to a layout operation override callback.

`InvokeATSUDirectLayoutOperationOverrideUPP` (page 150)
: Calls your layout operation override callback.

`DisposeATSUDirectLayoutOperationOverrideUPP` (page 145)
: Disposes of a universal procedure pointer (UPP) to a layout operation override callback.

`NewRedrawBackgroundUPP` (page 155)
: Creates a new universal procedure pointer (UPP) to a redraw background callback.

`InvokeRedrawBackgroundUPP` (page 150)
: Invokes your redraw background callback.

`DisposeRedrawBackgroundUPP` (page 146)
: Disposes of a new universal procedure pointer (UPP) to a redraw background callback.

`NewATSCubicMoveToUPP` (page 152)
: Creates a new universal procedure pointer (UPP) to a cubic move-to callback.

`InvokeATSCubicMoveToUPP` (page 147)
: Calls your cubic move-to callback.

`DisposeATSCubicMoveToUPP` (page 143)
: Disposes of a universal procedure pointer (UPP) to a cubic move-to callback.

`NewATSCubicLineToUPP` (page 152)
: Creates a new universal procedure pointer (UPP) to a cubic line-to callback.

`InvokeATSCubicLineToUPP` (page 147)
: Calls your cubic line-to callback.

`DisposeATSCubicLineToUPP` (page 143)
: Disposes of a universal procedure pointer (UPP) to a cubic line-to callback.

`NewATSCubicCurveToUPP` (page 151)
: Creates a new universal procedure pointer (UPP) to a cubic curve-to callback.

`InvokeATSCubicCurveToUPP` (page 146)
: Calls your cubic curve-to callback.

`DisposeATSCubicCurveToUPP` (page 142)
: Disposes of a universal procedure pointer (UPP) to a cubic curve-to callback.

`NewATSCubicClosePathUPP` (page 151)
: Creates a new universal procedure pointer (UPP) to a cubic close-path callback.

`InvokeATSCubicClosePathUPP` (page 146)
: Calls your cubic close-path callback.

`DisposeATSCubicClosePathUPP` (page 142)
: Disposes of a universal procedure pointer (UPP) to a cubic close-path callback.

`NewATSQuadraticNewPathUPP` (page 154)
: Creates a new universal procedure pointer (UPP) to a quadratic new-path callback.

`InvokeATSQuadraticNewPathUPP` (page 149)
: Calls your quadratic new-path callback.

`DisposeATSQuadraticNewPathUPP` (page 145)
: Disposes of a universal procedure pointer (UPP) to a quadratic new-path callback.

`NewATSQuadraticLineUPP` (page 154)
: Creates a new universal procedure pointer (UPP) to a quadratic line callback.

`InvokeATSQuadraticLineUPP` (page 149)
: Calls your quadratic line callback.

`DisposeATSQuadraticLineUPP` (page 144)
: Disposes of a universal procedure pointer (UPP) to a quadratic line callback.

`NewATSQuadraticCurveUPP` (page 153)
: Creates a new universal procedure pointer (UPP) to a quadratic curve callback.

`InvokeATSQuadraticCurveUPP` (page 148)
: Calls your quadratic curve callback.

`DisposeATSQuadraticCurveUPP` (page 144)
: Disposes of a universal procedure pointer (UPP) to a quadratic curve callback.

`NewATSQuadraticClosePathUPP` (page 153)
: Creates a new universal procedure pointer (UPP) to a quadratic close-path callback.

`InvokeATSQuadraticClosePathUPP` (page 148)
: Calls your quadratic close-path callback.

`DisposeATSQuadraticClosePathUPP` (page 144)
: Disposes of a universal procedure pointer (UPP) to a quadratic close-path callback.

Not Recommended

`ATSUFONDtoFontID` (page 56)
: Finds the ATSUI font ID that corresponds to a font family number, if one exists. (Deprecated. There is no replacement because FONDs are a QuickDraw concept and QuickDraw is deprecated.)

`ATSUFontIDtoFOND` (page 58)
: Finds the font family number that corresponds to an ATSUI font ID, if one exists. (Deprecated. There is no replacement because FONDs are a QuickDraw concept and QuickDraw is deprecated.)

`ATSUDrawGlyphInfo` (page 244) Deprecated in Mac OS X v10.3
: Draws glyphs at the specified location, based on style and layout information specified for each glyph. (Deprecated. Use functions from "Accessing Glyph Data" (page 18) instead.)

`ATSUGetFontFallbacks` (page 245) Deprecated in Mac OS X v10.3
: Obtains the global font list and search order that ATSUI uses when a font does not have the glyph needed to image a character. (Deprecated. Use font fallback objects instead.)

`ATSUGetGlyphInfo` (page 246) Deprecated in Mac OS X v10.3
: Obtains a copy of the style and layout information for each glyph in a line. (Deprecated. Use functions from "Accessing Glyph Data" (page 18) instead.)

`ATSUMeasureText` (page 247) Deprecated in Mac OS X v10.3
: (Deprecated. Use `ATSUGetUnjustifiedBounds` (page 93) instead.)

`ATSUSetFontFallbacks` (page 247) Deprecated in Mac OS X v10.3
: Sets, on a global scope, the font list and search order for ATSUI to use when a font does not have the glyph needed to image a character. (Deprecated. Use font fallback objects instead.)

`ATSUCopyToHandle` (page 243) Deprecated in Mac OS X v10.1
: Copies an ATSUI style to a handle. (Deprecated. Use `ATSUFlattenStyleRunsToStream` (page 55) instead.)

`ATSUCreateTextLayoutWithTextHandle` (page 239) Deprecated in Mac OS X v10.0
: Creates an opaque text layout object containing default text layout attributes as well as associated text and text styles. (Deprecated. Use `ATSUCreateTextLayoutWithTextPtr` (page 42) instead. See the Discussion for more details.)

`ATSUIdle` (page 241) Deprecated in Mac OS X v10.0
: Performs background processing. (Deprecated. There is no replacement because this function does nothing in Mac OS X.)

`ATSUSetTextHandleLocation` (page 241) Deprecated in Mac OS X v10.0
: Associates text with a text layout object. (Deprecated. Use `ATSUSetTextPointerLocation` (page 131) instead. See the Discussion for more details.)

Functions

ATSUBatchBreakLines

Calculates soft line breaks for the text associated with a text layout object.

```
OSStatus ATSUBatchBreakLines (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iRangeStart,
   UniCharCount iRangeLength,
   ATSUTextMeasurement iLineWidth,
   ItemCount *oBreakCount
);
```

Parameters

iTextLayout

The `ATSUTextLayout` for which you want to determine soft line breaks.

iRangeStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the text range to examine. To specify the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`.

iRangeLength

The number of characters in which to consider in the determination of the soft line breaks.

iLineWidth

An `ATSUTextMeasurement` value specifying the line width for the text, as measured from the offset provided in the `iLineStart` parameter. You must pass a nonzero value. You should use the same width as the width layout control set for the text layout object since the final layout of each line is based on the controls set for the line or the entire text layout object. If no line width has been set for the line, `ATSUBatchBreakLines` uses the line width set for the text layout object; if this value is not set, `ATSUBatchBreakLines` returns `paramErr`.

Note that the value you pass for the `iLineWidth` parameter is used only for the line-breaking operation. For justification, flushness, and other operations to work properly you must also use this value as the line width for the text layout object. You can set the line width for the text layout object by calling the function `ATSUSetLineControls` or `ATSUSetLayoutControls` with the `kATSULineWidthTag` and the line width value.

oBreakCount

The number of soft line breaks found and set from the call. If you do not want to obtain the number of soft line breaks, then set this parameter to `NULL`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUBatchBreakLines` function is equivalent to repeatedly calling the `ATSUBreakLine` function with the parameter `iUseAsSoftLineBreak` set to `true`. However the `ATSUBatchBreakLines` function performs more efficiently than repeated call to the `ATSUBreakLine` function.

You must call the `ATSUGetSoftLineBreaks` function to obtain the actual soft line breaks that were determined and set by the `ATSUBatchBreakLines` function.

Availability

Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeDrawing.h`

ATSUBreakLine

Calculates and, optionally, sets a soft line break in a range of text.

```
OSStatus ATSUBreakLine (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart,
   ATSUTextMeasurement iLineWidth,
   Boolean iUseAsSoftLineBreak,
   UniCharArrayOffset *oLineBreak
);
```

Parameters

iTextLayout

An ATSUTextLayout value specifying the text layout object to examine.

iLineStart

A UniCharArrayOffset value specifying the offset from the beginning of the text layout object's text buffer to the first character of the text range to examine. To specify the beginning of the text buffer, pass the constant kATSUFromTextBeginning. When calling ATSUBreakLine repeatedly to obtain all the soft line breaks for a given text range, in each subsequent call pass the value produced in the oLineBreak parameter by the immediately prior call to ATSUBreakLine.

iLineWidth

An ATSUTextMeasurement value specifying the line width for the text, as measured from the offset provided in the iLineStart parameter. You must pass a nonzero value. You can pass kATSUUseLineControlWidth to indicate that ATSUBreakLine should use the previously set line width attribute for the current line to determine how many characters can fit on the line. If no line width has been set for the line, ATSUBreakLine uses the line width set for the text layout object; if this value is not set, ATSUBreakLine returns paramErr.

Note that the value you pass for the iLineWidth parameter is used only for the line-breaking operation. For justification, flushness, and other operations to work properly you must also use this value as the line width for the text layout object. You can set the line width for the text layout object by calling the function ATSUSetLineControls or ATSUSetLayoutControls with the kATSULineWidthTag and the line width value.

iUseAsSoftLineBreak

A Boolean value indicating whether ATSUBreakLine should automatically set the line break produced in the oLineBreak parameter. If true, ATSUBreakLine sets the line break and clears any previously-set soft line breaks that precede the new break in the line but lie after the offset specified by iLineStart.

oLineBreak

A pointer to a UniCharArrayOffset value. On return, the value specifies the offset from the beginning of the text layout object's text buffer to the location of the calculated soft line break. If the value produced is the same value as specified in iLineStart, you have made an input parameter error. In this case, check to make sure that the line width specified in iLineWidth is big enough for ATSUBreakLine to perform line breaking. ATSUBreakLine does not return an error in this case. ATSUI usually calculates a soft line break to be at the beginning of the first word that does not fit on the line. But if ATSUBreakLine calculates the most optimal line break to be in the middle of a word, it returns the result code kATSULineBreakInWord. Note that ATSUI produces a line break in the middle of a word only as a last resort.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion
When the user inserts or deletes text or changes text layout attributes that affect how glyphs are laid out, you must determine whether the affected range of text still fits in the set line width, that is, whether the text needs to be rewrapped. You can use the `ATSUBreakLine` function to calculate a soft line break, based on the line width and text range you specify. If you pass `true` for `iUseAsSoftLineBreak`, `ATSUBreakLine` sets the soft line break it calculates and performs line layout on the characters.

If you need to calculate and set soft line breaks for a range of text and you want to use the same width for all lines in this range, you should call the function `ATSUBatchBreakLines` (page 20). Calling `ATSUBatchBreakLines` is equivalent to repeatedly calling the `ATSUBreakLine` function with the parameter `iUseAsSoftLineBreak` set to true. However, the `ATSUBatchBreakLines` function performs more efficiently than repeated calls to the `ATSUBreakLine` function.

If you do choose to call the `ATSUBreakLine` function repeatedly to obtain all possible line breaks for a range of text it will produce the previously set soft line break(s) if there are no additional line breaks to be found, or if the user has altered the text range or its attributes in a way that does not affect glyph layout.

The `ATSUBreakLine` function suggests a soft line break each time it encounters a hard line break character such as a carriage return, line feed, form feed, line separator, or paragraph separator. If `ATSUBreakLine` does not encounter a hard line break, it uses the line width you specify to determine how many characters fit on a line and suggests soft line breaks accordingly.

If you pass `true` for `iUseAsSoftLineBreak`, `ATSUBreakLine` uses the soft line break it calculates to perform line layout on the characters. `ATSUBreakLine` then determines whether the characters still fit within the line, which is necessary due to end-of-line effects such as swashes. When `ATSUBreakLine` sets a soft line break, it clears any previously-set soft line breaks that precede the new break in the line but lie after the offset specified by `iLineStart`.

Before calculating soft line breaks, `ATSUBreakLine` turns off any previously set line justification, rotation, width, alignment, descent, and ascent values and treats the text as a single line. Additionally, `ATSUBreakLine` examines the text layout object to ensure that each of the characters in the range is assigned to a style run. If there are gaps between style runs, `ATSUBreakLine` assigns the characters in the gap to the style run that precedes (in storage order) the gap. If there is no style run at the beginning of the text range, `ATSUBreakLine` assigns these characters to the first style run it finds. If there no style run at the end of the text range, `ATSUBreakLine` assigns the remaining characters to the last style run it finds.

For optimal performance, you should use `ATSUBreakLine` or `ATSUBatchBreakLines` to both calculate and set soft line breaks in your text. You should typically only call the function `ATSUSetSoftLineBreak` (page 128) to set soft line breaks when you are using your own line-breaking algorithm to calculate soft line breaks.

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

Declared In
`ATSUnicodeDrawing.h`

ATSUCalculateBaselineDeltas

Obtains the optimal baseline positions for glyphs in a style run.

```
OSStatus ATSUCalculateBaselineDeltas (
   ATSUStyle iStyle,
   BslnBaselineClass iBaselineClass,
   BslnBaselineRecord oBaselineDeltas
);
```

Parameters

iStyle

An ATSUStyle value specifying the style object to examine.

iBaselineClass

A BslnBaselineClass constant identifying the primary baseline from which to measure other baselines. See SFNTLayoutTypes.h for an enumeration of possible values. Pass the constant kBSLNNoBaselineOverride to use the standard baseline value from the current font.

oBaselineDeltas

A BslnBaselineRecord array consisting of Fixed values. On return, the array contains baseline offsets, specifying distances measured in points, from the default baseline to each of the other baseline types in the style object. Positive values indicate baselines above the default baseline and negative values indicate baselines below it. See SFNTLayoutTypes.h for a description of the BslnBaselineRecord type.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

Depending on the writing system, a baseline may be above, below, or through the centers of glyphs. In general, a style run has a default baseline, to which all glyphs are visually aligned when the text is laid out. For example, in a run of Roman text, the default baseline is the Roman baseline, upon which glyphs sit (except for descenders, which extend below the baseline).

You can call the ATSUCalculateBaselineDeltas function to obtain the distances from a specified baseline type to that of other baseline types for a given style object. ATSUCalculateBaselineDeltas takes into account font and text size when performing these calculations. ATSUI uses these distances to determine the cross-stream shifting to apply when aligning glyphs in a style run. You can use the resulting array to set or obtain the optimal baseline positions of glyphs in a style run. You can also set various baseline values to create special effects such as drop capitals.

The functions ATSUSetLineControls (page 124) and ATSUSetLayoutControls (page 122) allow you to set baseline offset values at the line or layout level, respectively, using the kATSULineBaselineValuesTag control attribute tag. For more information on kATSULineBaselineValuesTag, see "Attribute Tags" (page 196).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

ATSUnicodeObjects.h

ATSUClearAttributes

Restores default values to the specified style attributes of a style object.

```
OSStatus ATSUClearAttributes (
   ATSUStyle iStyle,
   ItemCount iTagCount,
   const ATSUAttributeTag iTag[]
);
```

Parameters

iStyle
An `ATSUStyle` value specifying the style object for which to restore default style attribute values.

iTagCount
An `ItemCount` value specifying the number of attributes to restore to default values. This value should correspond to the number of elements in the `iTag` array. To restore all style attributes in the specified style object, pass the constant `kATSUClearAll` in this parameter. In this case, the value in the `iTag` parameter is ignored.

iTag
A pointer to the initial `ATSUAttributeTag` constant in an array of attribute tags. Each tag should identify a style attribute to restore to its default value. See "Attribute Tags" (page 196) for a description of the Apple-defined style attribute tag constants.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUClearAttributes` function removes those style attribute values identified by the tag constants in the `iTag` array and replaces them with the default values described in "Attribute Tags" (page 196). If you specify that any currently unset attribute values be removed, the function does not return an error.

To remove all previously set style attribute, font feature, and font variation values from a style object, call the function `ATSUClearStyle` (page 30).

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

Declared In
`ATSUnicodeObjects.h`

ATSUClearFontFeatures

Restores default settings to the specified font features of a style object.

```
OSStatus ATSUClearFontFeatures (
   ATSUStyle iStyle,
   ItemCount iFeatureCount,
   const ATSUFontFeatureType iType[],
   const ATSUFontFeatureSelector iSelector[]
);
```

Parameters

iStyle
An `ATSUStyle` value specifying the style object for which to restore default font feature settings.

iFeatureCount
An `ItemCount` value specifying the number of font features to restore to default settings. This value should correspond to the number of elements in the *iType* and *iSelector* arrays. To restore default settings to all the font features in the specified style object, pass the constant `kATSUClearAll` in this parameter. In this case, the values in the `iType` and `iSelector` parameters are ignored.

iType
A pointer to the initial `ATSUFontFeatureType` value in an array of feature types. Each value should identify a font feature to restore to its default setting. To obtain all previously set font features for a given style object, you can call the function `ATSUGetAllFontFeatures` (page 60).

iSelector
A pointer to the initial `ATSUFontFeatureSelector` value in an array of feature selectors. Each element in the array must contain a valid feature selector corresponding to a font feature you provide in the *iType* parameter. To obtain all previously set feature selectors for a given style object, you can call the function `ATSUGetAllFontFeatures` (page 60).

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUClearFontFeatures` function removes those font features that are identified by the feature selector and type constants in the `iSelector` and `iType` arrays and replaces them with their font-defined default values. Note that if you pass `ATSUClearFontFeatures` a font feature and selector that are already at default settings, the function does not return an error.

To restore default font variations to a style object, call the function `ATSUClearFontVariations` (page 26). To restore default style attributes to a style object, call `ATSUClearAttributes` (page 24). To restore all default settings to a style object (for font features, variations, and style attributes), call the function `ATSUClearStyle` (page 30).

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

Declared In
`ATSUnicodeFonts.h`

ATSUClearFontVariations

Restores default values to the specified font variation axes of a style object.

```
OSStatus ATSUClearFontVariations (
   ATSUStyle iStyle,
   ItemCount iAxisCount,
   const ATSUFontVariationAxis iAxis[]
);
```

Parameters

iStyle
An `ATSUStyle` value specifying the style object for which to restore default font variation axis settings.

iAxisCount

An `ItemCount` value specifying the number of font variation axes to restore to default settings. This value should correspond to the number of elements in the *iAxis* array. To restore default values to all the font variation axes in the style object, pass the constant `kATSUClearAll` in this parameter. If you pass `kATSUClearAll` the value in the `iAxis` parameter is ignored.

iAxis

A pointer to the initial `ATSUFontVariationAxis` tag in an array of font variation axes. Each element in the array must contain a valid tag that corresponds to a font variation axis to restore to its default setting. You can obtain variation axis tags for a style object from the function `ATSUGetAllFontVariations` (page 61).

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUClearFontVariations` function removes those font variation axis values identified by variation axis tags in the `iAxis` array and replaces them with their font-defined default values. You can remove unset font variation values from a style object without a function error.

To restore default font features to a style object, call the function ATSUClearFontFeatures (page 25). To restore default style attributes, call `ATSUClearAttributes` (page 24). To restore all default settings to a style object (for font features, variations, and style attributes), call the function `ATSUClearStyle` (page 30).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFonts.h`

ATSUClearLayoutCache

Clears the layout cache of a line or an entire text layout object.

```
OSStatus ATSUClearLayoutCache (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value identifying the text layout object for which to clear a layout cache.

iLineStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the beginning of the line for which to discard the layout cache. If the range of text spans multiple lines, you should call `ATSUClearLayoutCache` for each line, passing the offset corresponding to the beginning of the new line to draw with each call. To clear the layout cache of the entire text layout object, you can pass the constant `kATSUFromTextBeginning`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The layout cache contains all the layout information ATSUI calculates and needs to draw a range of text in a text layout object. This includes caret positions, the memory locations of glyphs, and other information needed to lay out the glyphs. ATSUI uses information in the layout cache to avoid laying out the text again, thereby improving performance. When you clear the layout cache of a line or block of text, ATSUI takes longer to redraw a line, since it must perform the calculations that support glyph layout again.

You should call the function `ATSUClearLayoutCache` when you need to decrease the amount of memory your application uses. This function reclaims memory at the cost of optimal performance.

By default, the `ATSUClearLayoutCache` function removes the layout cache of a single line. To clear the layout cache for multiple lines, you should call `ATSUClearLayoutCache` for each line. To clear the layout cache of an entire text layout object, pass the constant `kATSUFromTextBeginning` in the `iLineStart` parameter. Note that `ATSUClearLayoutCache` does not produce a function error if lines do not have a layout cache.

The `ATSUClearLayoutCache` function flushes the layout cache but does not alter previously set text layout attributes, soft line break positions, or the text memory location. If you do not want to retain these values, you should dispose of the text layout object by calling the `ATSUDisposeTextLayout` (page 49) function.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeObjects.h`

ATSUClearLayoutControls

Restores default values to the specified layout control attributes of a text layout object.

```
OSStatus ATSUClearLayoutControls (
   ATSUTextLayout iTextLayout,
   ItemCount iTagCount,
   const ATSUAttributeTag iTag[]
);
```

Parameters

`iTextLayout`

An `ATSUTextLayout` value specifying the text layout object for which to restore default layout control attribute values.

`iTagCount`

An `ItemCount` value specifying the number of layout control attributes to restore to default values. This value should correspond to the number of elements in the `iTag` array. To restore all layout control attributes in the specified text layout object, pass the constant `kATSUClearAll` in this parameter. In this case, the value in the `iTag` parameter is ignored.

`iTag`

A pointer to the initial `ATSUAttributeTag` constant in an array of attribute tags. Each tag should identify a layout control attribute to restore to its default value. See "Attribute Tags" (page 196) for a description of the Apple-defined layout control attribute tag constants.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUClearLayoutControls` function removes those layout control attribute values identified by the tag constants in the `iTag` array and replaces them with the default values described in "Attribute Tags" (page 196). If you specify that any currently unset attribute values be removed, the function does not return an error.

To restore default values to line control attributes in a text layout object, call the function `ATSUClearLineControls` (page 29).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUClearLineControls

Restores default values to the specified line control attributes of a line in a text layout object.

```
OSStatus ATSUClearLineControls (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart,
   ItemCount iTagCount,
   const ATSUAttributeTag iTag[]
);
```

Parameters

`iTextLayout`

An `ATSUTextLayout` value specifying the text layout object containing the line for which to restore default line control attribute values.

`iLineStart`

A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the line for which to restore attribute values.

`iTagCount`

An `ItemCount` value specifying the number of line control attributes to restore to default values. This value should correspond to the number of elements in the `iTag` array. To restore all line control attributes of the specified line, pass the constant `kATSUClearAll` in this parameter. In this case, the value in the `iTag` parameter is ignored.

`iTag`

A pointer to the initial `ATSUAttributeTag` constant in an array of attribute tags. Each tag should identify a line control attribute to restore to its default value. See "Attribute Tags" (page 196) for a description of the Apple-defined line control attribute tag constants.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUClearLineControls` function removes those line control attribute values identified by the tag constants in the `iTag` array and replaces them with the default values described in "Attribute Tags" (page 196). If you specify that any currently unset attribute values be removed, the function does not return an error.

To restore default values to layout control attributes in a text layout object, call the function `ATSUClearLayoutControls` (page 28).

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

Declared In
`ATSUnicodeObjects.h`

ATSUClearSoftLineBreaks

Removes soft line breaks from a range of text.

```
OSStatus ATSUClearSoftLineBreaks (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iRangeStart,
   UniCharCount iRangeLength
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value specifying the text layout object for which to remove line breaks.

iRangeStart
A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the text range. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iRangeLength` parameter.

iRangeLength
A `UniCharCount` value specifying the length of the text range. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUClearSoftLineBreaks` function clears all previously set soft line breaks for the specified text range and clears any associated layout caches as well.

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

Declared In
`ATSUnicodeDrawing.h`

ATSUClearStyle

Restores default values to a style object.

```
OSStatus ATSUClearStyle (
   ATSUStyle iStyle
);
```

Parameters

`iStyle`

An `ATSUStyle` value specifying the style object for which to restore default values.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUClearStyle` function clears a style object of all style attributes (including any application-defined attributes), font features, and font variations and returns these values to their default settings. Default font variations and font features are defined by the font; default style attribute values are described in "Attribute Tags" (page 196). `ATSUClearStyle` does not remove reference constants.

To restore only default style attributes to a style object, you should call the function `ATSUClearAttributes` (page 24). To restore only font variations to a style object, call `ATSUClearFontVariations` (page 26). To restore only font features, call `ATSUClearFontFeatures` (page 25).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUCompareStyles

Compares the attribute values of two style objects.

```
OSStatus ATSUCompareStyles (
   ATSUStyle iFirstStyle,
   ATSUStyle iSecondStyle,
   ATSUStyleComparison *oComparison
);
```

Parameters

`iFirstStyle`

An `ATSUStyle` value specifying the first style object to compare.

`iSecondStyle`

An `ATSUStyle` value specifying the second style object to compare.

`oComparison`

A pointer to an `ATSUStyleComparison` value. On return, the value contains the results of the comparison and indicates whether the two style objects are the same, different, or one a subset of the another. See "Style Comparison Options" (page 229) for a description of possible values.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUCompareStyles` function compares the contents of two style objects, including their style attributes, font features, and font variations. It does not consider reference constants or application-defined style attributes in the comparison.

You can call `ATSUCompareStyles`, in conjunction with the function `ATSUGetAllAttributes` (page 59), to implement style sheets and tables of style runs.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUCopyAttributes

Copies all style attribute settings from a source style object to a destination style object.

```
OSStatus ATSUCopyAttributes (
   ATSUStyle iSourceStyle,
   ATSUStyle iDestinationStyle
);
```

Parameters

`iSourceStyle`

An `ATSUStyle` value specifying the style object from which to copy style attributes.

`iDestinationStyle`

An `ATSUStyle` value specifying the style object to set style attributes to.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUCopyAttributes` function copies all style attributes to a destination style object from a source style object, including any default values (those values not set by your application) in the source object. Default values for style attributes are described in "Attribute Tags" (page 196).

The `ATSUCopyAttributes` function does not copy the contents of memory referenced by pointers within custom style attributes or within reference constants. You are responsible for ensuring that this memory remains valid until both the source and destination style objects are disposed of.

To copy style attributes that are explicitly set in the source but not in the destination style object, call the function `ATSUUnderwriteAttributes` (page 138). To copy all style attributes that are explicitly set in the source object into the destination object, whether or not the destination object has its own settings for these values, call the function `ATSUOverwriteAttributes` (page 113).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUCopyLayoutControls

Copies all layout control attribute settings from a source text layout object to a destination text layout object.

```
OSStatus ATSUCopyLayoutControls (
   ATSUTextLayout iSourceTextLayout,
   ATSUTextLayout iDestTextLayout
);
```

Parameters

iSourceTextLayout

An `ATSUTextLayout` value specifying the text layout object from which to copy layout control attributes.

iDestTextLayout

An `ATSUTextLayout` value specifying the text layout object for which to set layout control attributes.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUCopyLayoutControls` function copies all layout control attribute values to a destination text layout object from a source text layout object, including any default (unset) values in the source object. Default values for unset layout control attributes are described in "Attribute Tags" (page 196).

`ATSUCopyLayoutControls` does not copy the contents of memory referenced by pointers within reference constants. You are responsible for ensuring that this memory remains valid until both the source and destination text layout objects are disposed.

To copy line control attribute values from one text layout object to another, call the function `ATSUCopyLineControls` (page 33).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUCopyLineControls

Copies line control attribute settings from a line in a source text layout object to a line in a destination text layout object.

```
OSStatus ATSUCopyLineControls (
   ATSUTextLayout iSourceTextLayout,
   UniCharArrayOffset iSourceLineStart,
   ATSUTextLayout iDestTextLayout,
   UniCharArrayOffset iDestLineStart
);
```

Parameters

iSourceTextLayout

An `ATSUTextLayout` value specifying the text layout object from which to copy line control attributes.

iSourceLineStart
: A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the line from which to copy control attributes.

iDestTextLayout
: An `ATSUTextLayout` value specifying the text layout object for which to set line control attributes. This can be the same text layout object passed in the `iSourceTextLayout` parameter if you want to copy line control attributes from one line to another within a text layout object.

iDestLineStart
: A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the line for which to set control attributes.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCopyLineControls` function copies all line control attribute values to a line in a destination text layout object from a line in a source text layout object, including any default (unset) values in the source line. Unset line control attributes are assigned the default values described in "Attribute Tags" (page 196).

`ATSUCopyLineControls` does not copy the contents of memory referenced by pointers within reference constants. You are responsible for ensuring that this memory remains valid until the source text layout object is disposed.

To copy layout control attributes from one text layout object to another, call the function `ATSUCopyLayoutControls` (page 33).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUCountFontFeatureSelectors

Obtains the number of available feature selectors for a given feature type in a font.

```
OSStatus ATSUCountFontFeatureSelectors (
   ATSUFontID iFontID,
   ATSUFontFeatureType iType,
   ItemCount *oSelectorCount
);
```

Parameters

iFont
: An `ATSUFontID` value identifying the font to examine.

iType
: An `ATSUFontFeatureType` value specifying one of the font's supported feature types. To obtain the available feature types for a font, call the function `ATSUGetFontFeatureTypes` (page 70).

oSelectorCount
: A pointer to an `ItemCount` value. On return, the value specifies the actual number of feature selectors defined for the feature type by the font.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCountFontFeatureSelectors` function obtains the total number of feature selectors defined for a given feature type in the font. You can use the count produced by `ATSUCountFontFeatureSelectors` to determine how much memory to allocate for the `oSelectors` array in the function `ATSUGetFontFeatureSelectors` (page 68).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUCountFontFeatureTypes

Obtains the number of available feature types in a font.

```
OSStatus ATSUCountFontFeatureTypes (
   ATSUFontID iFontID,
   ItemCount *oTypeCount
);
```

Parameters

`iFont`
An `ATSUFontID` value identifying the font to examine.

`oTypeCount`
A pointer to an `ItemCount` value. On return, the value specifies the actual number of feature types defined for the font.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCountFontFeatureTypes` function obtains the total number of feature types defined for a font. You can use the count produced by `ATSUCountFontFeatureTypes` to determine how much memory to allocate for the `oTypes` array in the function `ATSUGetFontFeatureTypes` (page 70).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUCountFontInstances

Obtains the number of defined font instances in a font.

```
OSStatus ATSUCountFontInstances (
   ATSUFontID iFontID,
   ItemCount *oInstances
);
```

Parameters

iFont

An `ATSUFontID` value identifying the font to examine.

oInstances

A pointer to an `ItemCount` value. On return, the value specifies the number of font instances defined for the font.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUCountFontInstances` function obtains the total number of font instances defined in a font. You can use an index value derived from this count to get information about a specific font instance by calling the function `ATSUGetFontInstance` (page 72).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFonts.h`

ATSUCountFontNames

Obtains the number of font names that correspond to a given ATSUI font ID.

```
OSStatus ATSUCountFontNames (
   ATSUFontID iFontID,
   ItemCount *oFontNameCount
);
```

Parameters

iFontID

An `ATSUFontID` value specifying the font to examine.

oFontNameCount

A pointer to an `ItemCount` value. On return, the value specifies the number of entries in the font name table corresponding to the given ATSUI font ID.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUCountFontNames` function obtains the number of font names defined in a font name table for a given ATSUI font ID. This number includes repetitions of the same name in different platforms, languages, and scripts; names of font features, variations, tracking settings, and instances for the font; and font names identified by name code constants.

You can pass an index value based on this count to the function `ATSUGetIndFontName` (page 78) to obtain a name string, name code, platform, script, and language for a given ATSUI font ID.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUCountFontTracking

Obtains the number of entries in the font tracking table that correspond to a given ATSUI font ID and glyph orientation.

```
OSStatus ATSUCountFontTracking (
   ATSUFontID iFontID,
   ATSUVerticalCharacterType iCharacterOrientation,
   ItemCount *oTrackingCount
);
```

Parameters

iFont

An `ATSUFontID` value specifying the font to examine.

iCharacterOrientation

An `ATSUVerticalCharacterType` constant identifying the glyph orientation of the font tracking entries, for example `kATSUStronglyHorizontal` or `kATSUStronglyVertical`. See "Vertical Character Types" (page 234) for a description of possible values.

oTrackingCount

A pointer to an `ItemCount` value. On return, the value specifies the number of entries in the font tracking table corresponding to the given ATSUI font ID and glyph orientation.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCountFontTracking` function obtains the number of font tracking entries defined in a font tracking table for a given ATSUI font ID and glyph orientation. You can pass an index value based on this count to the function `ATSUGetIndFontTracking` (page 80) to obtain the name code and tracking value of a font tracking.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUCountFontVariations

Obtains the number of defined variation axes in a font.

```
OSStatus ATSUCountFontVariations (
   ATSUFontID iFontID,
   ItemCount *oVariationCount
);
```

Parameters

iFont

An ATSUFontID value identifying the font to examine.

oVariationCount

A pointer to an ItemCount value. On return, the value specifies the number of variation axes defined for the font.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The ATSUCountFontVariations function obtains the total number of variation axes defined for a font. You can use the count produced by ATSUCountFontVariations to get information about a specific font variation axis from the function ATSUGetIndFontVariation (page 81).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

ATSUnicodeFonts.h

ATSUCreateAndCopyStyle

Creates a copy of a style object.

```
OSStatus ATSUCreateAndCopyStyle (
   ATSUStyle iStyle,
   ATSUStyle *oStyle
);
```

Parameters

iStyle

An ATSUStyle value specifying the style object to copy.

oStyle

A pointer to an ATSUStyle value. On return, the pointer refers to a newly created style object. This style object contains the same values for style attributes, font features, and font variations as those of the style object passed in the iStyle parameter.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The ATSUCreateAndCopyStyle function creates a new style object with values obtained from the source style object's style attributes, font features, and font variations. ATSUCreateAndCopyStyle does not copy reference constants.

To create a new style object without copying a source object, you can call the function `ATSUCreateStyle` (page 40). Alternately, to copy the contents of a source style object into an existing style object, call the function `ATSUCopyAttributes` (page 32).

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

Declared In
`ATSUnicodeObjects.h`

ATSUCreateAndCopyTextLayout

Creates a copy of a text layout object.

```
OSStatus ATSUCreateAndCopyTextLayout (
   ATSUTextLayout iTextLayout,
   ATSUTextLayout *oTextLayout
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object to copy.

`oTextLayout`
A pointer to an `ATSUTextLayout` value. On return, the pointer refers to a newly created text layout object containing the contents of the text layout object in the `iTextLayout` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCreateAndCopyTextLayout` function creates a copy of the source text layout object's style runs (including references to the associated text buffer and style objects), line attributes, layout attributes, and layout caches. `ATSUCreateAndCopyTextLayout` does not copy reference constants.

To create a text layout object without copying a source object, you can the function `ATSUCreateTextLayout` (page 41) or the function `ATSUCreateTextLayoutWithTextPtr` (page 42).

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

Declared In
`ATSUnicodeObjects.h`

ATSUCreateFontFallbacks

Creates an opaque object that can be set to contain a font list and a font-search method.

```
OSStatus ATSUCreateFontFallbacks (
   ATSUFontFallbacks *oFontFallback
);
```

Parameters

oFontFallback

A pointer to an `ATSUFontFallbacks` value. On return, the pointer refers to a newly created font fallback object.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUCreateFontFallbacks` function creates an "empty" font fallback object, which can be used to define ATSUI's search behavior when seeking substitute fonts for a text layout object. Font fallback objects are thread safe and can be shared among threads.

You set the font list and search method for the font fallback object by calling the function `ATSUSetObjFontFallbacks` (page 126). To associate the font fallback object with a text layout object, call either of the functions `ATSUSetLayoutControls` (page 122) or `ATSUSetLineControls` (page 124). You pass these functions the control attribute value `kATSULineFontFallbacksTag` to set the font fallback object.

Similarly to a style object, a font fallback object can be used with any number of text layout objects. While it is innately more efficient to reuse font fallback objects, instead of repeatedly creating (and destroying) them, there is another reason to share a given font fallback object among text layout objects. That is, as a font fallback object is used, it continues to amass data about the system's fonts and which are best applied to the various ranges of Unicode. Therefore, for best performance, once you create a font fallback object, you should keep it and use it as often as needed.

You should dispose of a font fallback object only when it is no longer needed in your application. To dispose of the memory associated with a font fallback object, call the function `ATSUDisposeFontFallbacks` (page 48).

Availability

Available in Mac OS X v10.1 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUCreateStyle

Creates an opaque style object containing only default style attributes, font features, and font variations.

```
OSStatus ATSUCreateStyle (
   ATSUStyle *oStyle
);
```

Parameters

oStyle

A pointer to an `ATSUStyle` value. On return, the pointer refers to an empty style object.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCreateStyle` function creates a style object containing only default values for style attributes, font features, and font variations. The default values for the font features and variations are assigned by the font. The default style attribute values are described in "Attribute Tags" (page 196).

To make changes to the default style attribute values, you can call the function `ATSUSetAttributes` (page 119). To set font features and font variations, call the functions `ATSUSetFontFeatures` (page 120) and `ATSUSetVariations` (page 133), respectively. You can also use the function `ATSUCreateAndCopyStyle` (page 38) to create a new style object by copying all the settings from an existing one.

For ATSUI to apply your selected character-style information, you must associate the style object with a text run in a text layout object. A text run consists of one or more characters that are contiguous in memory. If you associate these characters with a distinct style, you define a style run. You can use the function `ATSUSetRunStyle` (page 127) to define a style run by associating a style object with a run of text in a text layout object. Or, to create a text layout object and associate style objects with it at the same time, you can call the function `ATSUCreateTextLayoutWithTextPtr` (page 42). In either case, each text run in a text layout object must be assigned a style object, which may or may not differ from other style objects assigned to other text runs in the text layout object.

Style objects are readily reusable and should be cached for later use, if possible. You can create a style object once and then use it for as many text layout objects as appropriate. Style objects are thread-safe starting with ATSUI version 2.3.

Note that you are responsible for disposing of the memory allocated for the style object. However, you should dispose of any text layout objects with which the style object is associated prior to disposing of the style object itself. To dispose of a style object, call the function `ATSUDisposeStyle` (page 49).

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

Declared In
`ATSUnicodeObjects.h`

ATSUCreateTextLayout

Creates an opaque text layout object containing only default text layout attributes.

```
OSStatus ATSUCreateTextLayout (
   ATSUTextLayout *oTextLayout
);
```

Parameters

`oTextLayout`
A valid pointer to an `ATSUTextLayout` value. On return, the value refers to an empty text layout object.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCreateTextLayout` function creates a text layout object containing only the default text layout attributes described in "Attribute Tags" (page 196). The resulting text layout object is associated with neither text nor style objects. However, most ATSUI functions that operate on text layout objects require that the objects be associated with style information and text. To associate style objects and text with an empty text

layout object, you can call the functions ATSUSetRunStyle (page 127) and ATSUSetTextPointerLocation (page 131). Or, to create a text layout object and associate style objects and text with it at the same time, you can call the function ATSUCreateTextLayoutWithTextPtr (page 42).

To provide nondefault line or layout attributes for a text layout object, you can call the functions ATSUSetLineControls (page 124) or ATSUSetLayoutControls (page 122). After setting text attributes, call ATSUDrawText (page 50) to draw the text.

Text layout objects are readily reusable and should be cached for later use, if possible. You can reuse a text layout object even if the text associated with it is altered. Call the functions ATSUSetTextPointerLocation (page 131), ATSUTextDeleted (page 135), or ATSUTextInserted (page 136) to manage the altered text.

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

Declared In
`ATSUnicodeObjects.h`

ATSUCreateTextLayoutWithTextPtr

Creates an opaque text layout object containing default text layout attributes as well as associated text and text styles.

```
OSStatus ATSUCreateTextLayoutWithTextPtr (
   ConstUniCharArrayPtr iText,
   UniCharArrayOffset iTextOffset,
   UniCharCount iTextLength,
   UniCharCount iTextTotalLength,
   ItemCount iNumberOfRuns,
   const UniCharCount iRunLengths[],
   ATSUStyle iStyles[],
   ATSUTextLayout *oTextLayout
);
```

Parameters

iText
A pointer of type `ConstUniCharArrayPtr`, referring to a text buffer containing UTF-16–encoded text. ATSUI associates this buffer with the new text layout object and analyzes the complete text of the buffer when obtaining the layout context for the current text range. Thus, for paragraph-format text, if you specify a buffer containing less than a complete paragraph, some of ATSUI's layout results are not guaranteed to be accurate. For example, with a buffer of less than a full paragraph, ATSUI can neither reliably obtain the context for bidirectional processing nor reliably generate accent attachments and ligature formations for Roman text.

iTextOffset
A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range to include in the layout. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iTextLength` parameter.

iTextLength
: A `UniCharCount` value specifying the length of the text range. Note that `iTextOffset + iTextLength` must be less than or equal to the value of the *iTextTotalLength* parameter. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

iTextTotalLength
: A `UniCharCount` value specifying the length of the entire text buffer. This value should be greater than or equal to the range of text defined by the `iTextLength` parameter.

iNumberOfRuns
: An `ItemCount` value specifying the number of text style runs you want to define within the overall text range. The number of style objects and style run lengths passed in the `iStyles` and `iRunLengths` parameters, respectively, should be equal to the number of runs specified here.

iRunLengths
: A pointer to the first element in a `UniCharCount` array. This array provides ATSUI with the lengths of each of the text's style runs. You can pass `kATSUToTextEnd` for the last style run length if you want the style run to extend to the end of the text range. If the sum of the style run lengths is less than the total length of the text range, the remaining characters are assigned to the last style run.

iStyles
: A pointer to the first element in an `ATSUStyle` array. Each element in the array must contain a valid style object that corresponds to a style run defined by the `iRunLengths` array.

oTextLayout
: A valid pointer to an `ATSUTextLayout` value. On return, the value refers to the newly created text layout object.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUCreateTextLayoutWithTextPtr` function creates a text layout object associated with style objects and text and containing the default text layout attributes described in "Attribute Tags" (page 196). To provide nondefault line or layout attributes for a text layout object, you can call the functions `ATSUSetLineControls` (page 124) or `ATSUSetLayoutControls` (page 122). After setting text attributes, call `ATSUDrawText` (page 50) to draw the text.

Because the only way that ATSUI interacts with text is via the memory references you associate with a text layout object, you are responsible for keeping these references updated, as in the following cases:

1. When the user deletes or inserts a subrange within a text buffer (but the buffer itself is not relocated), you should call the functions `ATSUTextDeleted` (page 135) and `ATSUTextInserted` (page 136), respectively.

2. When you relocate the entire text buffer (but no other changes have occurred that would affect the buffer's current subrange), you should call the function `ATSUTextMoved` (page 137).

3. When both the buffer itself is relocated and a subrange of the buffer's text is deleted or inserted (that is, a combination of cases 1 and 2, above), you must use the function `ATSUSetTextPointerLocation` (page 131) to inform ATSUI.

4. When you are associating an entirely different buffer with a text layout object, you must call the function `ATSUSetTextPointerLocation` (page 131).

Note that, because ATSUI objects retain state information, doing superfluous calling can degrade performance. For example, you could call `ATSUSetTextPointerLocation` rather than `ATSUTextInserted` when the user inserts text, but there would be a performance penalty, as all the layout caches are flushed when you call `ATSUSetTextPointerLocation`, rather than just the affected ones.

Text layout objects are readily reusable and should themselves be cached for later use, if possible. Text objects are thread-safe starting with ATSUI version 2.4.

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

Declared In
`ATSUnicodeObjects.h`

ATSUDirectAddStyleSettingRef

Looks up, and if necessary, adds a style setting to a line of text.

```
OSStatus ATSUDirectAddStyleSettingRef (
   ATSULineRef iLineRef,
   ATSUStyleSettingRef iStyleSettingRef,
   UInt16 *oStyleIndex
);
```

Parameters

`iLineRef`
: An `ATSULineRef` value that specifies the line of text to which you want to add a style setting. You should pass the same reference provided as a parameter to your `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) callback function.

`iStyleSettingRef`
: An `ATSUStyleSettingRef` value that specifies the style setting you want ATSUI to look up or add to the text layout object referenced by the line starting at the offset `iLineOffset`.

`oStyleIndex`
: On return, points to the index of the `ATSUStyleSettingRef` passed in `iStyleSettingRef` for the line referenced by `iLineRef`. If the `ATSUStyleSettingRef` does not exist in that context, ATSUI adds it and returns the index value.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The function `ATSUDirectAddStyleSettingRef` checks to see if a line of text has a specified style setting reference associated with it. If the style setting reference is not associated with the line of text, ATSUI adds the style setting reference.

You must call this function from within an `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) callback function. You can use the function `ATSUDirectAddStyleSettingRef` to replace or substitute glyphs. For example, you can check a line of text for a specific character, such as a whitespace character. When your application finds a whitespace character, it can call the function `ATSUDirectAddStyleSettingRef` to set style attributes that achieve the desired effect.

Do not call this function if you obtained an `ATSUStyleSettingRef` array for the line specified by `iLineRef` and have not yet disposed of the pointer to this array by calling the function `ATSUDirectReleaseLayoutDataArrayPtr` (page 48), as the pointer is not guaranteed to be valid after you call the function `ATSUDirectAddStyleSettingRef`.

Availability
Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeDirectAccess.h`

ATSUDirectGetLayoutDataArrayPtrFromLineRef

Obtains the glyph data specified by a direct-data selector and for a specific line of text.

```
OSStatus ATSUDirectGetLayoutDataArrayPtrFromLineRef (
   ATSULineRef iLineRef,
   ATSUDirectDataSelector iDataSelector,
   Boolean iCreate,
   void *oLayoutDataArrayPtr[],
   ItemCount *oLayoutDataCount
);
```

Parameters

`iLineRef`
An `ATSULineRef` value that specifies the line of text whose data you want to obtain. You should pass the same `ATSULineRef` value passed to the `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) callback function from which you are calling this function.

`iDataSelector`
A direct-data selector constant that specifies the data you want to obtain. You can pass any of the constants described in "Direct Data Selectors" (page 210).

`iCreate`
A `Boolean` value that specifies whether to create an array if one does not already exist. Pass `true` if you want an array created. If the line referenced by the `iLineRef` parameter does not already have an array created that contains the data specified by the `iDataSelector` parameter, then ATSUI creates a zero-filled array and returns the array in the `oLayoutDataArray` parameter. The `iCreate` parameter has no effect for some data specified by the direct-data selector. See "Direct Data Selectors" (page 210) for details.

`oLayoutDataArrayPtr[]`
On return, points to an array that contains the data specified by the `iDataSelector` parameter. The data is for the line of text referenced by the `iLineRef` parameter. If an array for the specified data does not exist, and if the `iCreate` is set to `false`, ATSUI returns `NULL`. If an array for the specified data does not exist, and if the `iCreate` is set to `true`, ATSUI creates a zero-filled array. You can pass `NULL` if you only want to obtain the number of entries in the array returned in the *`oLayoutDataArray`* array.

`oLayoutDataCount`
On return, the number of entries in the array returned in the *`oLayoutDataArray`* array.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The function ATSUDirectGetLayoutDataArrayPtrFromLineRef returns the data pointer specified by the iDataSelector parameter and referenced by the iLineRef parameter. You must call this function from within an ATSUDirectLayoutOperationOverrideProcPtr (page 164) callback function. You must only release the data pointer by calling the function ATSUDirectReleaseLayoutDataArrayPtr (page 48). When you call this function, it signals ATSUI that you are done with the data and that ATSUI can merge your modifications with the font's data. If you do not properly free the data by calling the function ATSUDirectReleaseLayoutDataArrayPtr, a memory leak may result.

The data you obtain is the actual data used by ATSUI in its layout process; it is not a copy. This function is very efficient because ATSUI does not need to allocate memory and copy data. Furthermore, because you obtain a pointer to the data that ATSUI uses for its layout, any modifications you make to the data effect the final layout.

Many of the data arrays you can request are created by ATSUI only when necessary. If you plan to alter the data in an array, make sure you set the iCreate parameter to true. This ensures that the array is created. If an arrays are not created, ATSUI assumes all entries in the array are zero.

The pointer returned by this function is only valid within the context of the ATSUDirectLayoutOperationOverrideProcPtr callback function. You must not retain it for later use.

Availability
Available in Mac OS X v10.2 and later.

Declared In
ATSUnicodeDirectAccess.h

ATSUDirectGetLayoutDataArrayPtrFromTextLayout

Obtains a copy of the glyph data specified by a direct-data selector and for a specific line of text in a text layout object.

```
OSStatus ATSUDirectGetLayoutDataArrayPtrFromTextLayout (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineOffset,
   ATSUDirectDataSelector iDataSelector,
   void *oLayoutDataArrayPtr[],
   ItemCount *oLayoutDataCount
);
```

Parameters

iTextLayout
An ATSUTextLayout value that specifies the text layout object whose data you want to obtain.

iLineOffset
The edge offset that corresponds to the beginning of the line of text whose data you want to obtain.

iDataSelector
A direct-data selector constant that specifies the data you want to obtain. You can pass any of the constants described in "Direct Data Selectors" (page 210).

oLayoutDataArrayPtr[]
: On return, points to an array that contains the data specified by the `iDataSelector` parameter. The data is for the line of text referenced by the `iLineOffset` parameter. If an array for the specified data does not exist, ATSUI returns `NULL`. You can pass `NULL` if you only want to obtain the number of entries in the array in the *oLayoutDataArray* array.

oLayoutDataCount
: On return, the number of entries in the array `oLayoutDataArray`.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The function `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` returns a pointer to the data specified by `iDataSelector` and referenced by `iTextLayout` for the line starting at `iLineOffset`. You must not call this function from within an `ATSUDirectLayoutOperationOverrideProcPtr` (page 164)callback function.

You should only release the data pointer by calling the function `ATSUDirectReleaseLayoutDataArrayPtr`. When you call this function, it signals ATSUI that you are done with the data and that ATSUI can merge your modifications with the font's data. If you do not properly free the data by calling the function `ATSUDirectReleaseLayoutDataArrayPtr`, a memory leak may result.

The data you obtain is a copy of the data ATSUI uses for its layout processes. This means the following:

- Obtaining data through a copy operation takes more time than obtaining the actual data. This function returns in order-n time instead of in a constant time.
- Changing any of the data values has no effect on the layout.

Before you use this function, you should consider using the function`ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) with the `kATSULayoutOperationPostLayoutAdjustment` selector.

If you use the function `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` to obtain the `ATSUStyleSettingRef` array, the structures referenced by each element of the array are invalid after you call the function `ATSUDirectReleaseLayoutDataArrayPtr` to release the array. If want to retain one or more of the elements in the `ATSUStyleSettingRef` array for later use, you must not call the function `ATSUDirectReleaseLayoutDataArrayPtr` until all operations that use the elements in the `ATSUStyleSettingRef` in the array are complete. The elements in the `ATSUStyleSettingRef` array are valid only within the context of the callback from which they were obtained

Many of the requested data arrays are created by ATSUI only when necessary. This means that it's possible for the function `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` to return a `NULL` pointer and a count of `0`. If this is case and if the function does not return an error, the array doesn't exist. You should interpret this result to mean that all values in the array are `0`.

Availability
Available in Mac OS X v10.2 and later.
Not available to 64-bit applications.

Declared In
`ATSUnicodeDirectAccess.h`

ATSUDirectReleaseLayoutDataArrayPtr

Releases a pointer to a direct-data array.

```
OSStatus ATSUDirectReleaseLayoutDataArrayPtr (
   ATSULineRef iLineRef,
   ATSUDirectDataSelector iDataSelector,
   void *iLayoutDataArrayPtr[]
);
```

Parameters

iLineRef

An `ATSULineRef` value that specifies the line of text whose data is pointed to by the `iLayoutDataArrayPtr` parameter. Pass `NULL` if you did not obtain the layout data array pointer using a `lineRef`.

iDataSelector

A direct-data selector constant that specifies the data pointed to by the `iLayoutDataArrayPtr` parameter. You can pass any of the constants described in "Direct Data Selectors" (page 210).

iLayoutDataArrayPtr[]

A pointer to the layout data array of which you want to dispose.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

You must call the function `ATSUDirectReleaseLayoutDataArrayPtr` when you no longer need the direct-data pointer you obtained from the `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) functions. You must dispose of the pointer to inform ATSUI you no longer need the data and to allow for ATSUI to make any internal adjustments prior to completing the layout process.

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeDirectAccess.h`

ATSUDisposeFontFallbacks

Disposes of the memory associated with a font fallback object.

```
OSStatus ATSUDisposeFontFallbacks (
   ATSUFontFallbacks iFontFallbacks
);
```

Parameters

iFontFallbacks

An `ATSUFontFallbacks` value specifying the font fallback object to dispose. See the `ATSUFontFallbacks` data type.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUDisposeFontFallbacks` function frees the memory associated with the specified font fallback object and its internal structures.

For best performance, once you create a font fallback object, you should keep it and use it as often as needed. You should dispose of the font fallback object only when it is no longer needed in your application.

Availability
Available in Mac OS X v10.1 and later.
Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUDisposeStyle

Disposes of the memory associated with a style object.

```
OSStatus ATSUDisposeStyle (
   ATSUStyle iStyle
);
```

Parameters

`iStyle`
An `ATSUStyle` value specifying the style object to dispose of.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUDisposeStyle` function frees the memory associated with the specified style object and its internal structures, including style run attributes. It does not dispose of the memory pointed to by application-defined style run attributes or reference constants. You are responsible for doing so.

You should call this function after calling the function `ATSUDisposeTextLayout` (page 49) to dispose of any text layout objects associated with the style object.

For best performance, once you create a style object, you should keep it and use it as often as needed. You should dispose of the style object only when it is no longer needed in your application.

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

Declared In
`ATSUnicodeObjects.h`

ATSUDisposeTextLayout

Disposes of the memory associated with a text layout object.

```
OSStatus ATSUDisposeTextLayout (
   ATSUTextLayout iTextLayout
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value specifying the text layout object to dispose of.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUDisposeTextLayout` function frees the memory associated with the specified text layout object and its internal structures, including line and layout control attributes, style runs, and soft line breaks. `ATSUDisposeTextLayout` does not dispose of any memory that may be allocated for reference constants or style objects associated with the text layout object. You are responsible for doing so.

For best performance, text layout objects are readily reusable and should be cached for later use, if possible. You can reuse a text layout object even if the text associated with it is altered. Call the functions `ATSUSetTextPointerLocation` (page 131), `ATSUTextDeleted` (page 135), or `ATSUTextInserted` (page 136) to manage the altered text, rather than disposing of the text layout object and creating a new one.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeObjects.h`

ATSUDrawText

Renders a range of text at a specified location in a QuickDraw graphics port or Quartz graphics context.

```
OSStatus ATSUDrawText (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineOffset,
   UniCharCount iLineLength,
   ATSUTextMeasurement iLocationX,
   ATSUTextMeasurement iLocationY
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value identifying the text layout object for which to render text.

iLineOffset

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range to render. The function `ATSUDrawText` renders text to the first soft line break it encounters. If the range of text spans multiple lines, you should call `ATSUDrawText` for each line, passing the offset corresponding to the beginning of the new line to draw with each call. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iLineLength` parameter.

iLineLength

A `UniCharCount` value specifying the length of the text range to render. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`. Keep in mind that the function `ATSUDrawText` renders text one line at a time. If the range of text spans multiple lines, you must call `ATSUDrawText` for each line.

iLocationX

An `ATSUTextMeasurement` value specifying the x-coordinate of the origin (in either the current QuickDraw graphics port or in a Quartz graphics context) of the line containing the text range to render. Note that the `ATSUTextMeasurement` type is defined as a `Fixed` value, so you must ensure that your coordinates are converted to `Fixed` values before passing them to this function. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to draw relative to the current pen location in the current graphics port.

iLocationY

An `ATSUTextMeasurement` value specifying the y-coordinate of the origin (in either the current graphics port or Quartz graphics context) of the line containing the text range to render. Note that the `ATSUTextMeasurement` type is defined as a `Fixed` value, so you must ensure that your coordinates are converted to `Fixed` values before passing them to this function. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to draw relative to the current pen location in the current graphics port.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUDrawText` function renders a range of text at a specified location in a QuickDraw graphics port or Quartz graphics context. This function renders text to the first soft line break it encounters. If you draw into a QuickDraw graphics port you get the best performance by using a bit depth of 16 bits. If you use bit depths of 1, 4, or 8, your application incurs a performance penalty.

You typically call the `ATSUDrawText` function every time you need to draw or redraw unhighlighted text. To draw highlighted text, call the function `ATSUHighlightText` (page 101).

`ATSUDrawText` uses the transfer mode and resolution that are set in the graphics port or graphics context. If you explicitly set in the style object, then text color is taken from the style object, and the value in the graphics port/context is ignored. If the text color was not explicitly set in the style object, `ATSUDrawText` uses the graphics port/context setting.

`ATSUDrawText` examines the text layout object to ensure that each of the characters in the range is assigned to a style run. If there are gaps between style runs, ATSUI assigns the characters in the gap to the style run that precedes (in storage order) the gap. If there is no style run at the beginning of the text range, ATSUI assigns these characters to the first style run it finds. If there is no style run at the end of the text range, ATSUI assigns the remaining characters to the last style run it finds.

If you want to draw a range of text that spans multiple lines, you should call `ATSUDrawText` for each line of text to draw, even if all the lines are in the same text layout object. You should adjust the `iLineOffset` parameter to reflect the beginning of each line to be drawn.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeDrawing.h`

ATSUFindFontFromName

Obtains an ATSUI font ID for the first entry in a name table that matches the specified name string, name code, platform, script, and/or language.

```
OSStatus ATSUFindFontFromName (
   const void *iName,
   ByteCount iNameLength,
   FontNameCode iFontNameCode,
   FontPlatformCode iFontNamePlatform,
   FontScriptCode iFontNameScript,
   FontLanguageCode iFontNameLanguage,
   ATSUFontID *oFontID
);
```

Parameters

iName

A string that specifies the font name whose ATSUI font ID you want to obtain. The string that you pass must be appropriate for the value you pass in the `iFontNameCode` parameter. For example, if the `iFontNameCode` parameter is `kFontPostscriptName`, then you would supply a string that specifies the PostScript name of the font.

iNameLength

A `ByteCount` value specifying the length of the font name string provided in the *`iName`* parameter.

iFontNameCode

The `FontNameCode` value of the font name for which to obtain an ATSUI font ID. The `FontNameCode` is a `UInt32` data type, and it is defined in the `SFNTTypes.h` header file. You can supply any of the following constants, `kFontCopyrightName`, `kFontFamilyName`, `kFontStyleName`, `kFontUniqueName`, `kFontFullName`, `kFontVersionName`, `kFontPostscriptName`, `kFontTrademarkName`, `kFontManufacturerName`, `kFontDesignerName`, `kFontDescriptionName`, `kFontVendorURLName`, `kFontDesignerURLName`, `kFontLicenseDescriptionName`,or `kFontLicenseInfoURLName`.

iFontNamePlatform

A `FontPlatformCode` value specifying the encoding of the font name, for example, `kFontUnicodePlatform` (for UTF-16), `kFontMacintoshPlatform`, `kFontReservedPlatform`, `kFontMicrosoftPlatform`, or `kFontCustomPlatform`. If you pass the `kFontNoPlatformCode` constant, `ATSUFindFontFromName` produces the first font in the name table matching the other specified parameters. See the `SFNTTypes.h` header file for a definition of the `FontPlatformCode` type and a list of possible values.

iFontNameScript

A `FontScriptCode` value specifying the script code of the font name, for example, `kFontRomanScript`. Pass `kFontNoScriptCode` if you supplied the `kFontUnicodePlatform` constant for the `iFontNamePlatform` parameter. If you pass the `kFontNoScriptCode` constant, `ATSUFindFontFromName` produces the first font in the name table matching the other specified parameters. See the `SFNTTypes.h` header file for a definition of the `FontScriptCode` type and a list of possible values.

iFontNameLanguage

A `FontLanguageCode` value specifying the language of the font name, for example, `kFontNorwegianLanguage`. Pass `kFontNoLanguageCode` if you supplied the `kFontUnicodePlatform` constant for the `iFontNamePlatform` parameter. If you pass the `kFontNoLanguageCode` constant, `ATSUFindFontFromName` produces the first font in the name table matching the other specified parameters. See the `SFNTTypes.h` header file for a definition of the `FontLanguageCode` type and a list of possible values.

oFontID

On return, points to the unique identifier for the specified font that matches the specified name string, name code, platform, script, and/or language. Note that because Apple Type Services assigns `ATSUFontID` values systemwide at runtime, font IDs can change across system restarts.

Return Value

A result code. If no installed font matches the specified parameters, `ATSUFindFontFromName` produces the constant `kATSUInvalidFontID` and returns the result code `kATSUInvalidFontErr`. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUFindFontFromName` function obtains an ATSUI font ID for the first font that matches the specified name string, name code, platform, script, and/or language. Because ATSUI cannot guarantee the uniqueness of names among installed fonts, `ATSUFindFontFromName` does not necessarily find the only font ID that matches these parameters. As a result, you may want to create a more sophisticated name-matching algorithm or guarantee the uniqueness of names among installed fonts.

To find a name string and index value for the first font in a name table that matches an ATSUI font ID and the specified font parameters, call the function `ATSUFindFontName` (page 53).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFonts.h`

ATSUFindFontName

Obtains a name string and index value for the first font in a name table that matches the specified ATSUI font ID, name code, platform, script, and/or language.

```
OSStatus ATSUFindFontName (
   ATSUFontID iFontID,
   FontNameCode iFontNameCode,
   FontPlatformCode iFontNamePlatform,
   FontScriptCode iFontNameScript,
   FontLanguageCode iFontNameLanguage,
   ByteCount iMaximumNameLength,
   Ptr oName,
   ByteCount *oActualNameLength,
   ItemCount *oFontNameIndex
);
```

Parameters

iFontID

The `ATSUFontID` value of the font for which to obtain a name string. Note that because Apple Type Services assigns `ATSUFontID` values systemwide at runtime, font IDs can change across system restarts.

iFontNameCode

The `FontNameCode` value of the font for which to obtain a name string. The `FontNameCode` is a `UInt32` data type, and it is defined in the `SFNTTypes.h` header file.

iFontNamePlatform

A `FontPlatformCode` value specifying the encoding of the font, for example, `kFontUnicodePlatform`, `kFontMacintoshPlatform`, `kFontReservedPlatform`, `kFontMicrosoftPlatform`, or `kFontCustomPlatform`. If you pass the `kFontNoPlatformCode` constant, `ATSUFindFontName` produces the first font in the name table matching the other specified parameters. See the `SFNTTypes.h` header file for a definition of the `FontPlatformCode` type and a list of possible values.

iFontNameScript

A `FontScriptCode` value specifying the script code of the font, for example, `kFontRomanScript`. If you pass the `kFontNoScriptCode` constant, `ATSUFindFontName` produces the first font in the name table matching the other specified parameters. See the `SFNTTypes.h` header file for a definition of the `FontScriptCode` type and a list of possible values.

iFontNameLanguage

A `FontLanguageCode` value specifying the language of the font, for example, `kFontNorwegianLanguage`. If you pass the `kFontNoLanguageCode` constant, `ATSUFindFontName` produces the first font in the name table matching the other specified parameters. See the `SFNTTypes.h` header file for a definition of the `FontLanguageCode` type and a list of possible values.

iMaximumNameLength

A `ByteCount` value specifying the maximum length of the font name to obtain. Typically, this is equivalent to the size of the buffer that you have allocated in the `oName` parameter. To determine this length, see the Discussion.

oName

A pointer to a buffer. On return, the buffer contains the name string of the first font in the font name table matching your specified parameters. If the buffer you allocate is not large enough, `ATSUFindFontName` produces a partial string.

oActualNameLength

A pointer to a `ByteCount` value. On return, the value specifies the actual length of the complete name string. This may be greater than the value passed in the `iMaximumNameLength` parameter. You should check this value to ensure that you have allocated sufficient memory and therefore obtained the complete name string for the font.

oFontNameIndex

A pointer to an `ItemCount` value. On return, the value provides a 0-based index to the font name in the font name table.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUFindFontName` function obtains a name string and index value for the first font in a name table that matches the specified ATSUI font ID, name code, platform, script, and/or language.

Typically you use the `ATSUFindFontName` function by calling it twice, as follows:

1. Pass `NULL` for the *oName* and *oFontNameIndex* parameters, `0` for the *iMaximumNameLength* parameter, and valid values for the other parameters. `ATSUFindFontName` returns the length of the font name string in the `oActualNameLength` parameter.

2. Allocate enough space for a buffer of the returned size, then call the function again, passing a valid pointer to the buffer in the `oName` parameter. On return, the buffer contains the font name string.

To obtain an ATSUI font ID for the first font in a name table that matches the specified name string, name code, platform, script, and/or language, call the function `ATSUFindFontFromName` (page 52). To obtain the font name string, name code, platform, script, and language for the font that matches an ATSUI font ID and name table index, call the function `ATSUGetIndFontName` (page 78).

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

Declared In
`ATSUnicodeFonts.h`

ATSUFlattenStyleRunsToStream

Flattens ATSUI style-run data so that it can be saved to disk or passed (through the pasteboard) to another application.

```
OSStatus ATSUFlattenStyleRunsToStream (
   ATSUFlattenedDataStreamFormat iStreamFormat,
   ATSUFlattenStyleRunOptions iFlattenOptions,
   ItemCount iNumberOfRunInfo,
   const ATSUStyleRunInfo iRunInfoArray[],
   ItemCount iNumberOfStyleObjects,
   const ATSUStyle iStyleArray[],
   ByteCount iStreamBufferSize,
   void *oStreamBuffer,
   ByteCount *oActualStreamBufferSize
);
```

Parameters

`iStreamFormat`
: The format of the flattened data. There is only one format supported at this time, `'ustl'` so you must pass the constant `kATSUDataStreamUnicodeStyledText`.

`iFlattenOptions`
: The options you want to use to flatten the data. There are no options supported at this time, so you must pass the constant `kATSUFlattenOptionsNoOptionsMask`.

`iNumberOfRunInfo`
: The number of style run information structures passed in the `iRunInfoArray` parameter. If you pass `0`, ATSUI assumes there is only one style for the entire text block passed in the `oStreamBuffer` parameter. The flattened data format passed to the `iStreamFormat` parameter must support the use of one style.

`iRunInfoArray[]`
: An array of `ATSUStyleRunInfo` structures that describes the style runs to be flattened. This array must contain `iNumberOfRunInfo` entries. An `ATSUStyleRunInfo` structure contains an index into an array of unique ATSUI style objects (`ATSUStyle`) and the length of the run to which the style object applies. Each index in the `ATSUStyleRunInfo` structure must reference a valid `ATSUStyle` object passed in the `iStyleArray` parameter. You can pass `NULL`, only if `iNumberOfRunInfo` is set to zero.

`iNumberOfStyleObjects`
: The number of `ATSUStyle` objects in the array passed to the `iStyleArray` parameter. You must pass a value that is greater than `0`.

iStyleArray[]

An array of `ATSUStyle` objects to be flattened. You cannot pass `NULL`.

iStreamBufferSize

The size of the stream buffer, pointed to by the `oStreamBuffer` parameter. You can pass `0` only if the `iStreamBufferSize` parameter is set to `NULL`. If you are uncertain of the size of the array, see the Discussion.

oStreamBuffer

On input, a pointer to the data you want to flatten. On return, points to the flattened data. If you pass `NULL` for this parameter, no data is flattened. Instead, the size of the buffer is calculated by ATSUI and returned in `oActualStreamSize` parameter. See the Discussion for more details. You are responsible for allocating the text buffer passed in the `oStreamBuffer` parameter.

oActualStreamBufferSize

On return, the size of the data written to the `oStreamBuffer` parameter. You can pass `NULL` only if the `oStreamBuffer` parameter is not `NULL`.

Return Value

A result code. See "ATSUI Result Codes" (page 234). This function can also return `paramErr` if you pass invalid values for any of the parameters.

Discussion

The function `ATSUFlattenStyleRunsToStream` takes an array of `ATSUStyle` objects and style run information and flattens the data to the specified format. The style runs must all reference the same block of Unicode text (usually passed separately as text in the `'utxt'` format). The style runs must also be in ascending order relative to the text in the text block.

Typically you use the function `ATSUFlattenStyleRunsFromStream` by calling it twice, as follows:

1. Provide appropriate values for the `iStreamFormat`, `iFlattenOptions`, `iNumberOfRunInfo`, `iRunInfoArray`, `iNumberOfStyleObjects`, and `iStyleArray` parameters. Set `iStreamBufferSize` to `0`, `oStreamBuffer` to `NULL`, and pass a valid reference to a `ByteCount` variable in the `oActualStreamBufferSize` parameter. Call the function `ATSUFlattenStyleRunsToStream`. On return, `oActualStreamBufferSize` points to the size needed for the buffer.

2. Allocate an appropriately-sized buffer for the `oStreamBuffer` parameter and then call the function `ATSUFlattenStyleRunsToStream` a second time.

Availability

Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFlattening.h`

ATSUFONDtoFontID

Finds the ATSUI font ID that corresponds to a font family number, if one exists. (**Deprecated.** There is no replacement because FONDs are a QuickDraw concept and QuickDraw is deprecated.)

Not recommended.

```
OSStatus ATSUFONDtoFontID (
   short iFONDNumber,
   Style iFONDStyle,
   ATSUFontID *oFontID
);
```

Parameters

iFONDNumber
: The font family number of the ATSUI-compatible font for which to obtain an ATSUI font ID.

iFONDStyle
: The font family style of the font, if any. Style identifiers exist only for fonts that split a font family into subgroups.

oFontID
: A pointer to a `ATSUFontID` value. On return, the value provides a unique identifier for the specified font family number and style.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The function `ATSUFONDtoFontID` is not recommended for use. Instead, use the Font Manager functions that translate font family numbers to `FMFont` values, which are equivalent to `ATSUFontID` values. Font family numbers were used by QuickDraw to represent fonts to the Font Manager. Some of these fonts, even if compatible with ATSUI, may not have font IDs.

Note that Apple Type Services assigns `ATSUFontID` values systemwide at runtime. As a result, these font IDs can change when the system is restarted.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFonts.h`

ATSUFontCount

Obtains the number of ATSUI-compatible fonts installed on a user's system.

```
OSStatus ATSUFontCount (
   ItemCount *oFontCount
);
```

Parameters

oFontCount
: A pointer to an `ItemCount` value. On return, the value specifies the current number of ATSUI-compatible fonts installed on the user's system.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUFontCount` function obtains the number of fonts on a user's system that are compatible with ATSUI. Incompatible fonts include those that cannot be used to represent Unicode, the missing-character glyph font, and fonts whose names begin with a period or a percent sign. You can use the count produced in the `oFontCount` parameter to determine the amount of memory to allocate for the `oFontIDs` array in the function `ATSUGetFontIDs` (page 71).

It is important to note that the set of installed ATSUI-compatible fonts may change while your application is running. In Mac OS X, the set of installed fonts may change at any time. Although in Mac OS 9, fonts cannot be removed from the Fonts folder while an application other than the Finder is running, they can be removed from other locations, and it is possible for fonts to be added.

Additionally, just because the number of fonts stays the same between two successive calls to `ATSUFontCount`, this does not mean that the font lists are the same. It is possible for a font to be added and another removed between two successive calls to `ATSUFontCount`, leaving the total number unchanged.

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

Declared In
`ATSUnicodeFonts.h`

ATSUFontIDtoFOND

Finds the font family number that corresponds to an ATSUI font ID, if one exists. (**Deprecated.** There is no replacement because FONDs are a QuickDraw concept and QuickDraw is deprecated.)

Not recommended.

```
OSStatus ATSUFontIDtoFOND (
   ATSUFontID iFontID,
   short *oFONDNumber,
   Style *oFONDStyle
);
```

Parameters

`iFontID`
The `ATSUFontID` value of the font for which to obtain a font family number. Note that because Apple Type Services assigns `ATSUFontID` values systemwide at runtime, font IDs can change across system restarts.

`oFONDNumber`
A pointer to a signed sixteen-bit integer. On return, the value identifies the font family number corresponding to the specified ATSUI font ID.

`oFONDStyle`
A pointer to a `Style` value. On return, the value identifies the font family style of the font, if any. Style identifiers exist only for fonts that split a font family into subgroups.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The function `ATSUFontIDtoFOND` is not recommended for use. Instead, use the Font Manager functions that translate `FMFont` values, which are equivalent to `ATSUFontID` values, to font family numbers. Font family numbers were used by QuickDraw to represent fonts to the Font Manager. Some of these fonts, even if compatible with ATSUI, may not have font IDs.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUGetAllAttributes

Obtains an array of style attribute tags and value sizes for a style object.

```
OSStatus ATSUGetAllAttributes (
   ATSUStyle iStyle,
   ATSUAttributeInfo oAttributeInfoArray[],
   ItemCount iTagValuePairArraySize,
   ItemCount *oTagValuePairCount
);
```

Parameters

`iStyle`
: An `ATSUStyle` value specifying the style object to examine.

`oAttributeInfoArray`
: A pointer to memory you have allocated for an array of `ATSUAttributeInfo` values. On return, the array contains pairs of tags and value sizes for any of the object's style attributes that are not at default values. If you are uncertain of how much memory to allocate for this array, see the Discussion.

`iTagValuePairArraySize`
: An `ItemCount` value specifying the maximum number of tag and value size pairs to obtain for the style object. Typically, this is equivalent to the number of `ATSUAttributeInfo` structures for which you have allocated memory in the `oAttributeInfoArray` parameter. To determine this value, see the Discussion.

`oTagValuePairCount`
: A pointer to an `ItemCount` value. On return, the value specifies the actual number of `ATSUAttributeInfo` structures in the style object. This may be greater than the value you specified in the `iTagValuePairArraySize` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetAllAttributes` function obtains all nondefault style attribute tags and values sizes for a style object. You can pass a tag and value-size pair obtained from `ATSUGetAllAttributes` to the function `ATSUGetAttribute` (page 65) to determine the corresponding attribute value.

Typically you use the function `ATSUGetAllAttributes` by calling it twice, as follows:

1. Pass a reference to the style object to examine in the `iStyle` parameter, a valid pointer to an `ItemCount` value in the *`oTagValuePairCount`* parameter, `NULL` for the `oAttributeInfoArray` parameter, and 0 for the *`iTagValuePairArraySize`* parameter. `ATSUGetAllAttributes` returns the size of the tag and value-size arrays in the `oTagValuePairCount` parameter.

2. Allocate enough space for an array of the returned size, then call the `ATSUGetAllAttributes` function again, passing a valid pointer in the `oAttributeInfoArray` parameter. On return, the pointer refers to an array of the style attribute tag and value-size pairs contained in the style object.

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

Declared In
`ATSUnicodeObjects.h`

ATSUGetAllFontFeatures

Obtains the font features of a style object that are not at default settings.

```
OSStatus ATSUGetAllFontFeatures (
   ATSUStyle iStyle,
   ItemCount iMaximumFeatureCount,
   ATSUFontFeatureType oFeatureType[],
   ATSUFontFeatureSelector oFeatureSelector[],
   ItemCount *oActualFeatureCount
);
```

Parameters

`iStyle`
An `ATSUStyle` value specifying the style object to examine.

`iMaximumFeatureCount`
An `ItemCount` value specifying the maximum number of feature types and selectors to obtain for the style object. Typically, this is equivalent to the number of `ATSUFontFeatureType` and `ATSUFontFeatureSelector` values for which you have allocated memory in the `oFeatureType` and `oFeatureSelector` parameters, respectively. To determine this value, see the Discussion.

`oFeatureType`
A pointer to memory you have allocated for an array of `ATSUFontFeatureType` values. On return, the array contains constants identifying each type of font feature that is at a nondefault setting in the style object. If you are uncertain of how much memory to allocate for this array, see the Discussion.

`oFeatureSelector`
A pointer to memory you have allocated for an array of `ATSUFontFeatureSelector` values. On return, the array contains constants identifying the feature selectors that are at nondefault settings in the style object. Each selector determines the setting for a corresponding feature type produced in the *`oFeatureType`* parameter. If you are uncertain of how much memory to allocate for this array, see the Discussion.

`oActualFeatureCount`
A pointer to an `ItemCount` value. On return, the value specifies the actual number of font feature types and selectors in the style object. This may be greater than the value you specified in the `iMaximumFeatureCount` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetAllFontFeatures` function obtains all of a style object's font features that are not at default settings. Font features are grouped into categories called feature types, within which individual feature selectors define particular feature settings. The arrays produced by `ATSUGetAllFontFeatures` contain constants identifying the object's font types and their corresponding font selectors.

Typically you use the function `ATSUGetAllFontFeatures` by calling it twice, as follows:

1. Pass a reference to the style object to examine in the `iStyle` parameter, a valid pointer to an `ItemCount` value in the *`oActualFeatureCount`* parameter, `NULL` for the `oFeatureType` and `oFeatureSelector` parameters, and 0 for the *`iMaximumFeatureCount`* parameter. `ATSUGetAllFontFeatures` returns the size in the `oActualFeatureCount` parameter to use for the feature type and selector arrays.

2. Allocate enough space for arrays of the returned size, then call `ATSUGetAllFontFeatures` again, passing a pointer to the arrays in the `oFeatureType` and `oFeatureSelector` parameters. On return, the arrays contain the font feature types and selectors, respectively, for the style object.

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

Declared In
`ATSUnicodeFonts.h`

ATSUGetAllFontVariations

Obtains a style object's font variation values that are not at default settings.

```
OSStatus ATSUGetAllFontVariations (
   ATSUStyle iStyle,
   ItemCount iVariationCount,
   ATSUFontVariationAxis oVariationAxes[],
   ATSUFontVariationValue oFontVariationValues[],
   ItemCount *oActualVariationCount
);
```

Parameters

`iStyle`
An `ATSUStyle` value specifying the style object to examine.

`iVariationCount`
An `ItemCount` value specifying the maximum number of font variation values to obtain for the style object. Typically, this is equivalent to the number of `ATSUFontVariationAxis` and `ATSUFontVariationValue` values for which you have allocated memory in the `oVariationAxes` and `oFontVariationValues` parameters, respectively. To determine this value, see the Discussion.

`oVariationAxes`
A pointer to memory you have allocated for an array of `ATSUFontVariationAxis` values. On return, the array contains tags identifying those font variation axes in the style object that are not at default values. If you are uncertain of how much memory to allocate for this array, see the Discussion.

oFontVariationValues
: A pointer to memory you have allocated for an array of `ATSUFontVariationValue` values. On return, the array contains the current font variation values for the font variation axes produced in the `oVariationAxes` array. If you are uncertain of how much memory to allocate for this array, see the Discussion.

oActualVariationCount
: A pointer to an `ItemCount` value. On return, the value specifies the actual number of nondefault font variation values in the style object. This may be greater than the value you passed in the `iVariationCount` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetAllFontVariations` function obtains all of a style object's font variation axes that are not at default settings, as well as the current values for the axes.

Typically you use the function `ATSUGetAllFontVariations` by calling it twice, as follows:

1. Pass a reference to the style object to examine in the `iStyle` parameter, a pointer to an `ItemCount` value in the *`oActualVariationCount`* parameter, `NULL` for the `oVariationAxes` and `oFontVariationValues` parameters, and 0 for the *`iVariationCount`* parameter. `ATSUGetAllFontVariations` returns the size to use for the variation axes and value arrays in the `oActualVariationCount` parameter.

2. Allocate enough space for arrays of the returned size, then call `ATSUGetAllFontVariations` again, passing a pointer to the arrays in the `oVariationAxes` and `oFontVariationValues` parameters. On return, the arrays contain the font variation axes and their corresponding values, respectively, for the style object.

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

Declared In
`ATSUnicodeFonts.h`

ATSUGetAllLayoutControls

Obtains an array of layout control attribute tags and value sizes for a text layout object.

```
OSStatus ATSUGetAllLayoutControls (
   ATSUTextLayout iTextLayout,
   ATSUAttributeInfo oAttributeInfoArray[],
   ItemCount iTagValuePairArraySize,
   ItemCount *oTagValuePairCount
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object to examine.

`oAttributeInfoArray`

A pointer to memory you have allocated for an array of `ATSUAttributeInfo` values. On return, the array contains pairs of tags and value sizes for the object's layout control attributes that are not at default values. If you are uncertain of how much memory to allocate for this array, see the Discussion.

`iTagValuePairArraySize`

An `ItemCount` value specifying the maximum number of tag and value size pairs to obtain for the text layout object. Typically, this is equivalent to the number of `ATSUAttributeInfo` structures for which you have allocated memory in the `oAttributeInfoArray` parameter. To determine this value, see the Discussion.

`oTagValuePairCount`

A pointer to an `ItemCount` value. On return, the value specifies the actual number of `ATSUAttributeInfo` structures in the text layout object. This may be greater than the value you specified in the `iTagValuePairArraySize` parameter.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUGetAllLayoutControls` function obtains all nondefault layout control attribute tags and their values sizes for a text layout object. You can pass a tag and value size pair obtained from `ATSUGetAllLayoutControls` to the function `ATSUGetLayoutControl` (page 82) to determine the corresponding attribute value.

Typically you use the function `ATSUGetAllLayoutControls` by calling it twice, as follows:

1. Pass a reference to the text layout object to examine in the `iTextLayout` parameter, `NULL` for the `oAttributeInfoArray` parameter, a pointer to an `ItemCount` value in the *`oTagValuePairCount`* parameter, and `0` for the *`iTagValuePairArraySize`* parameter. `ATSUGetAllLayoutControls` returns the size of the tag and value size arrays in the `oTagValuePairCount` parameter.

2. Allocate enough space for an array of the returned size, then call the `ATSUGetAllLayoutControls` function again, passing a valid pointer in the `oAttributeInfoArray` parameter. On return, the pointer refers to an array of the layout control attribute tag and value size pairs contained in the text layout object.

To obtain the nondefault line control attribute tags and value sizes for a text layout object, call the function `ATSUGetAllLineControls` (page 63).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUGetAllLineControls

Obtains an array of line control attribute tags and value sizes for a line in a text layout object.

```
OSStatus ATSUGetAllLineControls (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart,
   ATSUAttributeInfo oAttributeInfoArray[],
   ItemCount iTagValuePairArraySize,
   ItemCount *oTagValuePairCount
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value specifying the text layout object to examine.

iLineStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the line for which to obtain line control attribute values.

oAttributeInfoArray

A pointer to memory you have allocated for an array of `ATSUAttributeInfo` values. On return, the array contains pairs of tags and value sizes for the object's line control attributes that are not at default values. If you are uncertain of how much memory to allocate for this array, see the Discussion.

iTagValuePairArraySize

An `ItemCount` value specifying the maximum number of tag and value size pairs to obtain for the line. Typically, this is equivalent to the number of `ATSUAttributeInfo` structures for which you have allocated memory in the `oAttributeInfoArray` parameter. To determine this value, see the Discussion.

oTagValuePairCount

A pointer to an `ItemCount` value. On return, the value specifies the actual number of `ATSUAttributeInfo` structures in the line. This may be greater than the value you specified in the `iTagValuePairArraySize` parameter.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUGetAllLineControls` function obtains all nondefault line control attribute tags and their values sizes for a line in a text layout object. You can pass a tag and value size pair obtained from `ATSUGetAllLineControls` to the function `ATSUGetLineControl` (page 83) to determine the corresponding attribute value.

Typically you use the function `ATSUGetAllLineControls` by calling it twice, as follows:

1. Pass a reference to the text layout object to examine in the `iTextLayout` parameter, the appropriate `UniCharArrayOffset` value in the *`iLineStart`* parameter, `NULL` for the `oAttributeInfoArray` parameter, a pointer to an `ItemCount` value in the *`oTagValuePairCount`* parameter, and 0 for the *`iTagValuePairArraySize`* parameter. `ATSUGetAllLineControls` returns the size of the tag and value size arrays in the `oTagValuePairCount` parameter.

2. Allocate enough space for an array of the returned size, then call the `ATSUGetAllLineControls` function again, passing a valid pointer in the `oAttributeInfoArray` parameter. On return, the pointer refers to an array of the line control attribute tag and value size pairs contained in the specified line.

To obtain the nondefault layout control attribute tags and value sizes for a text layout object, call the function `ATSUGetAllLayoutControls` (page 62).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetAttribute

Obtains a style attribute value for a style object.

```
OSStatus ATSUGetAttribute (
   ATSUStyle iStyle,
   ATSUAttributeTag iTag,
   ByteCount iExpectedValueSize,
   ATSUAttributeValuePtr oValue,
   ByteCount *oActualValueSize
);
```

Parameters

`iStyle`
An `ATSUStyle` value specifying the style object for which to obtain an attribute value.

`iTag`
An `ATSUAttributeTag` constant identifying the attribute value to obtain. See "Attribute Tags" (page 196) for a description of the Apple-defined style attribute tag constants.

`iExpectedValueSize`
The expected size (in bytes) of the value to obtain. To determine the size of an application-defined style attribute value, see the Discussion.

`oValue`
An `ATSUAttributeValuePtr` value, identifying the memory you have allocated for the attribute value. If you are uncertain of how much memory to allocate, see the Discussion. On return, *`oValue`* contains a valid pointer to the actual attribute value.

`oActualValueSize`
A pointer to a `ByteCount` value. On return, the value contains the actual size (in bytes) of the attribute value. You should examine this parameter if you are unsure of the size of the attribute value being obtained, as in the case of custom style run attributes.

Return Value
A result code. See "ATSUI Result Codes" (page 234). Note that if the attribute value you want to obtain is not set, `ATSUGetAttribute` produces the default value in the `oValue` parameter and returns the result code `kATSUNotSetErr`.

Discussion
The `ATSUGetAttribute` function obtains the value of a specified style attribute for a given style object.

Before calling `ATSUGetAttribute`, you should call the function `ATSUGetAllAttributes` (page 59) to obtain an array of nondefault style attribute tags and value sizes for the style object. You can then pass `ATSUGetAttribute` the tag and value size for the attribute value to obtain.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetContinuousAttributes

Obtains the style attribute values that are continuous over a given text range.

```
OSStatus ATSUGetContinuousAttributes (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOffset,
   UniCharCount iLength,
   ATSUStyle oStyle
);
```

Parameters

`iTextLayout`
: An `ATSUTextLayout` value specifying the text layout object to examine.

`iOffset`
: A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the text range to examine. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iLength` parameter.

`iLength`
: A `UniCharCount` value specifying the length of the text range to examine. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

`oStyle`
: An `ATSUStyle` value. On return, the style object contains those attributes that are the same for the entire text range specified by the `iOffset` and `iLength` parameters.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetContinuousAttributes` function examines the specified text range to obtain the style attribute values (including those at default values) that remain consistent for the entire text range. You should call `ATSUGetContinuousAttributes` to determine the style information that remains constant over text that has been selected by the user.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetFontFeature

Obtains the font feature corresponding to an index into an array of font features for a style object.

```
OSStatus ATSUGetFontFeature (
   ATSUStyle iStyle,
   ItemCount iFeatureIndex,
   ATSUFontFeatureType *oFeatureType,
   ATSUFontFeatureSelector *oFeatureSelector
);
```

Parameters

iStyle

An `ATSUStyle` value specifying the style object to examine.

iFeatureIndex

An `ItemCount` value specifying an index into the array of font features for the style object. This index identifies the font feature to examine. Because this index is zero-based, you must pass a value between 0 and one less than the value produced in the `oActualFeatureCount` parameter of the function `ATSUGetAllFontFeatures` (page 60).

oFeatureType

A pointer to memory you have allocated for an `ATSUFontFeatureType` value. On return, the value identifies the font feature type corresponding to the index passed in the `iFeatureIndex` parameter.

oFeatureSelector

A pointer to memory you have allocated for an `ATSUFontFeatureSelector` value. On return, the value identifies the font feature selector that corresponds to the feature type produced in the *oFeatureType* parameter.

Return Value

A result code. Note that if the index specifies a font feature that is not set, `ATSUGetFontFeature` produces the font-specified default value for the feature and returns the result code `kATSUNotSetErr`. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUGetFontFeature` function obtains the setting for a specified font feature in a style object. You might typically call `ATSUGetFontFeature` if you need to obtain one previously set feature after another within your program's processing loop. To obtain all previously set font features for a given style object, you can call the function `ATSUGetAllFontFeatures` (page 60).

Before calling `ATSUGetFontFeature`, you should call the function `ATSUGetAllFontFeatures` (page 60) to obtain a count of the font features that are set in the style object. You can then pass the index for the feature whose setting you want to obtain in the `iTag` and `iMaximumValueSize` parameters of `ATSUGetFontFeature`.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFonts.h`

ATSUGetFontFeatureNameCode

Obtains the name code for a font's feature type or selector that matches an ASTUI font ID, feature type, and feature selector.

```
OSStatus ATSUGetFontFeatureNameCode (
   ATSUFontID iFontID,
   ATSUFontFeatureType iType,
   ATSUFontFeatureSelector iSelector,
   FontNameCode *oNameCode
);
```

Parameters

iFont

The ATSUFontID value of the font for which to obtain the name code for a feature type or selector. Note that because Apple Type Services assigns ATSUFontID values systemwide at runtime, font IDs can change across system restarts.

iType

An ATSUFontFeatureType constant identifying a valid feature type. To obtain the valid feature types for a font, call the function ATSUGetFontFeatureTypes (page 70).

iSelector

An ATSUFontFeatureSelector constant identifying a valid feature selector that corresponds to the feature type passed in the iType parameter. If you pass the constant kATSUNoSelector, the name code produced by ATSUGetFontFeatureNameCode is that of the feature type, not the feature selector. To obtain the valid feature selectors for a font, call the functionATSUGetFontFeatureSelectors (page 68).

oNameCode

A pointer to a FontNameCode value. On return, the value contains the name code for the font feature selector or type. The FontNameCode is a UInt32 data type, and it is defined in the SFNTTypes.h header file.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The ATSUGetFontFeatureNameCode function obtains the name code for a font's feature type or selector that matches an ASTUI font ID, feature type and feature selector values. By default, ATSUGetFontFeatureNameCode function obtains the name code of a feature selector. To determine the name code of a feature type, pass the constant kATSUNoSelector in the iSelector parameter.

You can use the function ATSUFindFontName (page 53) to obtain the localized name string for the name code produced by ATSUGetFontFeatureNameCode.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

ATSUnicodeFonts.h

ATSUGetFontFeatureSelectors

Obtains the available feature selectors for a given feature type in a font.

```
OSStatus ATSUGetFontFeatureSelectors (
   ATSUFontID iFontID,
   ATSUFontFeatureType iType,
   ItemCount iMaximumSelectors,
   ATSUFontFeatureSelector oSelectors[],
   Boolean oSelectorIsOnByDefault[],
   ItemCount *oActualSelectorCount,
   Boolean *oIsMutuallyExclusive
);
```

Parameters

iFont

An `ATSUFontID` value identifying the font to examine.

iType

An `ATSUFontFeatureType` value specifying one of the font's supported feature types. To obtain the available feature types for a font, call the function `ATSUGetFontFeatureTypes` (page 70).

iMaximumSelectors

An `ItemCount` value specifying the maximum number of feature selectors to obtain for the font's specified feature type. Typically, this is equivalent to the number of elements in the `oSelectors` array.

oSelectors

A pointer to memory you have allocated for an array of `ATSUFontFeatureSelector` values. You can call the function `ATSUCountFontFeatureSelectors` (page 34) to obtain the number of available feature selectors for a given font feature type and thus determine the amount of memory to allocate. On return, the array contains constants identifying each available feature selector for the given feature type. The constants that represent font feature selectors are defined in the header file `SFNTLayoutTypes.h` and are described in *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

oSelectorIsOnByDefault

A pointer to memory you have allocated for an array of `Boolean` values. The number of elements in this array should correspond to the number of elements in the *oSelectors* array. On return, the array contains `Boolean` values indicating whether the corresponding feature selector in the *oSelectors* array is on or off. If `true`, the feature selector is on by default; if false, off.

oActualSelectorCount

A pointer to an `ItemCount` value. On return, the value specifies the actual number of feature selectors defined for the given feature type. This value may be greater than the value you specify in the `iMaximumSelectors` parameter.

oIsMutuallyExclusive

A pointer to a `Boolean` value. On return, the value indicates whether the feature selectors for the given feature type are exclusive or nonexclusive. If a feature type is exclusive you can choose only one of its available feature selectors at a time, such as whether to display numbers as proportional or fixed-width. If a feature type is nonexclusive, you can enable any number of feature selectors at once. If `true`, the feature type is exclusive and only one selector can be used at a time.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

A given font may not support all possible feature types and selectors. If you select features that are not available in a font, you won't see a change in the glyph's appearance. To determine the available features of a font, you can call the functions `ATSUGetFontFeatureTypes` (page 70) and `ATSUGetFontFeatureSelectors`.

The `ATSUGetFontFeatureSelectors` function reads the font data table for the specified font and obtains its supported feature selectors for the given feature types. You can then use this information both to present the user a list of font features from which to select and to call such functions as `ATSUSetFontFeatures` (page 120) with more accuracy.

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

Declared In
`ATSUnicodeFonts.h`

ATSUGetFontFeatureTypes

Obtains the available feature types of a font.

```
OSStatus ATSUGetFontFeatureTypes (
   ATSUFontID iFontID,
   ItemCount iMaximumTypes,
   ATSUFontFeatureType oTypes[],
   ItemCount *oActualTypeCount
);
```

Parameters

`iFont`
An `ATSUFontID` value identifying the font to examine.

`iMaximumTypes`
An `ItemCount` value specifying the maximum number of feature types to obtain for the font. Typically, this is equivalent to the number of elements in the `oTypes` array.

`oTypes`
A pointer to memory you have allocated for an array of `ATSUFontFeatureType` values. You can call the function `ATSUCountFontFeatureTypes` (page 35) to obtain the number of available feature types for a given font and thus determine the amount of memory to allocate. On return, the array contains constants identifying each type of feature that is defined for the font. The constants that represent font feature types are defined in the header file `SFNTLayoutTypes.h` and are described in *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

`oActualTypeCount`
A pointer to an `ItemCount` value. On return, the value specifies the actual number of feature types defined in the font. This may be greater than the value you specify in the `iMaximumTypes` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
A given font may not support all possible feature types and selectors. If you select features that are not available in a font, you won't see a change in the glyph's appearance. To determine the available features of a font, you can call the functions `ATSUGetFontFeatureTypes` and `ATSUGetFontFeatureSelectors` (page 68).

The `ATSUGetFontFeatureTypes` function reads the font data table for the specified font and obtains its supported feature types. You can then use this information both to present the user a list of font features from which to select and to call such functions as `ATSUSetFontFeatures` (page 120) with more accuracy.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUGetFontIDs

Obtains a list of all the ATSUI-compatible fonts installed on the user's system.

```
OSStatus ATSUGetFontIDs (
   ATSUFontID oFontIDs[],
   ItemCount iArraySize,
   ItemCount *oFontCount
);
```

Parameters

`oFontIDs`

A pointer to memory you have allocated for an array of `ATSUFontID` values. On return, the array contains unique identifiers for each of the ATSUI-compatible fonts installed on the user's system. You should allocate enough memory to contain an array the size of the count produced by the function `ATSUFontCount` (page 57).

`iArraySize`

An `ItemCount` value specifying the maximum number of fonts to obtain. Typically, this is equivalent to the number of `ATSUFontID` values for which you have allocated memory in the `oFontIDs` parameter.

`oFontCount`

A pointer to an `ItemCount` value. On return, the value specifies the actual number of ATSUI-compatible fonts installed on the user's system. This may be greater than the value you specified in the `iArraySize` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetFontIDs` function obtains the IDs of all the fonts on the user's system except for the last-resort font. It is important to note that the set of installed ATSUI-compatible fonts may change while your application is running. In Mac OS X, the set of installed fonts may change at any time. Although in Mac OS 9, fonts cannot be removed from the Fonts folder while an application other than the Finder is running, they can be removed from other locations, and it is possible for fonts to be added.

To ensure an accurate representation of the set of installed ATSUI-compatible fonts, you should call `ATSUGetFontIDs` to rebuild your font menu each time your application is brought to the foreground.

Finally, note that Apple Type Services assigns `ATSUFontID` values systemwide at runtime. As a result, these font IDs can change across system restarts.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUGetFontInstance

Obtains the font variation axis values for a font instance.

```
OSStatus ATSUGetFontInstance (
   ATSUFontID iFontID,
   ItemCount iFontInstanceIndex,
   ItemCount iMaximumVariations,
   ATSUFontVariationAxis oAxes[],
   ATSUFontVariationValue oValues[],
   ItemCount *oActualVariationCount
);
```

Parameters

`iFont`
An `ATSUFontID` value identifying the font to examine.

`iFontInstanceIndex`
An `ItemCount` value specifying an index into an array of instances for the font. This index identifies the font instance to examine. Because this index is zero-based, you must pass a value between 0 and one less than the value produced in the `oInstances` parameter of the function `ATSUCountFontInstances` (page 35).

`iMaximumVariations`
An `ItemCount` value specifying the maximum number of font variation axes to obtain for the font instance. Typically, this is equivalent to the number of `ATSUFontVariationAxis` and `ATSUFontVariationValue` values for which you have allocated memory in the `oAxes` and `oValues` parameters, respectively. To determine this value, see the Discussion.

`oAxes`
A pointer to memory you have allocated for an array of `ATSUFontVariationAxis` values. On return, the array contains tags identifying the font variation axes that constitute the font instance. If you are uncertain of how much memory to allocate for this array, see the Discussion.

`oValues`
A pointer to memory you have allocated for an array of `ATSUFontVariationValue` values. On return, the array contains the defined values for the font variation axes produced in the `oAxes` array. If you are uncertain of how much memory to allocate for this array, see the Discussion.

`oActualVariationCount`
A pointer to an `ItemCount` value. On return, the value specifies the actual number of font variation axes that constitute the font instance. This may be greater than the value you passed in the `iMaximumVariations` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
A font instance consists of a named set of values for each variation axis in a font. For example, suppose a font has the variation axis `'wght'` with a minimum value of 0.0, a default of 0.5, and a maximum of 1.0. Additionally, the variation axis `'wdth'` is also defined for the font, with a similar value range. The type designer can then choose to declare a font instance for a set of specific values within these axes, such as

"Demibold" for a value of 0.8 for the `'wght'` axis and 0.5 for the `'wdth'` axis. By calling the function `ATSUGetFontInstance`, you can obtain the variation axis values for a given index into an array of font instances.

Typically you use the function `ATSUGetFontInstance` by calling it twice, as follows:

1. Pass the ID of the font to examine in the `iFont` parameter, a valid pointer to an `ItemCount` value in the *`oActualVariationCount`* parameter, `NULL` for the `oAxes` and `oValues` parameters, and 0 for the other parameters. `ATSUGetFontInstance` returns the size to use for the `oAxes` and `oValues` arrays in the `oActualVariationCount` parameter.

2. Allocate enough space for arrays of the returned size, then call the `ATSUGetFontInstance` again, passing pointers to the arrays in the `oAxes` and `oValues` parameters. On return, the arrays contain the font variation axes and their corresponding values, respectively, for the font instance.

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

Declared In
`ATSUnicodeFonts.h`

ATSUGetFontInstanceNameCode

Obtains the name code for the font instance that matches an ASTUI font ID and font instance index value.

```
OSStatus ATSUGetFontInstanceNameCode (
   ATSUFontID iFontID,
   ItemCount iInstanceIndex,
   FontNameCode *oNameCode
);
```

Parameters

`iFont`
The `ATSUFontID` value of the font for which to obtain a font instance name code. Note that because Apple Type Services assigns `ATSUFontID` values systemwide at runtime, font IDs can change across system restarts.

`iInstanceIndex`
An `ItemCount` value providing an index to the font instance for which to obtain a name code. Because this index must be 0-based, you should pass a value between 0 and one less than the count produced by the function `ATSUCountFontInstances` (page 35).

`oNameCode`
A pointer to a `FontNameCode` value. On return, the value contains the name code for the font instance. The `FontNameCode` is a `UInt32` data type, and it is defined in the `SFNTTypes.h` header file.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
A font instance consists of a named set of values for each variation axis in a font. The `ATSUGetFontInstanceNameCode` function obtains the name code for the font instance that matches an ASTUI font ID and font instance index value.

You can use the function ATSUFindFontName (page 53) to obtain the localized name string for the name code produced by ATSUGetFontInstanceNameCode. You can obtain the font variation axis values for a font instance by calling the functionATSUGetFontInstance (page 72).

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

Declared In
ATSUnicodeFonts.h

ATSUGetFontVariationNameCode

Obtains the name code for the font variation that matches an ASTUI font ID and font variation axis.

```
OSStatus ATSUGetFontVariationNameCode (
   ATSUFontID iFontID,
   ATSUFontVariationAxis iAxis,
   FontNameCode *oNameCode
);
```

Parameters

iFont
: The ATSUFontID value of the font for which to obtain a font variation name code. Note that because Apple Type Services assigns ATSUFontID values systemwide at runtime, font IDs can change across system restarts.

iAxis
: An ATSUFontVariationAxis value representing a valid variation axis tag. To obtain a valid variation axis tag for a font, you can call the functionsATSUGetIndFontVariation (page 81) or ATSUGetFontInstance (page 72).

oNameCode
: A pointer to a FontNameCode value. On return, the value contains the name code for the font variation. The FontNameCode is a UInt32 data type, and it is defined in the SFNTTypes.h header file.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The ATSUGetFontVariationNameCode function obtains the name code for the font variation that matches an ASTUI font ID and font variation axis tag. You can use the function ATSUFindFontName (page 53) to obtain the localized name string for the name code produced by ATSUGetFontVariationNameCode.

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

Declared In
ATSUnicodeFonts.h

ATSUGetFontVariationValue

Obtains the current value for a font variation axis in a style object.

```
OSStatus ATSUGetFontVariationValue (
   ATSUStyle iStyle,
   ATSUFontVariationAxis iFontVariationAxis,
   ATSUFontVariationValue *oFontVariationValue
);
```

Parameters

iStyle

An `ATSUStyle` value specifying the style object to examine.

iFontVariationAxis

An `ATSUFontVariationAxis` tag specifying the style object's variation axis to examine. You can obtain variation axis tags for a style object from the function `ATSUGetAllFontVariations` (page 61).

oFontVariationValue

A pointer to memory you have allocated for an `ATSUFontVariationValue` value. On return, `ATSUGetFontVariationValue` produces the currently set value for the style object's specified variation axis. If this value has not been set, `ATSUGetFontVariationValue` produces the font-defined default value.

Return Value

A result code. Note that if no value has been set for the specified variation axis, `ATSUGetFontVariationValue` produces the font-defined default value and returns the result code `kATSUNotSetErr`. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUGetFontVariationValue` function obtains the setting for a specified font variation axis in a style object. You might typically call `ATSUGetFontVariationValue` if you need to obtain one previously set variation axis value after another within your program's processing loop. To obtain all nondefault font variation axis values for a given style object, you can call the function `ATSUGetAllFontVariations` (page 61).

Before calling `ATSUGetFontVariationValue`, call the function ATSUGetAllFontVariations (page 61) to obtain the font variation axes that are set for the style object.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFonts.h`

ATSUGetGlyphBounds

Obtains the typographic bounds of a line of glyphs after final layout.

```
OSStatus ATSUGetGlyphBounds (
   ATSUTextLayout iTextLayout,
   ATSUTextMeasurement iTextBasePointX,
   ATSUTextMeasurement iTextBasePointY,
   UniCharArrayOffset iBoundsCharStart,
   UniCharCount iBoundsCharLength,
   UInt16 iTypeOfBounds,
   ItemCount iMaxNumberOfBounds,
   ATSTrapezoid oGlyphBounds[],
   ItemCount *oActualNumberOfBounds
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value specifying the text layout object to examine.

iTextBasePointX

An `ATSUTextMeasurement` value specifying the x-coordinate of the origin of the line containing the glyphs in the current graphics port or Quartz graphics context. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to obtain the glyph bounds relative to the current pen location in the current graphics port or graphics context. You can pass 0 to obtain only the dimensions of the bounds relative to one another, not their actual onscreen position.

iTextBasePointY

An `ATSUTextMeasurement` value specifying the y-coordinate of the origin of the line containing the glyphs in the current graphics port or Quartz graphics context. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to obtain the glyph bounds relative to the current pen location in the current graphics port or graphics context. You can pass `0` to obtain only the dimensions of the bounds relative to one another, not their actual onscreen position.

iBoundsCharStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the character corresponding to the first glyph to measure. To indicate that the text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`.

iBoundsCharLength

A `UniCharCount` value specifying the length of the text range to measure. If you want the range to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

iTypeOfBounds

A glyph bounds constant indicating whether the width of the resulting typographic glyph bounds is determined using the caret origin (midway between two characters), the glyph origin in device space, or the glyph origin in fractional absolute positions (uncorrected for device display). See "Glyph Origin Selectors" (page 215) for a description of possible values.

iMaxNumberOfBounds

An `ItemCount` value specifying the maximum number of bounding trapezoids to obtain. Typically, this is equivalent to the number of bounds in the `oGlyphBounds` array. To determine this value, see the Discussion.

oGlyphBounds

A pointer to memory you have allocated for an array of `ATSTrapezoid` values. On return, the array contains a trapezoid representing the typographic bounds for glyphs in the text range. If the specified range of text encloses nested bidirectional text, `ATSUGetGlyphBounds` produces multiple trapezoids defining these regions.In ATSUI 1.1, the maximum number of enclosing trapezoids that can be returned is 31; in ATSUI 1.2, the maximum number is 127. If you pass a range that covers an entire line, `ATSUGetGlyphBounds` returns 1 trapezoid. If you are uncertain of how much memory to allocate for this array, see the Discussion.

oActualNumberOfBounds

A pointer to an `ItemCount` value. On return, the value specifies the actual number of enclosing trapezoids bounding the specified characters. This may be greater than the value you provide in the `iMaxNumberOfBounds` parameter.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

There are two kinds of bounds that your application may typically want to obtain for a block of text: typographic bounds and image bounds. The image bounds define the smallest rectangle that completely encloses the filled or framed parts of a block of text—that is, the text's "inked" glyphs. Because of the potential differences in glyph height in a text block, your application may instead need to determine the typographic bounds. The typographic bounding rectangle contains the extra space above and below the image bounding rectangle where characters with ascenders or descenders would be drawn (even if none currently are).

The `ATSUGetGlyphBounds` function produces the enclosing trapezoid(s) that represent the typographic bounds for glyphs in a final, laid-out range of text. You typically call this function when you need to obtain an enclosing trapezoid for a line, taking rotation and all other layout attributes into account.

ATSUI determines the height of each trapezoid by examining any line ascent and descent attribute values you may have set for the line. If you have not set these attributes for the line, the `ATSUGetGlyphBounds` function uses any line ascent and descent values you may have set for the text layout object containing the line. If these are not set, `ATSUGetGlyphBounds` uses the font's natural line ascent and descent values for the line. If these are previously set, `ATSUGetGlyphBounds` uses the `ATSUStyle` ascent and or descent/leading values.

Depending on the value you pass in the `iTypeOfBounds` parameter, the width of the resulting trapezoid(s) is determined using one of the following values:

- the caret origin, located halfway between two characters, which should be used when performing your own highlighting
- the glyph origin in device space, which is useful for obtaining bounds adjusted for specific rendering and device constraints
- the glyph origin in fractional (or "ideal") absolute positions, uncorrected for device display

Note that the coordinates produced for the trapezoid(s) are offset by the amount specified in the `iTextBasePointX` and `iTextBasePointY` parameters. If your goal in calling the `ATSUGetGlyphBounds` function is to obtain metrics for drawing the typographic bounds on the screen, pass the position of the origin of the line in the current graphics port or graphics context in these parameters. This enables `ATSUGetGlyphBounds` to match the trapezoids to their onscreen image.

Before calculating the typographic glyph bounds for the given text range, the `ATSUGetGlyphBounds` function examines the text layout object to make sure that the style runs cover the entire range of text. If there are gaps between style runs, `ATSUGetGlyphBounds` assigns the characters in the gap to the style run following the gap. If there is no style run at the beginning of the range of text, `ATSUGetGlyphBounds` assigns these characters to the first style run it can find. If there is no style run at the end of the range of text, `ATSUGetGlyphBounds` assigns the remaining characters to the last style run it can find.

Typically you use the `ATSUGetGlyphBounds` function by calling it twice, as follows:

1. Pass `NULL` for the `oGlyphBounds` parameter, `0` for the *`iMaxNumberOfBounds`* parameter, and valid values for the other parameters. The `ATSUGetGlyphBounds` function returns the actual number of trapezoids needed to enclose the glyphs in the `oActualNumberOfBounds` parameter.

2. Allocate enough space for a buffer of the returned size, then call the function again, passing a valid pointer to the buffer in the `oGlyphBounds` parameter. On return, the buffer contains the trapezoids for the glyphs' typographic bounds.

To obtain the typographic bounds of a line of text prior to line layout, call the function `ATSUGetUnjustifiedBounds` (page 93). To calculate the image bounding rectangle for a final laid-out line, call the function `ATSUMeasureTextImage` (page 107).

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

Declared In
`ATSUnicodeDrawing.h`

ATSUGetIndFontName

Obtains a name string, name code, platform, script, and language for the font that matches an ATSUI font ID and name table index value.

```
OSStatus ATSUGetIndFontName (
   ATSUFontID iFontID,
   ItemCount iFontNameIndex,
   ByteCount iMaximumNameLength,
   Ptr oName,
   ByteCount *oActualNameLength,
   FontNameCode *oFontNameCode,
   FontPlatformCode *oFontNamePlatform,
   FontScriptCode *oFontNameScript,
   FontLanguageCode *oFontNameLanguage
);
```

Parameters

`iFontID`
The `ATSUFontID` value of the font for which to obtain information. Note that because Apple Type Services assigns `ATSUFontID` values systemwide at runtime, font IDs can change across system restarts.

`iFontNameIndex`
An `ItemCount` value providing an index to the font for which to obtain information. Because this index must be 0-based, you should pass a value between 0 and one less than the count produced by the function `ATSUCountFontNames` (page 36).

iMaximumNameLength
: A `ByteCount` value specifying the maximum length of the font name string to obtain. Typically, this is equivalent to the size of the buffer that you have allocated in the `oName` parameter. To determine this length, see the Discussion.

oName
: A pointer to a buffer. On return, the buffer contains the name string of the font matching the ATSUI font ID and name table index value being passed. If the buffer you allocate is not large enough to contain the name string, `ATSUGetIndFontName` produces a partial string.

oActualNameLength
: A pointer to a `ByteCount` value. On return, the value specifies the actual length of the complete name string. This may be greater than the value passed in the `iMaximumNameLength` parameter. You should check this value to ensure that you have allocated sufficient memory and therefore obtained the complete name string for the font.

oFontNameCode
: A pointer to a `FontNameCode` value. On return, the value contains the name code for the font. The `FontNameCode` is a `UInt32` data type, and it is defined in the `SFNTTypes.h` header file. ATSUI can return any of the following constants, `kFontCopyrightName`, `kFontFamilyName`, `kFontStyleName`, `kFontUniqueName`, `kFontFullName`, `kFontVersionName`, `kFontPostscriptName`, `kFontTrademarkName`, `kFontManufacturerName`, `kFontDesignerName`, `kFontDescriptionName`, `kFontVendorURLName`, `kFontDesignerURLName`, `kFontLicenseDescriptionName`, or `kFontLicenseInfoURLName`.

oFontNamePlatform
: A pointer to a `FontPlatformCode` value. On return, this value specifies the encoding of the font, for example, `kFontUnicodePlatform, kFontMacintoshPlatform, kFontReservedPlatform, kFontMicrosoftPlatform,` or `kFontCustomPlatform`. See the `SFNTTypes.h` header file for a definition of the `FontPlatformCode` type and a list of possible values.

oFontNameScript
: A pointer to a `FontScriptCode` value. On return, this value specifies the script code of the font, for example, `kFontRomanScript`. See the `SFNTTypes.h` header file for a definition of the `FontScriptCode` type and a list of possible values.

oFontNameLanguage
: A pointer to a `FontLanguageCode` value. On return, this value specifies the language of the font, for example, `kFontNorwegianLanguage`. See the `SFNTTypes.h` header file for a definition of the `FontLanguageCode` type and a list of possible values.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetIndFontName` function obtains a name string, name code, language code, script code, and platform code for the font that matches the specified ATSUI font ID and name table index value.

Typically you use the `ATSUGetIndFontName` function by calling it twice, as follows:

1. Pass valid values for the *iFontID*, *iFontNameIndex*, and *oActualNameLength* parameters, 0 for the *iMaximumNameLength* parameter, and `NULL` for the other parameters. `ATSUGetIndFontName` returns the length of the font name string in the `oActualNameLength` parameter.

2. Allocate enough space for a buffer of the returned size, then call the function again, passing a valid pointer to the buffer in the `oName` parameter. On return, the buffer contains the font name string.

To find a name string and index value for the first font in a name table that matches an ATSUI font ID and the specified font parameters, call the function `ATSUFindFontName` (page 53). To obtain an ATSUI font ID for the first font in a name table that matches the specified name string, name code, platform, script, and/or language, call the function `ATSUFindFontFromName` (page 52).

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

Declared In
`ATSUnicodeFonts.h`

ATSUGetIndFontTracking

Obtains the name code and tracking value for the font tracking that matches an ASTUI font ID, glyph orientation, and tracking table index.

```
OSStatus ATSUGetIndFontTracking (
   ATSUFontID iFontID,
   ATSUVerticalCharacterType iCharacterOrientation,
   ItemCount iTrackIndex,
   Fixed *oFontTrackingValue,
   FontNameCode *oNameCode
);
```

Parameters

iFont
: The `ATSUFontID` value of the font tracking for which to obtain a name code and tracking value. Note that because Apple Type Services assigns `ATSUFontID` values systemwide at runtime, font IDs can change across system restarts.

iCharacterOrientation
: An `ATSUVerticalCharacterType` constant identifying the glyph orientation of the font tracking value to obtain, for example `kATSUStronglyHorizontal` or `kATSUStronglyVertical`. See "Vertical Character Types" (page 234) for a description of possible values.

iTrackIndex
: An `ItemCount` value providing an index to the font tracking for which to obtain information. Because this index must be 0-based, you should pass a value between 0 and one less than the count produced by the function `ATSUCountFontTracking` (page 37).

oFontTrackingValue
: A pointer to a `Fixed` value. On return, the value contains the font tracking value.

oNameCode
: A pointer to a `FontNameCode` value. On return, the value contains the name code for the font tracking. The `FontNameCode` is a `UInt32` data type, and it is defined in the `SFNTTypes.h` header file.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You can call the `ATSUGetIndFontTracking` function to obtain the name code and tracking value that matches the specified ATSUI font ID, glyph orientation, and tracking table index value.

You can use the function `ATSUFindFontName` (page 53) to obtain the localized name string for the name code produced by `ATSUGetIndFontTracking`.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeFonts.h`

ATSUGetIndFontVariation

Obtains a variation axis and its value range for a font.

```
OSStatus ATSUGetIndFontVariation (
   ATSUFontID iFontID,
   ItemCount iVariationIndex,
   ATSUFontVariationAxis *oATSUFontVariationAxis,
   ATSUFontVariationValue *oMinimumValue,
   ATSUFontVariationValue *oMaximumValue,
   ATSUFontVariationValue *oDefaultValue
);
```

Parameters

iFont
: An `ATSUFontID` value identifying the font to examine.

iVariationIndex
: An `ItemCount` value specifying an index into the array of variation axes for the font. This index identifies the font variation axis to examine. Because this index is zero-based, you must pass a value between 0 and one less than the value produced in the `oVariationCount` parameter of the function `ATSUCountFontVariations` (page 37).

oATSUFontVariationAxis
: A pointer to an `ATSUFontVariationAxis` value. On return, the value provides a four-character code identifying the font variation axis corresponding to the specified index.

oMinimumValue
: A pointer to an `ATSUFontVariationValue` value. On return, the value identifies the variation axis minimum.

oMaximumValue
: A pointer to an `ATSUFontVariationValue` value. On return, the value identifies the variation axis maximum.

oDefaultValue
: A pointer to an `ATSUFontVariationValue` value. On return, the value identifies the variation axis default.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
By calling the function `ATSUGetIndFontVariation`, you can obtain a variation axis and its maximum, minimum, and default values for a font.

If you supply font variation axes and values to the function `ATSUSetVariations` (page 133), you can change the appearance of a style object's font accordingly.

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

Declared In
`ATSUnicodeFonts.h`

ATSUGetLayoutControl

Obtains a layout control attribute value for a text layout object.

```
OSStatus ATSUGetLayoutControl (
   ATSUTextLayout iTextLayout,
   ATSUAttributeTag iTag,
   ByteCount iExpectedValueSize,
   ATSUAttributeValuePtr oValue,
   ByteCount *oActualValueSize
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object for which to obtain a layout control attribute value.

`iTag`
An `ATSUAttributeTag` constant identifying the attribute value to obtain. See "Attribute Tags" (page 196) for a description of the Apple-defined attribute tag constants.

`iExpectedValueSize`
The expected size (in bytes) of the value to obtain. To determine the size of an application-defined style attribute value, see the Discussion.

`oValue`
An `ATSUAttributeValuePtr` pointer, identifying the memory you have allocated for the attribute value. If you are uncertain of how much memory to allocate, see the Discussion. On return, *`oValue`* contains a valid pointer to the actual attribute value. If the value is unset, `ATSUGetLayoutControl` produces the default value in this parameter.

`oActualValueSize`
A pointer to a `ByteCount` value. On return, the value contains the actual size (in bytes) of the attribute value. You should examine this parameter if you are unsure of the size of the attribute value being obtained.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetLayoutControl` function obtains the value of a specified layout control attribute for a given text layout object.

Before calling `ATSUGetLayoutControl`, you should call the function `ATSUGetAllLayoutControls` (page 62) to obtain an array of nondefault layout control attribute tags and value sizes for the text layout object. You can then pass the tag and value size for the attribute value to obtain to `ATSUGetLayoutControl`.

Typically you use the function `ATSUGetLayoutControl` by calling it twice, as follows:

1. Pass a reference to the text layout object to examine in the `iTextLayout` parameter, `NULL` for the `oValue` parameter, `0` for the `iExpectedValueSize` parameter. `ATSUGetLayoutControl` returns the actual size of the attribute value in the `oActualValueSize` parameter.

2. Allocate enough space for an array of the returned size, then call the `ATSUGetLayoutControl` function again, passing a valid pointer in the `oValue` parameter. On return, the pointer refers to the actual attribute value contained in the text layout object.

To obtain the value of a line control attribute value for a text layout object, call the function `ATSUGetLineControl` (page 83).

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

Declared In
`ATSUnicodeObjects.h`

ATSUGetLineControl

Obtains a line control attribute value for a line in a text layout object.

```
OSStatus ATSUGetLineControl (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart,
   ATSUAttributeTag iTag,
   ByteCount iExpectedValueSize,
   ATSUAttributeValuePtr oValue,
   ByteCount *oActualValueSize
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object for which to obtain a line control attribute value.

`iLineStart`
A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the line for which to obtain a line control attribute value.

`iTag`
An `ATSUAttributeTag` constant identifying the attribute value to obtain. See "Attribute Tags" (page 196) for a description of the Apple-defined attribute tag constants.

`iExpectedValueSize`
The expected size (in bytes) of the value to obtain.

`oValue`
An `ATSUAttributeValuePtr` pointer, identifying the memory you have allocated for the attribute value. If you are uncertain of how much memory to allocate, see the Discussion. On return, *`oValue`* contains a valid pointer to the actual attribute value. If the value is unset, `ATSUGetLineControl` produces the default value in this parameter.

oActualValueSize

A pointer to a `ByteCount` value. On return, the value contains the actual size (in bytes) of the attribute value. You should examine this parameter if you are unsure of the size of the attribute value being obtained.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUGetLineControl` function obtains the value of a specified line control attribute for a given line of text in a text layout object.

Before calling `ATSUGetLineControl`, you should call the function `ATSUGetAllLineControls` (page 63) to obtain an array of nondefault line control attribute tags and value sizes for the line. You can then pass the tag and value size for the attribute value to obtain to `ATSUGetLineControl`.

To obtain the value of a layout control attribute value for a text layout object, call the function `ATSUGetLayoutControl` (page 82).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUGetNativeCurveType

Obtains the type of outline path used for glyphs associated with a given style object.

```
OSStatus ATSUGetNativeCurveType (
   ATSUStyle iATSUStyle,
   ATSCurveType *oCurveType
);
```

Parameters

iATSUStyle

An `ATSUStyle` value specifying the style object to examine.

oCurveType

A pointer to an `ATSCurveType` value. On return, the value provides a constant specifying the type of outline path being used. Possible values include `kATSCubicCurveType`, `kATSQuadCurveType`, and `kATSOtherCurveType`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

You can call the `ATSUGetNativeCurveType` function to obtain the type of outline path used for glyphs associated with a given style object.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeGlyphs.h`

ATSUGetObjFontFallbacks

Obtains the font list and font-search method associated with a font fallback object.

```
OSStatus ATSUGetObjFontFallbacks (
   ATSUFontFallbacks iFontFallbacks,
   ItemCount iMaxFontFallbacksCount,
   ATSUFontID oFonts[],
   ATSUFontFallbackMethod *oFontFallbackMethod,
   ItemCount *oActualFallbacksCount
);
```

Parameters

`iFontFallbacks`
An `ATSUFontFallbacks` value specifying the font fallback object to examine.

`iMaxFontFallbacksCount`
An `ItemCount` value specifying the maximum number of fonts that you want to obtain. Typically, this is equivalent to the size of the array allocated in the `oFonts` parameter. To determine this value, see the Discussion.

`oFonts`
A pointer to memory you have allocated for an array of `ATSUFontID` values. If you are uncertain of how much memory to allocate, see the Discussion. On return, the array contains font IDs identifying the fonts in the font list associated with the font fallback object.

`oFontFallbackMethod`
A pointer to an `ATSUFontFallbackMethod` value. On return, the value identifies the font-search method associated with the font fallback object. See "Font Fallback Methods" (page 214) for a description of possible values.

`oActualFallbacksCount`
A pointer to an `ItemCount` value. On return, the value specifies the actual number of fonts in the font list associated with the text layout object. This value may be greater than that passed in the `iMaxFontFallbacksCount` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetObjFontFallbacks` function obtains the list of fonts and the search order associated with a given font fallback object.

Typically you use the function `ATSUGetObjFontFallbacks` by calling it twice, as follows:

1. Pass valid values for the *`iFontFallbacks`* and *`oActualFallbacksCount`* parameters, `NULL` for the `oFonts` and *`oFontFallbackMethod`* parameters and `0` for the *`iMaxFontFallbacksCount`* parameter. `ATSUGetObjFontFallbacks` returns the size of the font array in the `oActualFallbacksCount` parameter.

2. Allocate enough space for an array of the returned size, then call the function again, passing a valid pointer in the `oFonts` parameter. On return, the array contains the font list associated with the font fallback object.

You set the font list and search method for a font fallback object by calling the function `ATSUSetObjFontFallbacks` (page 126).

Availability
Available in Mac OS X v10.1 and later.
Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetRunStyle

Obtains style run information for a character offset in a run of text.

```
OSStatus ATSUGetRunStyle (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOffset,
   ATSUStyle *oStyle,
   UniCharArrayOffset *oRunStart,
   UniCharCount *oRunLength
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object for which to obtain style run information.

`iOffset`
A pointer to a `UniCharArrayOffset` value. This value should specify the offset from the beginning of the text buffer to the character for which to obtain style run information. To specify the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`.

`oStyle`
A pointer to an `ATSUStyle` value. On return, the value specifies the style object assigned to the range of text containing the character at *`iOffset`*. Note that if you pass an offset in the `iOffset` parameter that is at a style run boundary, `ATSUGetRunStyle` produces style run information for the following, not preceding, style run.

`oRunStart`
A pointer to a `UniCharArrayOffset` value. On return, the value specifies the offset from the beginning of the text buffer to the first character of the style run containing the character at `iOffset`. Note that the entire style run does not necessarily share the same unset attribute values as the character at `iOffset`.

`oRunLength`
A pointer to a `UniCharCount` value. On return, the value specifies the length of the style run containing the character at `iOffset`.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You can use the `ATSUGetRunStyle` function to obtain the style object assigned to a given text offset. `ATSUGetRunStyle` also produces the encompassing text range that shares the style object with the offset.

Note that the style object contains those previously set style attributes, font features, and font variations that are continuous for the range of text that includes the specified text offset. If you want to obtain all shared style information for a style run, including any unset attributes, call the function `ATSUGetContinuousAttributes` (page 66) instead.

If only one style run is set in the text layout object, and it does not cover the entire text layout object, `ATSUGetRunStyle` uses the style run information for the `iOffset` parameter to set the style run information for the remaining text.

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

Declared In
`ATSUnicodeObjects.h`

ATSUGetSoftLineBreaks

Obtains soft line breaks in a range of text.

```
OSStatus ATSUGetSoftLineBreaks (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iRangeStart,
   UniCharCount iRangeLength,
   ItemCount iMaximumBreaks,
   UniCharArrayOffset oBreaks[],
   ItemCount *oBreakCount
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object to examine.

`iRangeStart`
A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the text range to examine. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`, To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iRangeLength` parameter.

`iRangeLength`
A `UniCharCount` value specifying the length of the text range. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

`iMaximumBreaks`
An `ItemCount` value specifying the maximum number of soft line breaks to obtain. Typically, this is equivalent to the number of `UniCharArrayOffset` values for which you have allocated memory in the `oBreaks` array. To determine this value, see the Discussion.

`oBreaks`
A pointer to memory you have allocated for an array of `UniCharArrayOffset` values. On return, the array contains offsets from the beginning of the text buffer to each of the soft line breaks in the text range. If you are uncertain of how much memory to allocate for this array, see the Discussion.

oBreakCount
: A pointer to an `ItemCount` value. On return, the value specifies the actual number of soft line breaks in the range of text. This may be greater than the value you specified in the `iMaximumBreaks` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetSoftLineBreaks` function obtains the soft line breaks that are currently set in a given text range.

Typically you use the function `ATSUGetSoftLineBreaks` by calling it twice, as follows:

1. Pass valid values for the *iTextLayout*, *iRangeStart*, *iRangeLength*, and *oBreakCount* parameters. Pass `NULL` for the `oBreaks` parameter and 0 for the *iMaximumBreaks* parameter. On return, the value of the `oBreakCount` parameter specifies the number of items in the offset array.

2. Allocate enough space for an array of the appropriate size (number of items in the array multiplied by 4 bytes per item), then call the function again, passing a valid pointer in the `oBreaks` parameter. On return, the pointer refers to an array containing the text range's soft line breaks.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUGetStyleRefCon

Obtains application-specific data for a style object.

```
OSStatus ATSUGetStyleRefCon (
   ATSUStyle iStyle,
   URefCon *oRefCon
);
```

Parameters

iStyle
: An `ATSUStyle` value specifying the style object for which to obtain application-specific data.

oRefCon
: A pointer to a 32-bit value. On return, the value contains or refers to application-specific style data.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetStyleRefCon` function obtains a reference constant (that is, application-specific data) associated with a style object. To associate a reference constant with a style object, call the function `ATSUSetStyleRefCon` (page 129).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetTabArray

Retrieves the tab ruler associated with a text layout object.

```
OSStatus ATSUGetTabArray (
   ATSUTextLayout iTextLayout,
   ItemCount iMaxTabCount,
   ATSUTab oTabs[],
   ItemCount *oTabCount
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value specifying the text layout object whose tab ruler you want to obtain.

iMaxTabCount
The maximum number of tabs that can be written to the `iTabs` array.

oTabs[]
An array of `ATSUTab` values. On return, this array contains the current tab values in order of position along the line from left to right. Pass `NULL` if you want to retrieve the number of tabs, but not the tab values.

oTabCount
The number of tabs set for the text layout object.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
This function can be used to retrieve all the tabs that were previously set for a text layout object, using the function `ATSUSetTabArray`. All the returned tabs will be in order of position along the line.Typically you use the `ATSUGetTabArray` function by calling it twice, as follows:

1. Pass `NULL` for the `oTabs` parameter, `0` for the *`iMaxTabCount`* parameter, and valid values for the other parameters. The `ATSUGetTabArray` function returns the actual number of tabs in the `oTabCount` parameter.

2. Allocate enough space for a buffer of the returned size, then call the function again, passing a valid pointer to the buffer in the `oTabs` parameter. On return, the buffer contains the tab values in order of position along the line from left to right.

Availability
Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetTextHighlight

Obtains the highlight region for a range of text.

```
OSStatus ATSUGetTextHighlight (
   ATSUTextLayout iTextLayout,
   ATSUTextMeasurement iTextBasePointX,
   ATSUTextMeasurement iTextBasePointY,
   UniCharArrayOffset iHighlightStart,
   UniCharCount iHighlightLength,
   RgnHandle oHighlightRegion
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value identifying the text layout object containing the text range.

iTextBasePointX

An `ATSUTextMeasurement` value specifying the x-coordinate of the origin (in either the current graphics port or in a Quartz graphics context) of the line containing the text range. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to obtain the highlight region relative to the current pen location in the current graphics port.

iTextBasePointY

An `ATSUTextMeasurement` value specifying the y-coordinate of the origin (in either the current graphics port or graphics context) of the line containing the text range. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to obtain the highlight region relative to the current pen location in the current graphics port.

iHighlightStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range. If the range of text spans multiple lines, you should call `ATSUGetTextHighlight` for each line, passing the offset corresponding to the beginning of the new line with each call. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iHighlightLength` parameter.

iHighlightLength

A `UniCharCount` value specifying the length of the text range. If you want the text range to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

oHighlightRegion

A valid `RgnHandle` value. On return, `ATSUGetTextHighlight` produces a `MacRegion` structure containing the highlight region for the specified range of text. In the case of discontinuous highlighting, the region consists of multiple components, with `MacRegion.rgnBBox` specifying the bounding box around the entire area of discontinuous highlighting.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUGetTextHighlight` function obtains the highlight region for a range of text. To highlight text, call the function `ATSUHighlightText` (page 101).

The `ATSUGetTextHighlight` function uses the previously set line ascent and descent values to calculate the height of the highlight region. If these values have not been set for the line, `ATSUGetTextHighlight` uses the line ascent and descent values set for the text layout object containing the line. If these are not set, it uses the default values.

Version Notes
When there are discontinuous highlighting regions, the structure produced in the `oHighlightRegion` parameter is made up of multiple components. In ATSUI 1.1, the maximum number of components that can be produced is 31. In ATSUI 1.2, the maximum number of components is 127.

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

Declared In
`ATSUnicodeDrawing.h`

ATSUGetTextLayoutRefCon

Obtains application-specific data for a text layout object.

```
OSStatus ATSUGetTextLayoutRefCon (
   ATSUTextLayout iTextLayout,
   URefCon *oRefCon
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object for which to obtain application-specific data.

`oRefCon`
A pointer to a 32-bit value. On return, the value contains or refers to application-specific text layout data.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetTextLayoutRefCon` function obtains a reference constant (that is, application-specific data) associated with a text layout object. To associate a reference constant with a text layout object, call the function `ATSUSetTextLayoutRefCon` (page 130).

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

Declared In
`ATSUnicodeObjects.h`

ATSUGetTextLocation

Obtains information about the text associated with a text layout object.

```
OSStatus ATSUGetTextLocation (
   ATSUTextLayout iTextLayout,
   void **oText,
   Boolean *oTextIsStoredInHandle,
   UniCharArrayOffset *oOffset,
   UniCharCount *oTextLength,
   UniCharCount *oTextTotalLength
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object to examine.

oText
: A pointer to data of any type. On return, the pointer is set to either a pointer or a handle that refers to the text buffer for the specified text layout object.

oTextIsStoredInHandle
: A pointer to a `Boolean` value. On return, the value is set to `true` if the text buffer in the `oText` parameter is accessed by a handle; if `false`, a pointer.

oOffset
: A pointer to a `UniCharArrayOffset` value. On return, the value specifies the offset from the beginning of the text buffer to the first character of the layout's current text range.

oTextLength
: A pointer to a `UniCharCount` value. On return, the value specifies the length of the text range.

oTextTotalLength
: A pointer to a `UniCharCount` value. On return, the value specifies the length of the entire text buffer.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
When you call the `ATSUGetTextLocation` function for a given text layout object, ATSUI obtains the location of the text layout object's associated text in physical memory, the length of the text range and its text buffer, and whether the text is accessed by a pointer or handle.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetTransientFontMatching

Obtains whether ATSUI automatically performs font substitution for a text layout object.

```
OSStatus ATSUGetTransientFontMatching (
   ATSUTextLayout iTextLayout,
   Boolean *oTransientFontMatching
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object to examine.

oTransientFontMatching
: A pointer to a `Boolean` value. On return, the value indicates whether ATSUI performs automatic font substitution for the text layout object. If `true`, ATSUI automatically performs font substitution for the text range associated with the text layout object.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

You can call the `ATSUGetTransientFontMatching` function to find out whether ATSUI automatically performs font substitution for a given text layout object when a character cannot be drawn with the assigned font. To turn automatic font substitution on or off for a text layout object, call the function `ATSUSetTransientFontMatching` (page 132).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUGetUnjustifiedBounds

Obtains the typographic bounding rectangle for a line of text prior to final layout.

```
OSStatus ATSUGetUnjustifiedBounds (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart,
   UniCharCount iLineLength,
   ATSUTextMeasurement *oTextBefore,
   ATSUTextMeasurement *oTextAfter,
   ATSUTextMeasurement *oAscent,
   ATSUTextMeasurement *oDescent
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object to examine.

iLineStart
: A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the line. To indicate that the line starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iLineLength` parameter.

iLineLength
: A `UniCharCount` value specifying the length of the line. If you want the line to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

oTextBefore
: A pointer to an `ATSUTextMeasurement` value. On return, the value specifies the starting point of the typographic bounds for the line, relative to the origin (0,0) of the line and taking into account cross-stream shifting. Note that the `ATSUMeasureText` function might produce negative values for the typographic starting point of the line if, for example, the initial character of the line is allowed to hang into the margin. For horizontal text, this value corresponds to the left side of the bounding rectangle.

oTextAfter
: A pointer to an `ATSUTextMeasurement` value. On return, the value specifies the end point of the typographic bounds for the line, relative to the origin (0,0) of the line and taking into account cross-stream shifting. For horizontal text, this value corresponds to the right side of the bounding rectangle.

oAscent
: A pointer to an `ATSUTextMeasurement` value. On return, the value specifies the ascent of the typographic bounds for the line, relative to the origin (0,0) of the line and taking into account cross-stream shifting. For horizontal text, this value corresponds to the top side of the bounding rectangle.

oDescent
: A pointer to an `ATSUTextMeasurement` value. On return, the value specifies the descent of the typographic bounds for the line, relative to the origin (0,0) of the line and taking into account cross-stream shifting. For horizontal text, this value corresponds to the bottom side of the bounding rectangle.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
There are two kinds of bounds that your application may typically want to obtain for a block of text: typographic bounds and image bounds. The image bounds define the smallest rectangle that completely encloses the filled or framed parts of a block of text—that is, the text's "inked" glyphs. Because of the potential differences in glyph height in a text block, your application may instead need to determine the typographic bounds. The typographic bounding rectangle contains the extra space above and below the image bounding rectangle where characters with ascenders or descenders would be drawn (even if none currently are).

The `ATSUGetUnjustifiedBounds` function calculates the typographic bounds (in coordinates independent of the rendering device) for a line of text. Note that `ATSUGetUnjustifiedBounds` calculates these bounds prior to the text's final layout, and therefore, the calculated bounds might not reflect those of the final laid-out line. To obtain the typographic bounds of a line after it is laid out, you can call the function `ATSUGetGlyphBounds` (page 75).

The `ATSUGetUnjustifiedBounds` function ignores any previously set line attributes such as line rotation, flushness, justification, ascent, and descent in its calculations. You typically only call `ATSUGetUnjustifiedBounds` when you need to find out what the width of a line is without these attributes, such as for determining your own line breaks or the leading and line spacing to impose on a line.

The `ATSUGetUnjustifiedBounds` function treats the specified text range as a single line. That is, if the range of text you specify is less than a line, it nevertheless treats the initial character in the range as the start of a line, for measuring purposes. If the range of text extends beyond a line, `ATSUGetUnjustifiedBounds` ignores soft line breaks, again, treating the text as a single line.

Before calculating the typographic bounds for the text range, the `ATSUGetUnjustifiedBounds` function examines the text layout object to ensure that each of the characters in the range is assigned to a style run. If there are gaps between style runs, `ATSUGetUnjustifiedBounds` assigns the characters in the gap to the style run that precedes (in storage order) the gap. If there is no style run at the beginning of the text range, `ATSUGetUnjustifiedBounds` assigns these characters to the first style run it finds. If there is no style run at the end of the text range, `ATSUGetUnjustifiedBounds` assigns the remaining characters to the last style run it finds.

To obtain the image bounding rectangle of a laid-out line, call the function `ATSUMeasureTextImage` (page 107).

Version Notes
As of ASTUI version 2.4, this function replaces the `ATSUMeasureText` function.

Availability
Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUGlyphGetCubicPaths

Obtains the cubic outline paths for a glyph.

```
OSStatus ATSUGlyphGetCubicPaths (
   ATSUStyle iATSUStyle,
   GlyphID iGlyphID,
   ATSCubicMoveToUPP iMoveToProc,
   ATSCubicLineToUPP iLineToProc,
   ATSCubicCurveToUPP iCurveToProc,
   ATSCubicClosePathUPP iClosePathProc,
   void *iCallbackDataPtr,
   OSStatus *oCallbackResult
);
```

Parameters

`iATSUStyle`
: An `ATSUStyle` value specifying the style object to examine.

`iGlyphID`
: A `GlyphID` value identifying the glyph for which to obtain an outline path.

`iMoveToProc`
: A pointer to your callback function for handling the pen move-to operation.

`iLineToProc`
: A pointer to your callback function for handling the line-to operation.

`iCurveToProc`
: A pointer to your callback function for handling the curve-to operation.

`iClosePathProc`
: A pointer to your callback function for handling the close-path operation.

iCallbackDataPtr

A pointer to any data your callback functions need. This pointer is passed through to your callback functions.

oCallbackResult

On output, a value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetCubicPaths` function stops parsing the glyph path outline and returns the result `kATSOutlineParseAbortedErr`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The glyph outlines that are returned are the hinted outlines at the font size specified in the style object. If you want to use unhinted outlines, set the font size to a very large size, (for example, 1000 points) and then scale down the returned curves to the desired size.

As of Mac OS X version 10.1, the curves returned by this function are derived from quadratic curves, irrespective of the native curve type of the font.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeGlyphs.h`

ATSUGlyphGetCurvePaths

Obtains the outline paths for a glyph associated with a given style object.

```
OSStatus ATSUGlyphGetCurvePaths (
   ATSUStyle iATSUStyle,
   GlyphID iGlyphID,
   ByteCount *ioBufferSize,
   ATSUCurvePaths *oPaths
);
```

Parameters

iATSUStyle

An `ATSUStyle` value specifying the style object to examine.

iGlyphID

A `GlyphID` value identifying the glyph for which to obtain an outline path.

ioBufferSize

A pointer to a `ByteCount` value specifying the size of the buffer you have allocated for the `ATSUCurvePaths` structure in the *oPaths* parameter. On return, the value provides the actual size of buffer needed to contain the produced `ATSUCurvePaths` structure.

oPaths

A pointer to an `ATSUCurvePaths` structure. On return, the `ATSUCurvePaths` structure contains a value specifying the number of contours that comprise the glyph's outline, as well as an array of `ATSUCurvePath` structures, each of which defines a contour.

Return Value
A result code. See "ATSUI Result Codes" (page 234). If the font is a protected font, returns `kATSUInvalidFontErr`.

Discussion
This function only returns quadratic paths. The glyph outlines that are returned are the hinted outlines at the font size specified in the style object. If you want to obtain unhinted outlines, set the font size to a very large size, (for example, 1000 points) and then scale down the returned curves to the desired size. More typically, however, you would use the functions `ATSUGlyphGetCubicPaths` (page 95) and `ATSUGlyphGetQuadraticPaths` (page 98) when drawing curves.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeGlyphs.h`

ATSUGlyphGetIdealMetrics

Obtains resolution-independent font metric information for glyphs associated with a given style object.

```
OSStatus ATSUGlyphGetIdealMetrics (
   ATSUStyle iATSUStyle,
   ItemCount iNumOfGlyphs,
   GlyphID iGlyphIDs[],
   ByteOffset iInputOffset,
   ATSGlyphIdealMetrics oIdealMetrics[]
);
```

Parameters

iATSUStyle
: An `ATSUStyle` value specifying the style object to examine.

iNumOfGlyphs
: An `ItemCount` value specifying the number of glyphs to examine. This value should be the same as the number of glyph IDs being passed in the *iGlyphIDs* parameter and the number of `ATSGlyphIdealMetrics` structures for which memory is allocated in the *oIdealMetrics* parameter.

iGlyphIDs
: A pointer to the first `GlyphID` value in an array of glyph IDs. Each ID should identify a glyph for which to obtain font metric information.

iInputOffset
: A `ByteOffset` value specifying the offset in bytes between glyph IDs in the *iGlyphIDs* array.

oIdealMetrics
: A pointer to memory you have allocated for an array of `ATSGlyphIdealMetrics` structures. On return, each structure contains advance and side-bearing values for a glyph.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The advance width is the full horizontal width of the glyph as measured from its origin to the origin of the next glyph on the line, including the left-side and right-side bearings. For vertical text, the advance height is the sum of the top-side bearing, the bounding-box height, and the bottom-side bearing.

You can call the `ATSUGlyphGetIdealMetrics` function to obtain an array of `ATSGlyphIdealMetrics` structures containing values for the specified glyphs' advance and side bearings. `ATSUGlyphGetIdealMetrics` can analyze both horizontal and vertical text, automatically producing the appropriate bearing values (oriented for width or height, respectively) for each.

You should call `ATSUGlyphGetIdealMetrics` to obtain resolution-independent glyph metrics. To obtain device-adjusted (that is, resolution-dependent) glyph metrics, call the function `ATSUGlyphGetScreenMetrics` (page 99).

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

Declared In
`ATSUnicodeGlyphs.h`

ATSUGlyphGetQuadraticPaths

Obtains the quadratic outline paths for a glyph.

```
OSStatus ATSUGlyphGetQuadraticPaths (
   ATSUStyle iATSUStyle,
   GlyphID iGlyphID,
   ATSQuadraticNewPathUPP iNewPathProc,
   ATSQuadraticLineUPP iLineProc,
   ATSQuadraticCurveUPP iCurveProc,
   ATSQuadraticClosePathUPP iClosePathProc,
   void *iCallbackDataPtr,
   OSStatus *oCallbackResult
);
```

Parameters

`iATSUStyle`
An `ATSUStyle` value specifying the style object to examine.

`iGlyphID`
A `GlyphID` value identifying the glyph for which to obtain an outline path.

`iNewPathProc`
A pointer to your callback function for handling the new-path operation.

`iLineProc`
A pointer to your callback function for handling the line operation.

`iCurveProc`
A pointer to your callback function for handling the curve operation.

`iClosePathProc`
A pointer to your callback function for handling the close-path operation.

`iCallbackDataPtr`
A pointer to any data your callback functions need. This pointer is passed through to your callback functions.

oCallbackResult

On output, a value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetQuadraticPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The glyph outlines that are returned are the hinted outlines at the font size specified in the style object. If you want to use unhinted outlines, set the font size to a very large size, (for example, 1000 points) and then scale down the returned curves to the desired size.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeGlyphs.h`

ATSUGlyphGetScreenMetrics

Obtains device-adjusted font metric information for glyphs associated with a given style object.

```
OSStatus ATSUGlyphGetScreenMetrics (
   ATSUStyle iATSUStyle,
   ItemCount iNumOfGlyphs,
   GlyphID iGlyphIDs[],
   ByteOffset iInputOffset,
   Boolean iForcingAntiAlias,
   Boolean iAntiAliasSwitch,
   ATSGlyphScreenMetrics oScreenMetrics[]
);
```

Parameters

iATSUStyle

An `ATSUStyle` value specifying the style object to examine.

iNumOfGlyphs

An `ItemCount` value specifying the number of glyphs to examine. This value should be the same as the number of glyph IDs being passed in the *iGlyphIDs* parameter and the number of `ATSGlyphScreenMetrics` structures for which memory is allocated in the *oScreenMetrics* parameter.

iGlyphIDs

A pointer to the first `GlyphID` value in an array of glyph IDs. Each ID should identify a glyph for which to obtain font metric information.

iInputOffset

A `ByteOffset` value specifying the offset in bytes between glyph IDs in the *iGlyphIDs* array.

iForcingAntiAlias

A `Boolean` value indicating whether anti-aliasing is forced for the style object.

iAntiAliasSwitch

A `Boolean` value indicating whether anti-aliasing is currently on or off.

oScreenMetrics

A pointer to memory you have allocated for an array of `ATSGlyphScreenMetrics` structures. On return, each structure contains device-adjusted metrics for a glyph, including advance and side bearings, but also values for the top left, height, and width of the glyph.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

You can call the `ATSUGlyphGetScreenMetrics` function to obtain an array of `ATSGlyphScreenMetrics` structures containing values for the specified glyphs' advance and side bearings, top left, height, and width.

You should call `ATSUGlyphGetScreenMetrics` to obtain device-adjusted (that is, resolution-dependent) glyph metrics. To obtain resolution-independent glyph metrics, call the function `ATSUGlyphGetIdealMetrics` (page 97).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeGlyphs.h`

ATSUHighlightInactiveText

Highlights previously selected text using an alpha value of 0.5.

```
OSStatus ATSUHighlightInactiveText (
   ATSUTextLayout iTextLayout,
   ATSUTextMeasurement iTextBasePointX,
   ATSUTextMeasurement iTextBasePointY,
   UniCharArrayOffset iHighlightStart,
   UniCharCount iHighlightLength
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value identifying the text layout object containing the text range.

iTextBasePointX

An `ATSUTextMeasurement` value specifying the x-coordinate of the origin (in either the current graphics port or in a Quartz graphics context) of the line containing the text range. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to obtain the highlight region relative to the current pen location in the current graphics port.

iTextBasePointY

An `ATSUTextMeasurement` value specifying the y-coordinate of the origin (in either the current graphics port or graphics context) of the line containing the text range. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to obtain the highlight region relative to the current pen location in the current graphics port.

iHighlightStart
: A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range. If the range of text spans multiple lines, you should call `ATSUGetTextHighlight` for each line, passing the offset corresponding to the beginning of the new line with each call. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iHighlightLength` parameter.

iHighlightLength
: A `UniCharCount` value specifying the length of the text range. If you want the text range to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Availability
Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUHighlightText

Renders a highlighted range of text at a specified location in a QuickDraw graphics port or Quartz graphics context.

```
OSStatus ATSUHighlightText (
   ATSUTextLayout iTextLayout,
   ATSUTextMeasurement iTextBasePointX,
   ATSUTextMeasurement iTextBasePointY,
   UniCharArrayOffset iHighlightStart,
   UniCharCount iHighlightLength
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value identifying the text layout object for which to render highlighted text.

iTextBasePointX
: An `ATSUTextMeasurement` value specifying the x-coordinate of the origin (in either the current graphics port or in a Quartz graphics context) of the line containing the text range to highlight. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to draw relative to the current pen location in the current graphics port.

iTextBasePointY
: An `ATSUTextMeasurement` value specifying the y-coordinate of the origin (in either the current graphics port or graphics context) of the line containing the text range to highlight. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to draw relative to the current pen location in the current graphics port.

iHighlightStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range to highlight. If the range of text spans multiple lines, you should call `ATSUHighlightText` for each line, passing the offset corresponding to the beginning of the new line to draw with each call. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iHighlightLength` parameter.

iHighlightLength

A `UniCharCount` value specifying the length of the text range to highlight. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

When the user selects a series of glyphs, the characters in memory corresponding to the glyphs make up the selection range and should be highlighted to indicate where the next editing operation is to occur. The characters in a selection range are always contiguous in memory, but their corresponding glyphs are not necessarily so onscreen. If the selection range crosses a direction boundary, it is appropriate to display discontinuous highlighting.

The `ATSUHighlightText` function renders a highlighted range of text at a specified location in a QuickDraw graphics port or Quartz graphics context, using the highlight information in the graphics port or context. `ATSUHighlightText` automatically produces discontinuous highlighting, if needed. You typically call the `ATSUHighlightText` function every time you need to draw or redraw highlighted text.

If you provide your own `CGContextRef` (for example, one created by calling the function `QDBeginCGContext`) for an `ATSUTextLayout`, highlighting performed by calling the function `ATSUHighlightText` will not work unless you first call the function `ATSUSetHighlightingMethod` with the *`iMethod`* parameter set to `kRedrawHighlighting` and a pointer to an `ATSUUnhighlightData` structure as the `iUnhighlightData` parameter.

Before drawing the highlighted text, `ATSUHighlightText` examines the text layout object to ensure that each of the characters in the range is assigned to a style run. If there are gaps between style runs, ATSUI assigns the characters in the gap to the style run that precedes (in storage order) the gap. If there is no style run at the beginning of the text range, ATSUI assigns these characters to the first style run it finds. If there is no style run at the end of the text range, ATSUI assigns the remaining characters to the last style run it finds.

`ATSUHighlightText` uses the previously set line ascent and descent values to calculate the height of the highlighted region. If these values have not been set for the line, `ATSUHighlightText` uses the line ascent and descent values set for the text layout object containing the line. If these are not set, it uses the default values.

To draw a highlighted text range that spans multiple lines, you should call `ATSUHighlightText` for each line of the text range, even if all the lines are in the same text layout object. You should adjust the `iHighlightStart` parameter to reflect the beginning of each line to be drawn.

After calling `ATSUHighlightText`, to properly redraw the unhighlighted text and background, you should always call the function `ATSUUnhighlightText` (page 140).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSULeftwardCursorPosition

Obtains the memory offset for the insertion point to the left of the high caret position, as determined by a move of the specified length at a line direction boundary.

```
OSStatus ATSULeftwardCursorPosition (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOldOffset,
   ATSUCursorMovementType iMovementType,
   UniCharArrayOffset *oNewOffset
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value identifying the text layout object to examine.

iOldOffset
A `UniCharArrayOffset` value specifying the memory offset corresponding to the current caret position. To specify the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`. For bidirectional text, you can specify the previous layout by passing the constant `kATSUFromPreviousLayout` and the following layout by passing the constant `kATSUFromFollowingLayout`. See the Discussion for example code that shows how to use these constants.

iMovementType
An `ATSUCursorMovementType` constant identifying the unit of movement. See "Caret Movement Types" (page 208) for a description of possible values (which range from a single Unicode character to a Unicode word in length). Note that ATSUI may not be able to move the caret by a single Unicode character in some cases, since doing so might place the insertion point in the middle of a surrogate pair.

oNewOffset
A pointer to a `UniCharArrayOffset` value. On return, the value provides the memory offset corresponding to the new insertion point. This offset may be outside the initial text buffer.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
Line direction boundaries can occur on the trailing edges of two glyphs, the leading edges of two glyphs, or at the beginning or end of a text segment. At direction boundaries, a single insertion point in memory can require two caret positions onscreen, one for text entry in each direction. The two separate carets (known as a split caret or a dual caret) consist of a high caret and a low caret. The high (primary) caret is displayed at the caret position for inserting text whose direction corresponds to the line direction (the dominant direction for the overall line of text). The low (secondary) caret is displayed at the caret position for inserting text whose direction is counter to the overall line direction.

The `ATSURightwardCursorPosition` function obtains the memory offset for the insertion point to the left of the high caret position, as determined by a move of the specified length at a line direction boundary.

You should use the `ATSULeftwardCursorPosition` function or the function `ATSURightwardCursorPosition` (page 118) to determine caret position when the user presses the right and left arrow keys.

Except in the case of Indic text (and other cases where the font rearranges the glyphs), for left-to-right text, calling the function `ATSULeftwardCursorPosition` has the same effect as calling `ATSUPreviousCursorPosition` (page 117). For right-to-left text, calling the function `ATSULeftwardCursorPosition` has the same effect as calling `ATSUNextCursorPosition` (page 109).

The following code shows how to use the constants `kATSUFromPreviousLayout` and `kATSUFromFollowingLayout` with the function `ATSULeftwardCursorPosition`:

```
typedef struct TLayoutWithEndOffset
{
    ATSUTextLayout      layout;
    UInt32              endOffset;
};

typedef struct TLayoutsWithEndOffsets
{
    UInt32                  count;
    TLayoutWithEndOffset    layouts[];
}

UniCharArrayOffset MyAbsoluteToRelativeOffset (
                TLayoutsWithEndOffsets * iLayouts,
                UniCharArrayOffset iAbsoluteOffset );
UniCharArrayOffset MyRelativeToAbsoluteOffset (
                TLayoutsWithEndOffsets * iLayouts,
                UInt32 iLayoutIndex,
                UniCharArrayOffset iRelativeOffset );
UniCharArrayOffset MyGetLayoutEndOffset (
                TLayoutsWithEndOffsets * iLayouts,
                UInt32 iLayoutIndex );

/* Passing in current offset relative to the beginning of */
/* the entire text buffer (absolute), */
/* not just the current paragraph. This returns the new (absolute)  */
/* offset relative to the beginning of the entire text buffer.*/
UniCharArrayOffset
MyLeftwardCursorPosition ( TLayoutsWithEndOffsets * iLayouts,
                    UInt32 iLayoutIndex,
                    UniCharArrayOffset iAbsoluteOffset,
                    ATSUCursorMovementType iType )
{
    OSStatus            status;
    UInt32              newLayoutIndex = iLayoutIndex;
    UniCharArrayOffset  newRelativeOffset;

    status = ATSULeftwardCursorPosition(
                iLayouts->layouts[iLayoutIndex].layout,
                MyAbsoluteToRelativeOffset (iLayouts, iAbsoluteOffset  ),
                iType, &newRelativeOffset );

    if ( status == noErr )
    {
        /* If the API returns the same value as */
        /* that passed in then we're at */
        /* the edge of the layout so need to move */
        /* to the adjacent layout. f */
        /* If that value is zero then we're moving to the previous  layout. */
        /* (This is left-to-right text.) */
        if ( (newRelativeOffset == iRelativeOffset) &&
                (iRelativeOffset == 0) )
        {
            /* Don't want to move before the first layout! */
            if ( iLayoutIndex != 0 )
            {
                /* Pass kATSUFromFollowingLayout to the previous  */
                /* ATSUTextLayout. */
                /* Note that the returned offset is relative to
                /* the ATSUTextLayout passed in here.*/
                newLayoutIndex--;
                status = ATSULeftwardCursorPosition(
                            iLayouts[newLayoutIndex],
                            kATSUFromFollowingLayout,
                            iType &newRelativeOffset );
            }
        }
        else
        {
            UniCharArrayOffset  endAbsoluteOffset = MyGetLayoutEndOffset(
                                 iLayouts, iLayoutIndex );

            /* We've moved to the very end of this layout */
            /* (past the trailing carriage return presumably) */
            /* so we're moving to the following layout. */
            /* Make sure we aren't at the */
            /* end of the text buffer. (This is right-to-left text.)  */
            if ( (newRelativeOffset == MyAbsoluteToRelativeOffset  (
                        iLayouts, endAbsoluteOffset )) &&
                       (iLayoutIndex != iLayouts->count) )
            {
                newLayoutIndex++;
                status = ATSULeftwardCursorPosition(
                             iLayouts->layouts[newLayoutIndex],
                             kATSUFromPreviousLayout, iType,
                             &newRelativeOffset );

                /* If we're moving from one paragraph to the following  one */
                /* and we aren't at the beginning of the layout  means
                /* that we're moving to a left-to-right */
                /* paragraph and we must back up one so that*/
                /* we're just before the line ending whitespace
                /* (space or <CR>), unless the */
                /* following layout is the last one. */
                if ( (newRelOffset > 0) && (newLayoutIndex  !=
                        iLayouts->count) )
                    newRelativeOffset--;
            }
        }
    }
    return MyRelativeToAbsoluteOffset( iLayouts, newLayoutIndex,
                     newRelativeOffset );
}
```

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUMatchFontsToText

Examines a text range for characters that cannot be drawn with the current font and suggests a substitute font, if necessary.

```
OSStatus ATSUMatchFontsToText (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iTextStart,
   UniCharCount iTextLength,
   ATSUFontID *oFontID,
   UniCharArrayOffset *oChangedOffset,
   UniCharCount *oChangedLength
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object to examine.

iTextStart
: A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the range to examine. To start at the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`.

iTextLength
: A `UniCharCount` value specifying the length of the text range to examine. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

oFontID
: A pointer to a `ATSUFontID` value. On return, the value provides a font ID for the suggested substitute font or `kATSUInvalidFontID`, if no substitute font is available.

oChangedOffset
: A pointer to a `UniCharArrayOffset` value. On return, this value specifies the offset from the beginning of the text buffer to the first character that cannot be drawn with the current font.

oChangedLength
: A pointer to a `UniCharCount` value. On return, this value specifies the length of the text range that cannot be drawn with the current font.

Return Value
A result code. See "ATSUI Result Codes" (page 234). The result code `noErr` indicates that all the characters in the given range can be rendered with their current font(s) and no font substitution is needed. If you receive either of the result codes `kATSUFontsMatched` or `kATSUFontsNotMatched`, you should update the input range and call `ATSUMatchFontsToText` again to ensure that all the characters in the range can be drawn.

Discussion
When you call the `ATSUMatchFontsToText` function, ATSUI scans the given range of text for characters that cannot be drawn with the currently assigned font. When ATSUI finds such a character, it identifies a substitute font for drawing the character. ATSUI then continues scanning the text range for subsequent characters that cannot be drawn, stopping when it

- finds a character that can be drawn with the currently assigned font, or
- finds a character that cannot be drawn with either the currently assigned font or the substitute font, or

- reaches the end of the text range you have specified

ATSUI's default behavior for finding a substitute font is to recommend the first valid font that it finds when scanning the fonts in the user's system. ATSUI first searches in the standard application fonts for various languages. If that fails, ATSUI searches through the remaining fonts on the system in the order in which the Font Manager returns the fonts. After ATSUI has searched all the fonts in the system, any unmatched text is drawn using the last-resort font. That is, missing glyphs are represented by and empty box to indicate to the user that a valid font for that character is not installed on their system. You can alter ATSUI's default search behavior by calling the function `ATSUCreateFontFallbacks` (page 39) and defining your own font fallback settings for the text layout object.

So, for example, if the subrange of text for which you wanted to perform font substitution was the text "abcde", and the characters 'c' and 'd' could not be drawn with the current font, but could be drawn with font X, and the character 'e' either could be drawn with the current font or could not be drawn with font X, then `ATSUMatchFontsToText` produces the ID of font X in the `oFont` parameter and sets the `oChangedOffset` parameter to 2 and the `oChangedLength` parameter to 2.

Because ATSUI does not necessarily completely scan the text range you specify with each call to `ATSUMatchFontsToText`, if ATSUI does find any characters that cannot be rendered with their current font, you should call `ATSUMatchFontsToText` again and update the input range to check that all the subsequent characters in the range can be drawn. For that reason, you should call `ATSUMatchFontsToText` from within a loop to assure that the entire range of text is checked.

Note that calling `ATSUMatchFontsToText` does not cause the suggested font substitution to be performed. If you want ATSUI to perform font substitution for you, you can call the function `ATSUSetTransientFontMatching` (page 132).

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

Declared In
`ATSUnicodeObjects.h`

ATSUMeasureTextImage

Obtains the image bounding rectangle for a line of text after final layout.

```
OSStatus ATSUMeasureTextImage (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineOffset,
   UniCharCount iLineLength,
   ATSUTextMeasurement iLocationX,
   ATSUTextMeasurement iLocationY,
   Rect *oTextImageRect
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object to examine.

iLineOffset
A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the line to examine. To indicate that the specified line starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iLineLength` parameter.

iLineLength
A `UniCharCount` value specifying the length of the text range. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`. However, the image bounds is restricted to the line in which `iLineOffset` resides.

iLocationX
An `ATSUTextMeasurement` value specifying the x-coordinate of the line's origin in the current graphics port or Quartz graphics context. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), for the dimensions of the bounds relative to the current pen location in the current graphics port or graphics context. You can pass `0` to obtain only the dimensions of the bounding rectangle relative to one another, not their actual onscreen position.

iLocationY
An `ATSUTextMeasurement` value specifying the y-coordinate of the line's origin in the current graphics port or Quartz graphics context. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), for the dimensions of the bounds relative to the current pen location in the current graphics port or graphics context. You can pass `0` to obtain only the dimensions of the bounding rectangle relative to one another, not their actual onscreen position.

oTextImageRect
A pointer to a `Rect` structure. On return, the structure contains the dimensions of the image bounding rectangle for the text, offset by the values specified in the `iLocationX` and `iLocationY` parameters. If the line is rotated, the sides of the rectangle are parallel to the coordinate axis.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUMeasureTextImage` function obtains the image bounds of a laid-out line of text. These bounds are described by the smallest rectangle that completely encloses the filled or framed parts of a block of text—that is, the text's "inked" glyphs.

In measuring the line, the `ATSUMeasureTextImage` function takes into account line rotation, alignment, and justification, as well as other characteristics that affect layout, such as hanging punctuation. (If the line is rotated, the sides of the rectangle are parallel to the coordinate axes and encompass the rotated line.) If no attributes are set for the line, `ATSUMeasureTextImage` uses the global attributes set for the text layout object.

Because the height of the image bounding rectangle is determined by the actual device metrics, `ATSUMeasureTextImage` ignores any previously set line ascent and descent values for the line it is measuring.

Before calculating the image bounds for the text range, the `ATSUMeasureTextImage` function examines the text layout object to ensure that each of the characters in the range is assigned to a style run. If there are gaps between style runs, `ATSUMeasureTextImage` assigns the characters in the gap to the style run that precedes (in storage order) the gap. If there is no style run at the beginning of the text range, the `ATSUMeasureTextImage` function assigns these characters to the first style run it finds. If there is no style run at the end of the text range, `ATSUMeasureTextImage` assigns the remaining characters to the last style run it finds.

To obtain the final typographic bounds of a line, call the function `ATSUGetGlyphBounds` (page 75). To calculate the unjustified typographic bounds of a line, call the function `ATSUGetUnjustifiedBounds` (page 93).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUNextCursorPosition

Obtains the memory offset for the insertion point that follows the current insertion point in storage order, as determined by a move of the specified length.

```
OSStatus ATSUNextCursorPosition (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOldOffset,
   ATSUCursorMovementType iMovementType,
   UniCharArrayOffset *oNewOffset
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value identifying the text layout object to examine.

iOldOffset
A `UniCharArrayOffset` value specifying the memory offset corresponding to the current caret position. To specify the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`.

iMovementType
An `ATSUCursorMovementType` constant identifying the unit of movement. See "Caret Movement Types" (page 208) for a description of possible values (which range from a single Unicode character to a Unicode word in length). Note that ATSUI may not be able to move the caret by a single Unicode character in some cases, since doing so might place the insertion point in the middle of a surrogate pair.

oNewOffset
A pointer to a `UniCharArrayOffset` value. On return, the value provides the memory offset corresponding to the following insertion point. This offset may be outside the initial text buffer.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUNextCursorPosition` function obtains the memory offset for the insertion point that follows the current insertion point in storage order, as determined by a move of the specified length.

You should use the `ATSUNextCursorPosition` function or the function `ATSUPreviousCursorPosition` (page 117) to determine caret position when the initial memory offset is not at a line direction boundary. If the initial offset is at a line direction boundary, you should instead use the functions `ATSURightwardCursorPosition` (page 118) or `ATSULeftwardCursorPosition` (page 103).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUOffsetToCursorPosition

Obtains the caret position(s) corresponding to a memory offset, after a move of the specified length.

```
OSStatus ATSUOffsetToCursorPosition (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOffset,
   Boolean iIsLeading,
   ATSUCursorMovementType iMovementType,
   ATSUCaret *oMainCaret,
   ATSUCaret *oSecondCaret,
   Boolean *oCaretIsSplit
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value identifying the text layout object to examine.

iOffset
A `UniCharArrayOffset` value specifying the memory offset corresponding to the glyph edge nearest the event, after a movement of the specified type. You can obtain this value by examining the offset produced in the `ioPrimaryOffset` parameter of the function `ATSUPositionToCursorOffset` (page 113).

iIsLeading
A `Boolean` value indicating whether the specified offset corresponds to the leading or trailing edge of the glyph. You can obtain this information from the function `ATSUPositionToCursorOffset` (page 113). This value is relevant if the offset occurs at a line direction boundary or within a glyph cluster.

iMovementType
An `ATSUCursorMovementType` constant identifying the unit of cursor movement. See "Caret Movement Types" (page 208) for a description of possible values (which range from a single Unicode character to a Unicode word in length). Note that ATSUI may not be able to move the cursor by a single Unicode character in some cases, since doing so might place the cursor in the middle of a surrogate pair.

oMainCaret
A pointer to an `ATSUCaret` structure. On return, the structure contains the starting and ending pen locations of the high caret if the value produced in the `oCaretIsSplit` parameter is `true`. If the value is `false`, the structure contains the starting and ending pen locations of the main caret.

oSecondCaret
A pointer to an `ATSUCaret` structure. On return, the structure contains the starting and ending pen locations of the low caret if the value passed back in the `oCaretIsSplit` parameter is `true`. If the value is `false`, the structure contains the starting and ending pen locations of the main caret (that is, the same values as the `oMainCaret` parameter).

oCaretIsSplit
A pointer to a `Boolean` value. On return, the value indicates whether the offset specified in the `iOffset` parameter occurs at a line direction boundary. If `true`, the offset occurs at a line direction boundary; otherwise, `false`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Availability

Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeDrawing.h`

ATSUOffsetToPosition

Obtains the caret position(s) corresponding to a memory offset.

```
OSStatus ATSUOffsetToPosition (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOffset,
   Boolean iIsLeading,
   ATSUCaret *oMainCaret,
   ATSUCaret *oSecondCaret,
   Boolean *oCaretIsSplit
);
```

Parameters

`iTextLayout`

An `ATSUTextLayout` value identifying the text layout object to examine.

`iOffset`

A `UniCharArrayOffset` value specifying the memory offset for which to obtain the corresponding caret position. To respond to a mouse-down event, pass the offset produced in the `ioPrimaryOffset` parameter of the function `ATSUPositionToOffset` (page 115)—that is, the offset corresponding to the glyph edge closest to the event.

`iIsLeading`

A `Boolean` value indicating whether the offset corresponds to the leading or trailing edge of the glyph. You can obtain this information from the function `ATSUPositionToOffset` (page 115). This value is relevant if the offset occurs at a line direction boundary or within a glyph cluster.

`oMainCaret`

A pointer to an `ATSUCaret` structure. On return, the structure contains the starting and ending pen locations of the high caret if the value produced in `oCaretIsSplit` is `true`. If the value is `false`, the structure contains the starting and ending pen locations of the main caret.

`oSecondCaret`

A pointer to an `ATSUCaret` structure. On return, the structure contains the starting and ending pen locations of the low caret if the value passed back in the `oCaretIsSplit` parameter is `true`. If the value is `false`, the structure contains the starting and ending pen locations of the main caret (that is, the same values as the `oMainCaret` parameter).

`oCaretIsSplit`

A pointer to a `Boolean` value. On return, the value indicates whether the offset specified in the `iOffset` parameter occurs at a line direction boundary. If `true`, the offset occurs at a line direction boundary; otherwise, `false`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion
The process of hit-testing text obtains the location of a mouse-down event relative both to the position of onscreen glyphs and to the corresponding offset between character codes in memory. You can then use the location information obtained by hit-testing to set the insertion point (that is, the caret) or selection range (for highlighting).

Hit-testing text is complicated by the fact that a given line of text may be bidirectional. Therefore, the onscreen order of glyphs may not readily correspond to the storage order of the corresponding character codes. And the concept of which glyph comes "first" in a line of text cannot always be limited to the visual terms "left" and "right." Because of these complexities, it is more accurate to speak in terms of "leading" and "trailing" edges to glyphs. A "leading edge" is defined as the edge of a glyph that you first encounter when you read the text that includes that glyph. For example, when reading Roman text, you first encounter the left edge of a Roman glyph. Similarly, the "trailing edge" is defined as the edge of the glyph encountered last.

ATSUI can translate the location of a mouse click into an onscreen position, as well as to a memory offset. When you use ATSUI for hit-testing, ATSUI takes into account the glyph edge (whether leading or trailing) nearest to where the click occurred, thus providing positional information in complex situations, such as at line direction boundaries or within glyph clusters.

Line direction boundaries can occur on the trailing edges of two glyphs, the leading edges of two glyphs, or at the beginning or end of a text segment. At direction boundaries, a single insertion point in memory can require two caret positions onscreen, one for text entry in each direction. The two separate carets (known as a split caret or a dual caret) consist of a high caret and a low caret. The high (primary) caret is displayed at the caret position for inserting text whose direction corresponds to the line direction (the dominant direction for the overall line of text). The low (secondary) caret is displayed at the caret position for inserting text whose direction is counter to the overall line direction.

The first step in obtaining the caret position(s) for a mouse-down event is to pass the location (in local coordinates, relative to the line origin) of the event to the function `ATSUPositionToOffset` (page 115). The `ATSUPositionToOffset` function produces the memory offset corresponding to the glyph edge nearest the event. If the mouse-down event occurs at a line direction boundary or within a glyph cluster, the `ATSUPositionToOffset` function produces two offsets. You can then provide the offset(s) to the `ATSUOffsetToPosition` function, to obtain the actual caret position(s) for the event.

The `ATSUOffsetToPosition` function produces two structures of type `ATSUCaret`. These structures contain the pen positioning information needed to draw the caret(s) for the event, specified relative to the origin of the line in the current graphics port or graphics context. Specifically, the `ATSUCaret` structures contain x-y coordinates for both the caret's starting and ending pen positions (the latter taking into account line rotation, caret slanting, and split-caret appearances).

If the memory offset you pass to `ATSUOffsetToPosition` is at a line boundary, the structure produced in the `oMainCaret` parameter contains the starting and ending pen locations for the high caret, while the `oSecondCaret` parameter contains the corresponding values for the low caret. If the offset is not at a line boundary, both parameters contain the starting and ending pen locations of the main caret.

Because you provide the `ATSUOffsetToPosition` function an offset relative to the origin of the line where the hit occurred, `ATSUOffsetToPosition` produces positioning information that is also relative. Therefore, you must transform the positions produced by the `ATSUOffsetToPosition` function before drawing the caret(s). To transform the caret location(s), add the starting and ending caret coordinates to the coordinates of the origin of the line in which the hit occurred. For example, if `ATSUOffsetToPosition` produces starting and ending pen locations of (25,0), (25,25) in the `oMainCaret` parameter (and the `oSecondCaret` parameter contains the same coordinates, meaning that the caret was not split), you would add these to the position of the origin of the line in the graphics port or context. If the position of the line origin was at (50,50), then the starting and ending pen locations of the caret would be (75,50), (75,75).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUOverwriteAttributes

Copies to a destination style object the nondefault style attribute settings of a source style object.

```
OSStatus ATSUOverwriteAttributes (
   ATSUStyle iSourceStyle,
   ATSUStyle iDestinationStyle
);
```

Parameters

`iSourceStyle`
An `ATSUStyle` value specifying the style object from which to copy nondefault style attributes.

`iDestinationStyle`
An `ATSUStyle` value specifying the style object containing the style attributes to be overwritten.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUOverwriteAttributes` function copies all nondefault style attribute values from a source style object to a destination style object. The source object's nondefault values are applied to the destination object whether or not the destination object also has nondefault values for the copied attributes. All other settings in the destination style object are left unchanged.

`ATSUOverwriteAttributes` does not copy the contents of memory referenced by pointers within custom style attributes or within reference constants. You are responsible for ensuring that this memory remains valid until both the source and destination style objects are disposed of.

To create a style object that contains all the contents of another style object, call the function `ATSUCreateAndCopyStyle` (page 38). To copy all the style attributes (including any default settings) of a style object into an existing style object, call the function `ATSUCopyAttributes` (page 32). To copy style attributes that are set in the source but not in the destination style object, call the function `ATSUUnderwriteAttributes` (page 138).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUPositionToCursorOffset

Obtains the memory offset for the glyph edge nearest a mouse-down event, after a move of the specified length.

```
OSStatus ATSUPositionToCursorOffset (
   ATSUTextLayout iTextLayout,
   ATSUTextMeasurement iLocationX,
   ATSUTextMeasurement iLocationY,
   ATSUCursorMovementType iMovementType,
   UniCharArrayOffset *ioPrimaryOffset,
   Boolean *oIsLeading,
   UniCharArrayOffset *oSecondaryOffset
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value identifying the text layout object in which the mouse-down event occurred.

iLocationX

An `ATSUTextMeasurement` value specifying the x-coordinate of the event, in local coordinates, relative to the origin of the line where the event occurred. That is, to specify the x-coordinate value, you should subtract the x-coordinate of the line origin from the x-coordinate of the event (in local coordinates). You can pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), for the location of the mouse-down event relative to the current pen location in the current graphics port.

iLocationY

An `ATSUTextMeasurement` value specifying the y-coordinate of the event, in local coordinates, relative to the origin of the line where the event occurred. That is, to specify the y-coordinate value, you should subtract the y-coordinate of the line origin from the y-coordinate of the event (in local coordinates). You can pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), for the location of the mouse-down event relative to the current pen location in the current graphics port.

iMovementType

An `ATSUCursorMovementType` constant identifying the unit of movement. See "Caret Movement Types" (page 208) for a description of possible values (which range from a single Unicode character to a Unicode word in length). Note that ATSUI may not be able to move the caret by a single Unicode character in some cases, since doing so might place the insertion point in the middle of a surrogate pair.

ioPrimaryOffset

A pointer to a `UniCharArrayOffset` value specifying the offset corresponding to the beginning of the line where the event occurred. On return, the value specifies the offset corresponding to the glyph edge nearest the event, after a movement of the specified type. This offset corresponds to where the insertion point would be placed after the move. To determine whether this offset indicates the leading or trailing edge of the glyph, you can examine the value produced in the *oIsLeading* parameter.

oIsLeading

A pointer to a `Boolean` value. On return, the value indicates whether the offset produced in the `ioPrimaryOffset` parameter is leading or trailing. The `ATSUPositionToOffset` function produces a value of `true` if the offset is leading (that is, more closely associated with the subsequent character in memory). It produces a value of `false` if the offset is trailing (that is, more closely associated with the preceding character in memory).

oSecondaryOffset

A pointer to a `UniCharArrayOffset` value. On return, the value typically specifies the same offset as that produced in the *ioPrimaryOffset* parameter, unless the event occurred within a glyph cluster or at a line direction boundary. If so, the value specifies the secondary offset, for the glyph edge furthest from the event.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUPositionToCursorOffset` function produces the memory offset for the glyph edge nearest a mouse-down event, after a move of the specified length. This offset corresponds to where an insertion point would be placed after the move.

Availability
Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUPositionToOffset

Obtains the memory offset for the glyph edge nearest a mouse-down event.

```
OSStatus ATSUPositionToOffset (
   ATSUTextLayout iTextLayout,
   ATSUTextMeasurement iLocationX,
   ATSUTextMeasurement iLocationY,
   UniCharArrayOffset *ioPrimaryOffset,
   Boolean *oIsLeading,
   UniCharArrayOffset *oSecondaryOffset
);
```

Parameters

`iTextLayout`
: An `ATSUTextLayout` value identifying the text layout object in which the mouse-down event occurred.

`iLocationX`
: An `ATSUTextMeasurement` value specifying the x-coordinate of the event, in local coordinates, relative to the origin of the line where the event occurred. That is, to specify the x-coordinate value, you should subtract the x-coordinate of the line origin from the x-coordinate of the hit point (in local coordinates). You can pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), for the location of the mouse-down event relative to the current pen location in the current graphics port.

`iLocationY`
: An `ATSUTextMeasurement` value specifying the y-coordinate of the event, in local coordinates, relative to the origin of the line where the event occurred. That is, to specify the y-coordinate value, you should subtract the y-coordinate of the line origin from the y-coordinate of the hit point (in local coordinates). You can pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), for the location of the mouse-down event relative to the current pen location in the current graphics port.

`ioPrimaryOffset`
: A pointer to a `UniCharArrayOffset` value specifying the offset corresponding to the beginning of the line where the event occurred. On return, the value specifies the offset corresponding to the glyph edge that is visually closest to the event. To determine whether this offset indicates the leading or trailing edge of the glyph, you can examine the value produced in the *`oIsLeading`* parameter.

oIsLeading

A pointer to a `Boolean` value. On return, the value indicates whether the offset produced in the `ioPrimaryOffset` parameter is leading or trailing. The function `ATSUPositionToOffset` produces a value of `true` if the offset is leading (that is, more closely associated with the subsequent character in memory). It produces a value of `false` if the offset is trailing (that is, more closely associated with the preceding character in memory).

oSecondaryOffset

A pointer to a `UniCharArrayOffset` value. On return, the value typically specifies the same offset as that produced in the *ioPrimaryOffset* parameter, unless the event occurred within a glyph cluster or at a line direction boundary. If so, the value specifies a secondary offset. The secondary offset is associated with the glyph that has a different direction from the primary line direction.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The process of hit-testing text obtains the location of a mouse-down event relative both to the position of onscreen glyphs and to the corresponding offset between character codes in memory. You can then use the location information obtained by hit-testing to set the insertion point (that is, the caret) or selection range (for highlighting).

Hit-testing text is complicated by the fact that a given line of text may be bidirectional. Therefore, the onscreen order of glyphs may not readily correspond to the storage order of the corresponding character codes. And the concept of which glyph comes "first" in a line of text cannot always be limited to the visual terms "left" and "right." Because of these complexities, it is more accurate to speak in terms of "leading" and "trailing" edges to glyphs. A "leading edge" is defined as the edge of a glyph that you first encounter when you read the text that includes that glyph. For example, when reading Roman text, you first encounter the left edge of a Roman glyph. Similarly, the "trailing edge" is defined as the edge of the glyph encountered last.

ATSUI can translate the location of a mouse click into an onscreen position, as well as to a memory offset. When you use ATSUI for hit-testing, ATSUI takes into account the glyph edge (whether leading or trailing) nearest to where the click occurred, thus providing positional information in complex situations, such as at line direction boundaries or within glyph clusters.

The first step in obtaining the caret position(s) for a mouse-down event is to pass the location (in local coordinates, relative to the line origin) of the event to the function `ATSUPositionToOffset`. For example, if you have a mouse-down event whose position in local coordinates is (75,50), you would subtract this value from the position of the origin of the line in the current graphics port. If the position of the origin of the line in the current graphics port is (50,50), then the relative position of the event that you would pass in the `iLocationX` and `iLocationY` parameters is (25,0).

The `ATSUPositionToOffset` function produces the memory offset corresponding to the glyph edge nearest the event. If the mouse-down event occurs at a line direction boundary or within a glyph cluster, `ATSUPositionToOffset` produces two offsets. You can then provide the offset(s) to the `ATSUOffsetToPosition` (page 111) function, to obtain the actual caret position(s) for the event.

When you call the `ATSUPositionToOffset` function, ATSUI examines the Unicode directionality of the character corresponding to the event location. The `ATSUPositionToOffset` function produces a value of `true` in the *oIsLeading* parameter if the offset is leading (that is, more closely associated with the subsequent character in memory and therefore indicative of a left-to-right line direction). It produces a value of `false` if the offset is trailing (that is, more closely associated with the preceding character in memory and indicative of a right-to-left line direction).

Finally, note that when the event occurs beyond the leftmost or rightmost caret positions of the line (not taking into account line rotation), such that no glyph corresponds to the location of the hit, the `ATSUPositionToOffset` function produces the primary offset of the closest edge of the line to the input location. The `oIsLeading` flag depends on the directionality of the closest glyph and the side of the line to which the input location is closest. In this case, the secondary offset is equal to the primary offset, since no glyph was hit.

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

Declared In
`ATSUnicodeDrawing.h`

ATSUPreviousCursorPosition

Obtains the memory offset for the insertion point that precedes the current insertion point in storage order, as determined by a move of the specified length.

```
OSStatus ATSUPreviousCursorPosition (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOldOffset,
   ATSUCursorMovementType iMovementType,
   UniCharArrayOffset *oNewOffset
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value identifying the text layout object to examine.

iOldOffset
A `UniCharArrayOffset` value specifying the memory offset corresponding to the current caret position. To specify the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`,

iMovementType
An `ATSUCursorMovementType` constant identifying the unit of movement. See "Caret Movement Types" (page 208) for a description of possible values (which range from a single Unicode character to a Unicode word in length). Note that ATSUI may not be able to move the caret by a single Unicode character in some cases, since doing so might place the insertion point in the middle of a surrogate pair.

oNewOffset
A pointer to a `UniCharArrayOffset` value. On return, the value provides the memory offset corresponding to the preceding insertion point. This offset may be outside the initial text buffer.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUPreviousCursorPosition` function obtains the memory offset for the insertion point that precedes the current insertion point in storage order, as determined by a move of the specified length.

You should use the `ATSUPreviousCursorPosition` function or the function `ATSUNextCursorPosition` (page 109) to determine caret position when the initial offset is not at a line direction boundary. If the initial offset is at a line direction boundary, you should instead use the functions `ATSURightwardCursorPosition` (page 118) or `ATSULeftwardCursorPosition` (page 103).

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSURightwardCursorPosition

Obtains the memory offset for the insertion point to the right of the high caret position, as determined by a move of the specified length at a line direction boundary.

```
OSStatus ATSURightwardCursorPosition (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iOldOffset,
   ATSUCursorMovementType iMovementType,
   UniCharArrayOffset *oNewOffset
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value identifying the text layout object to examine.

iOldOffset
A `UniCharArrayOffset` value specifying the memory offset corresponding to the current caret position. To specify the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`. For bidirectional text, you can specify the previous layout by passing the constant `kATSUFromPreviousLayout` and the following layout by passing the constant `kATSUFromFollowingLayout`. See the Discussion for the function `ATSULeftwardCursorPosition` (page 103) for an example of how these constants can be used.

iMovementType
An `ATSUCursorMovementType` constant identifying the unit of movement. See "Caret Movement Types" (page 208) for a description of possible values (which range from a single Unicode character to a Unicode word in length). Note that ATSUI may not be able to move the caret by a single Unicode character in some cases, since doing so might place the insertion point in the middle of a surrogate pair.

oNewOffset
A pointer to a `UniCharArrayOffset` value. On return, the value provides the memory offset corresponding to the new insertion point. This offset may be outside the initial text buffer.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
Line direction boundaries can occur on the trailing edges of two glyphs, the leading edges of two glyphs, or at the beginning or end of a text segment. At direction boundaries, a single insertion point in memory can require two caret positions onscreen, one for text entry in each direction. The two separate carets (known as a split caret or a dual caret) consist of a high caret and a low caret. The high (primary) caret is displayed at the caret position for inserting text whose direction corresponds to the line direction (the dominant direction for the overall line of text). The low (secondary) caret is displayed at the caret position for inserting text whose direction is counter to the overall line direction.

The `ATSURightwardCursorPosition` function obtains the memory offset for the insertion point to the right of the high caret position, as determined by a move of the specified length at a line direction boundary.

You should use the `ATSURightwardCursorPosition` function or the function `ATSULeftwardCursorPosition` (page 103) to determine caret position when the user presses the right and left arrow keys.

Except in the case of Indic text (and other cases where the font rearranges the glyphs), for left-to-right text, calling the function `ATSURightwardCursorPosition` has the same effect as calling `ATSUNextCursorPosition` (page 109). For right-to-left text, calling the function `ATSURightwardCursorPosition` has the same effect as calling `ATSUPreviousCursorPosition` (page 117).

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

Declared In
`ATSUnicodeDrawing.h`

ATSUSetAttributes

Sets style attribute values in a style object.

```
OSStatus ATSUSetAttributes (
   ATSUStyle iStyle,
   ItemCount iAttributeCount,
   const ATSUAttributeTag iTag[],
   const ByteCount iValueSize[],
   const ATSUAttributeValuePtr iValue[]
);
```

Parameters

`iStyle`
An `ATSUStyle` value specifying the style object for which to set attributes.

`iAttributeCount`
An `ItemCount` value specifying the number of attributes to set. This value should correspond to the number of elements in the `iTag` and `iValueSize` arrays.

`iTag`
A pointer to the initial `ATSUAttributeTag` value in an array of attribute tags. Each element in the array must contain a valid style attribute tag that corresponds to the style attribute value to set. Note that an attribute tag cannot be used in versions of the Mac OS that are earlier than the version in which the tag was introduced. For example, a tag available in Mac OS version 10.2 cannot be used in Mac OS version 10.1 or earlier. You can call the function Gestalt to check version information for ATSUI. See "Attribute Tags" (page 196) for a description of the Apple-defined style attribute tag constants and for availability information.

`iValueSize`
A pointer to the initial `ByteCount` value in an array of attribute value sizes. Each element in the array must contain the size (in bytes) of the corresponding style run attribute value being set. `ATSUSetAttributes` sets style attributes after confirming the sizes in the array.

`iValue`
A pointer to the initial `ATSUAttributeValuePtr` value in an array of attribute value pointers. Each pointer in the array must reference an attribute value corresponding to a tag in the `iTag` array. The value referenced by the pointer must be legal for that tag.

Return Value
A result code. See "ATSUI Result Codes" (page 234). If there is a function error, `ATSUSetAttributes` does not set any attributes in the style object.

Discussion
Style attributes are a collection of values and settings that override the font-specified behavior for displaying and formatting text in a style run. To specify a style attribute, ATSUI uses a "triple" consisting of (1) an attribute tag, (2) a value for that tag, and (3) the size of the value.

The `ATSUSetAttributes` function enables you to set multiple style attribute values for a style object. When you call `ATSUSetAttributes`, any style attributes that you do not set retain their previous values. To set font features and font variations, call the functions `ATSUSetFontFeatures` (page 120) and `ATSUSetVariations` (page 133), respectively.

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

Declared In
`ATSUnicodeObjects.h`

ATSUSetFontFeatures

Sets font features in a style object.

```
OSStatus ATSUSetFontFeatures (
   ATSUStyle iStyle,
   ItemCount iFeatureCount,
   const ATSUFontFeatureType iType[],
   const ATSUFontFeatureSelector iSelector[]
);
```

Parameters

`iStyle`
An `ATSUStyle` value specifying the style object for which to set font features.

`iFeatureCount`
An `ItemCount` value specifying the number of font features to set. This value should correspond to the number of elements in the `iType` and `iSelector` arrays.

`iType`
A pointer to the initial `ATSUFontFeatureType` value in an array of feature types. Each element in the array must contain a valid feature type that corresponds to a feature selector in the `iSelector` array. To obtain the valid feature types for a font, call the function `ATSUGetFontFeatureTypes` (page 70).

`iSelector`
A pointer to the initial `ATSUFontFeatureSelector` value in an array of feature selectors. Each element in the array must contain a valid feature selector that corresponds to a feature type in the `iType` array. To obtain the valid feature selectors for a font, call the function `ATSUGetFontFeatureSelectors` (page 68).

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUSetFontFeatures` function enables you to set multiple font features for a style object. Any unset font features retain their font-defined default values. To set style attributes and font variations for a style object, call the functions `ATSUSetAttributes` (page 119) and `ATSUSetVariations` (page 133), respectively.

The constants that represent font feature types are defined in the header file `SFNTLayoutTypes.h`. When you use ATSUI to access and set font features, you must use the constants defined in this header file, which are described in *Inside Mac OS X: Rendering Unicode Text With ATSUI*. As feature types can be added at any time, you should check Apple's font feature registry website for the most up-to-date list of font feature types and selectors: http://developer.apple.com/fonts/Registry/index.html.

Version Notes

Prior to ATSUI 1.2, `ATSUSetFontFeatures` does not remove contradictory font features. You are responsible for maintaining your own list and removing contradictory settings when they occur. Beginning with ATSUI 1.2, `ATSUSetFontFeatures` removes contradictory font features if they are set.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeFonts.h`

ATSUSetHighlightingMethod

Sets the method ATSUI uses to highlight and unhighlight text for a text layout object.

```
OSStatus ATSUSetHighlightingMethod (
   ATSUTextLayout iTextLayout,
   ATSUHighlightMethod iMethod,
   const ATSUUnhighlightData *iUnhighlightData
);
```

Parameters

`iTextLayout`

An `ATSUTextLayout` value identifying the text layout object for which to set the highlighting method.

`iMethod`

An `ATSUHighlightMethod` value specifying the type of highlighting for ATSUI to use (`kInvertHighlighting` or `kRedrawHighlighting`). The default highlighting method, if you do not call `ATSUSetHighlightingMethod`, is inversion. See "Highlight Methods" (page 219) for a description of available values.

`iUnhighlightData`

A pointer to an `ATSUUnhighlightData` structure if you are setting the *`iMethod`* parameter to `kRedrawHighlighting` or `NULL` if setting `iMethod` to `kInvertHighlighting`. Before calling `ATSUSetHighlightingMethod`, you should set the `ATSUUnhighlightData` structure to contain the data needed (either a color or a UPP for a background drawing callback) to redraw the background.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

In Mac OS 9 and by default in Mac OS X (except with Cocoa applications—see below), ATSUI highlights text by "inverting" the region containing the text, that is, its background color. Although inversion provides satisfactory highlighting in most cases, it does not always provide the best result for grayscale text. (Mac OS X sets a lower threshold for antialiasing, while in Mac OS 9 grayscale text can be turned off by the user.)

In Mac OS X, when using a Quartz graphics context, you can instruct ATSUI to use the redraw method of highlighting, rather than simple inversion. (Note that Cocoa applications always use the redraw method of highlighting.) The redraw method allows for accurate highlighting of more complex backgrounds, such as those containing multiple colors, patterns, or pictures. To set redrawing on, call the `ATSUSetHighlightingMethod` function and specify that the redraw method be used (by passing `kRedrawHighlighting` in the *`iMethod`* parameter).

If you specify the redraw method of highlighting when you call `ATSUSetHighlightingMethod`, then you must also specify how the background is to be redrawn when the function `ATSUUnhighlightText` (page 140) is called. ATSUI can restore the desired background in one of two ways, depending on the background's complexity:

- When the background is a single color (such as white), ATSUI can readily unhighlight the background. In such a case, you specify the background color that ATSUI uses by calling `ATSUSetHighlightingMethod` and setting `iUnhighlightData.dataType` to `kATSUBackgroundColor` and providing the background color in `iUnhighlightData.unhighlightData`. With these settings defined, when you call `ATSUUnhighlightText`, ATSUI simply calculates the previously highlighted area, repaints it with the specified background color, and then redraws the text.
- When the background is more complex (containing, for example, multiple colors, patterns, or pictures), you must provide a redraw background callback function when you call `ATSUSetHighlightingMethod`. You do this by setting `iUnhighlightData.dataType` to `kATSUBackgroundCallback` and providing a `RedrawBackgroundUPP` in `iUnhighlightData.unhighlightData`. Then when you call `ATSUUnhighlightText` and ATSUI calls your callback, you are responsible for redrawing the background of the unhighlighted area. If you choose to also redraw the text, then your callback should return `false` as a function result. If your callback returns `true` ATSUI redraws any text that needs to be redrawn. See `RedrawBackgroundProcPtr` (page 165) for additional information.

Version Notes
Mac OS 9 applications cannot use the redraw method of highlighting and must use the inversion method, instead.

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

Not available to 64-bit applications.

Declared In
`ATSUnicodeDrawing.h`

ATSUSetLayoutControls

Sets layout control attribute values in a text layout object.

```
OSStatus ATSUSetLayoutControls (
   ATSUTextLayout iTextLayout,
   ItemCount iAttributeCount,
   const ATSUAttributeTag iTag[],
   const ByteCount iValueSize[],
   const ATSUAttributeValuePtr iValue[]
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object for which to set layout control attributes.

iAttributeCount
: An `ItemCount` value specifying the number of attributes to set. This value should correspond to the number of elements in the `iTag` and `iValueSize` arrays.

iTag
: A pointer to the initial `ATSUAttributeTag` value in an array of layout control attribute tags. Each element in the array must contain a valid tag that corresponds to the layout control attribute to set. See "Attribute Tags" (page 196) for a description of the Apple-defined layout control attribute tag constants.

iValueSize
: A pointer to the initial `ByteCount` value in an array of attribute value sizes. Each element in the array must contain the size (in bytes) of the corresponding layout control attribute being set. `ATSUSetLayoutControls` sets layout attributes after confirming the sizes in the array.

iValue
: A pointer to the initial `ATSUAttributeValuePtr` value in an array of attribute value pointers. Each value in the array must correspond to a tag in the `iTag` array and be a legal value for that tag.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
When you use ATSUI to image your text, you can control the text's display and formatting at a number of different levels.

One level is that of the entire text range associated with your text layout object, also known as the "layout level." To affect display and formatting on this level, you can specify various layout control attributes using the `ATSUSetLayoutControls` function. These attributes affect the width of the text area from margin to margin, the alignment of the text, its justification, rotation, and direction, as well as other layout options.

Another level is that of a single line of text, that is, the "line level." To affect display and formatting on this level, you specify various line control attributes via the function `ATSUSetLineControls` (page 124). These attributes are similar to those that you can apply on a full-layout basis, but each affects only an individual text line.

Given that ATSUI allows you to control similar aspects of the display and formatting of your text at either the line level or the layout level (or both, or neither), it is up to you to decide how much layout control to take. However, you should note the following:

- Setting layout control attributes overrides the corresponding default layout-level settings for a text layout object. Any layout attributes that you do not set retain the default values described in "Attribute Tags" (page 196).

- Setting line control attributes overrides the corresponding layout-level settings (whether set or at default values) for a text layout object. This is true even if you set the layout-level attributes subsequently to the line-level ones.
- From a performance standpoint, it is preferable to work from the layout level and not specify layout line by line unless necessary.

Finally, it is also possible to control the display and formatting of your text at the level of an individual character or "run" of characters. At this level, you customize layout by manipulating style settings in a style object. Among the character-level aspects you can control are style attributes (such as font size and color), font features (such as ligatures), and font variations (such as continually varying font weights or widths). However, there are certain line control attributes (specified via the `ATSLineLayoutOptions` flags) that can override style attributes applied to the same text.

Similarly to style attributes, you use a "triple" to specify a line or layout control attribute. That is, an attribute tag, the value of the attribute it sets, and the size (in bytes) of the attribute value. Attribute tags are constants supplied by ATSUI. Attribute values may be a scalar, a structure, or a pointer.

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

Declared In
`ATSUnicodeObjects.h`

ATSUSetLineControls

Sets layout control attribute values for a single line in a text layout object.

```
OSStatus ATSUSetLineControls (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart,
   ItemCount iAttributeCount,
   const ATSUAttributeTag iTag[],
   const ByteCount iValueSize[],
   const ATSUAttributeValuePtr iValue[]
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object for which to set line control attribute values.

`iLineStart`
A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the first character of the line for which to set control attributes.

`iAttributeCount`
An `ItemCount` value specifying the number of attributes to set. This value should correspond to the number of elements in the `iTag` and `iValueSize` arrays.

`iTag`
A pointer to the initial `ATSUAttributeTag` value in an array of line control attribute tags. Each element in the array must contain a valid tag that corresponds to the line control attribute to set. See "Attribute Tags" (page 196) for a description of the Apple-defined line control attribute tag constants.

iValueSize

A pointer to the initial `ByteCount` value in an array of attribute value sizes. Each element in the array must contain the size (in bytes) of the corresponding line control attribute being set. `ATSUSetLineControls` sets line attributes after confirming the sizes in the array.

iValue

A pointer to the initial `ATSUAttributeValuePtr` value in an array of attribute value pointers. Each value in the array must correspond to a tag in the `iTag` array and be a legal value for that tag.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

When you use ATSUI to image your text, you can control the text's display and formatting at a number of different levels. One level is that of the entire text range associated with your text layout object, also known as the "layout level." To affect display and formatting on this level, you can specify various layout control attributes using the `ATSUSetLayoutControls` (page 122) function. These attributes affect the width of the text area from margin to margin, the alignment of the text, its justification, rotation, and direction, as well as other layout options.

Another level is that of a single line of text, that is, the "line level." To affect display and formatting on the line level, you specify various line control attributes using the function `ATSUSetLineControls`. These attributes are similar to those that you can apply on a full-layout basis, but each affects only an individual text line.

You can break text into lines by calling the functions `ATSUBatchBreakLines` (page 20) or `ATSUBreakLine` (page 22). You can define separate lines of text by specifying soft breaks either by

- calling the function `ATSUBatchBreakLines`
- calling the function `ATSUBreakLine` with the `iUseAsSoftBreak` parameter set to `true`
- specifying the soft line breaks using the function `ATSUSetSoftLineBreak`

Given that ATSUI allows you to control similar aspects of the display and formatting of your text at either the line level or the layout level (or both, or neither), it is up to you to decide how much layout control to take. However, you should note the following:

- Setting layout control attributes overrides the corresponding default layout-level settings for a text layout object. Any layout attributes that you do not set retain the default values described in "Attribute Tags" (page 196).
- Setting line control attributes overrides the corresponding layout-level settings (whether set or at default values) for a text layout object. This is true even if you set the layout-level attributes subsequently to the line-level ones. Any line attributes that you do not set retain their default values.
- From a performance standpoint, it is preferable to work from the layout level and not specify layout line by line unless necessary.

Finally, it is also possible to control the display and formatting of your text at the level of an individual character or "run" of characters. At this level, you customize layout by manipulating style settings in a style object. Among the character-level aspects you can control are style attributes (such as font size and color), font features (such as ligatures), and font variations (such as continually varying font weights or widths). However, there are certain line control attributes (specified via the `ATSLineLayoutOptions` flags) that can override style attributes applied to the same text.

Similarly to style attributes, you use a “triple” to specify a line or layout control attribute. That is, an attribute tag, the value of the attribute it sets, and the size (in bytes) of the attribute value. Attribute tags are constants supplied by ATSUI. Attribute values may be a scalar, a structure, or a pointer.

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

Declared In
`ATSUnicodeObjects.h`

ATSUSetObjFontFallbacks

Assigns a font list and a font-search method to a font fallback object.

```
OSStatus ATSUSetObjFontFallbacks (
   ATSUFontFallbacks iFontFallbacks,
   ItemCount iFontFallbacksCount,
   const ATSUFontID iFonts[],
   ATSUFontFallbackMethod iFontFallbackMethod
);
```

Parameters

`iFontFallbacks`
An `ATSUFontFallbacks` value specifying the font fallback object for which to define settings.

`iFontFallbacksCount`
An `ItemCount` value specifying the number of fonts that ATSUI is to search. This value is typically equal to the number of font IDs you are providing in the *`iFonts`* array.

`iFonts`
A pointer to the first `ATSUFontID` value in an array of font IDs identifying the fonts ATSUI is to search.

`iFontFallbackMethod`
An `ATSUFontFallbackMethod` value identifying the order in which ATSUI is to search. See “Font Fallback Methods” (page 214) for a description of possible values.

Return Value
A result code. See “ATSUI Result Codes” (page 234).

Discussion
The `ATSUSetObjFontFallbacks` function defines the settings for a font fallback object. These settings determine the font list and search order that ATSUI uses when seeking substitute fonts for the text layout object with which the font fallback object is associated.

Creating, defining settings for, and associating a font fallback object with a text layout object is the only way to ensure that ATSUI uses your preferred font fallback settings for your text. To create a font fallback object, you first call the function `ATSUCreateFontFallbacks` (page 39). You then define settings for the object by calling the `ATSUSetObjFontFallbacks` function. To associate the font fallback object with a text layout object call the function `ATSUSetLayoutControls` (page 122). You pass these functions the control attribute value `kATSULineFontFallbacksTag` to set the font fallback object.

If you do not call `ATSUSetObjFontFallbacks` to change ATSUI’s default search behavior, ATSUI searches all the fonts on the system sequentially and uses the first valid font it finds for a substitute. If you are careful in ordering the fonts that you supply to `ATSUSetObjFontFallbacks`, you can minimize the time ATSUI needs to find a substitute font.

Font fallback settings affect the behavior of the function `ATSUMatchFontsToText` (page 106) and of font selection during layout and drawing when the function `ATSUSetTransientFontMatching` (page 132) is set to on.

To obtain the font list and font-search method associated with a font fallback object, call the function `ATSUGetObjFontFallbacks` (page 85).

Availability
Available in Mac OS X v10.1 and later.
Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUSetRunStyle

Defines a style run by associating style information with a run of text.

```
OSStatus ATSUSetRunStyle (
   ATSUTextLayout iTextLayout,
   ATSUStyle iStyle,
   UniCharArrayOffset iRunStart,
   UniCharCount iRunLength
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying a text layout object with an associated text buffer. `ATSUSetRunStyle` assigns a style object to a run of text in this buffer.

`iStyle`
An `ATSUStyle` value specifying the style object to associate with the text run.

`iRunStart`
A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the text run.

`iRunLength`
A `UniCharCount` value specifying the length of the text run.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
A text run consists of one or more characters that are contiguous in memory. If you associate these characters with a distinct style, you define a style run. You can use the `ATSUSetRunStyle` function to define a style run, by associating a style object with a run of text in a text layout object. There is a limit of 64K different styles for each ATSUI text layout object. Each text run must be assigned its own style object, which may or may not differ from other style objects assigned to other text runs in a given text layout object.

You can create a new style object containing only default settings by calling the function `ATSUCreateStyle` (page 40). To make changes to the default style attributes, you can call the function `ATSUSetAttributes` (page 119). To set font features and font variations, call the functions `ATSUSetFontFeatures` (page 120) and `ATSUSetVariations` (page 133), respectively.

Note that if you call `ATSUSetRunStyle` on a text run that is already associated with a style object, the style set by `ATSUSetRunStyle` overrides the previous style. Additionally, upon completion, `ATSUSetRunStyle` adjusts the lengths of any style runs on either side of the affected style run.

For example, you may currently have a run of text, 40 characters long, that is assigned a single style, styleA. If you call `ATSUSetRunStyle`, you can reassign characters at offset 10–29 to a new style, styleB. If you do so, you would then have three style runs, where there once was one: characters at offset 0–9 (styleA), 10–29 (styleB), and 30–39 (styleA).

After calling `ATSUSetRunStyle`, you can call the function `ATSUDrawText` (page 50) to display the styled text. When you call `ATSUDrawText`, if you have not previously assigned styles to all the characters you request to be drawn, ATSUI automatically does so. Specifically, ATSUI extends the first style it locates immediately prior (in storage order) to the unstyled characters to include those unassigned characters. If the unstyled characters are at the beginning of the text stream, ATSUI finds the first style run in the stream and extends it backward to the first character.

You should call `ATSUSetRunStyle` whenever you create a new text layout object without any associated styles, as by using the function `ATSUCreateTextLayout` (page 41). You should also call `ATSUSetRunStyle` to assign a style to a text run in response to a user action, such as when the user selects a run of text and changes the font.

You do not need to call `ATSUSetRunStyle` when you change style attributes or text layout attributes. In such cases, ATSUI automatically updates the layout of the text as appropriate.

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

Declared In
`ATSUnicodeObjects.h`

ATSUSetSoftLineBreak

Sets a soft line break that you specify.

```
OSStatus ATSUSetSoftLineBreak (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineBreak
);
```

Parameters

`iTextLayout`
  An `ATSUTextLayout` value specifying the text layout object for which to set a line break.

`iLineBreak`
  A `UniCharArrayOffset` value specifying the offset from the beginning of the text layout object's text buffer to the line break to set.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUSetSoftLineBreak` function enables you to set a soft line break in a text range. You should typically only call `ATSUSetSoftLineBreak` to set line breaks when you are using your own line-breaking algorithm to calculate these breaks. For optimal performance, you should use `ATSUBatchBreakLines` (page 20) to both calculate and set soft line breaks in your text.

After calling `ATSUSetSoftLineBreak`, you should call the function `ATSUGetUnjustifiedBounds` (page 93) to determine whether the characters still fit within the line, which is necessary due to end-of-line effects such as swashes.

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

Declared In
`ATSUnicodeDrawing.h`

ATSUSetStyleRefCon

Sets application-specific data for a style object.

```
OSStatus ATSUSetStyleRefCon (
   ATSUStyle iStyle,
   URefCon iRefCon
);
```

Parameters

`iStyle`
An `ATSUStyle` value specifying the style object for which to set application-specific data.

`iRefCon`
A 32-bit value containing or referring to application-specific style data.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUSetStyleRefCon` function associates a reference constant (that is, application-specific data) with a style object. If you copy or clear a style object that contains a reference constant, the reference constant is neither copied nor removed. To obtain application-specific data for a style object, call the function `ATSUGetStyleRefCon` (page 88).

When you dispose of a style object that contains a reference constant, you are responsible for freeing any memory allocated for the reference constant. Calling the function `ATSUDisposeStyle` (page 49) does not do so.

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

Declared In
`ATSUnicodeObjects.h`

ATSUSetTabArray

Sets a tab ruler for a text layout object.

```
OSStatus ATSUSetTabArray (
   ATSUTextLayout iTextLayout,
   const ATSUTab iTabs[],
   ItemCount iTabCount
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object for which you want to set a tab ruler.

iTabs[]
: An array of the tab values you want applied to the text layout object. This tab ruler is applied to all lines in the text layout object. You can pass `NULL` if `iTabCount` equals `0`. Passing `NULL` effectively deletes any tab ruler that was set previously.

iTabCount
: The number of tabs in the given `iTabs` array. If value is `0`, any previously-set tab ruler is cleared from the text layout object.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

When a tab ruler is set for a text layout object, ATSUI automatically aligns text such that any tabs in the text are laid out to follow the tab ruler's specifications. If you want to use tabs in your text and you also want to use the function `ATSUBatchBreakLines`, then you must set tabs by calling the function `ATSUSetTabArray`.

Availability

Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUSetTextLayoutRefCon

Sets application-specific data for a text layout object.

```
OSStatus ATSUSetTextLayoutRefCon (
   ATSUTextLayout iTextLayout,
   URefCon iRefCon
);
```

Parameters

iTextLayout
: An `ATSUTextLayout` value specifying the text layout object for which to set application-specific data.

iRefCon
: A 32-bit value containing or referring to application-specific text layout data.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUSetTextLayoutRefCon` function associates a reference constant (that is, application-specific data) with a text layout object. You might typically use `ATSUSetTextLayoutRefCon` to track user preferences that can effect layout, for example.

If you copy or clear a text layout object containing a reference constant, the reference constant is not copied or removed. When you dispose of a text layout object that contains a reference constant, you are responsible for freeing any memory allocated for the reference constant. Calling the function `ATSUDisposeTextLayout` (page 49) does not do so.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeObjects.h`

ATSUSetTextPointerLocation

Associates text with a text layout object or updates previously associated text.

```
OSStatus ATSUSetTextPointerLocation (
   ATSUTextLayout iTextLayout,
   ConstUniCharArrayPtr iText,
   UniCharArrayOffset iTextOffset,
   UniCharCount iTextLength,
   UniCharCount iTextTotalLength
);
```

Parameters

`iTextLayout`

An `ATSUTextLayout` value specifying the text layout object for which to set text.

`iText`

A pointer of type `ConstUniCharArrayPtr`, referring to a text buffer containing UTF-16–encoded text. ATSUI associates this buffer with the text layout object and analyzes the complete text of the buffer when obtaining the layout context for the current text range. Thus, for paragraph-format text, if you specify a buffer containing less than a complete paragraph, some of ATSUI's layout results are not guaranteed to be accurate. For example, with a buffer of less than a full paragraph, ATSUI can neither reliably obtain the context for bidirectional processing nor reliably generate accent attachments and ligature formations.

`iTextOffset`

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range to include in the layout. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iTextLength` parameter.

`iTextLength`

A `UniCharCount` value specifying the length of the text range. Note that `iTextOffset + iTextLength` must be less than or equal to the value of the *`iTextTotalLength`* parameter. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

`iTextTotalLength`

A `UniCharCount` value specifying the length of the entire text buffer. This value should be greater than or equal to the range of text defined by the `iTextLength` parameter.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

For ATSUI to render your text, you must associate the text with both a text layout object and style information. Some functions, such as `ATSUCreateTextLayoutWithTextPtr` (page 42), create a text layout object and associate text with it concurrently. However, if you use the function `ATSUCreateTextLayout` (page 41) to create a text layout object, you must assign text to the object prior to attempting most ATSUI operations.

You can use the function `ATSUSetTextPointerLocation` or to associate text with a text layout object. When you call this function, you are both assigning a text buffer to a text layout object and specifying the current text subrange within the buffer to include in the layout.

If there is already text associated with a text layout object, calling `ATSUSetTextPointerLocation` overrides the previously associated text, as well as clearing the object's layout caches. You would typically only call this function for a text layout object with existing associated text if either (a) both the buffer itself is relocated and a subrange of the buffer's text is deleted or inserted or (b) when associating an entirely different buffer with a text layout object.

Note that, because ATSUI objects retain state, doing superfluous calling can degrade performance. For example, you could call `ATSUSetTextPointerLocation` rather than `ATSUTextInserted` (page 136) when the user simply inserts a subrange of text within a text buffer, but there would be a performance penalty, as all the layout caches are flushed by `ATSUSetTextPointerLocation`, rather than just the affected ones.

Similarly, you should not call `ATSUSetTextPointerLocation`, when an entire text buffer associated with a text layout object is relocated, but no other changes have occurred that would affect the buffer's current subrange. Instead, you should call `ATSUTextMoved` (page 137), which is a more focused function and therefore more efficient.

After associating text with a text layout object, use `ATSUSetRunStyle` (page 127) to associate style information with the text. You can then call the function `ATSUDrawText` (page 50) to display the text or a subrange of the text.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUSetTransientFontMatching

Turns automatic font substitution on or off for a text layout object.

```
OSStatus ATSUSetTransientFontMatching (
   ATSUTextLayout iTextLayout,
   Boolean iTransientFontMatching
);
```

Parameters

iTextLayout

An ATSUTextLayout value specifying the text layout object for which to set automatic font substitution on or off.

iTransientFontMatching

A Boolean value indicating whether ATSUI is to perform automatic font substitution for the text layout object. If you pass true, ATSUI performs automatic font substitution for the text range associated with the text layout object.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

Calling the ATSUSetTransientFontMatching function sets ATSUI's automatic font substitution to on or off for a given text layout object. When automatic font substitution is on, ATSUI scans the text range associated with specified text layout object looking for undrawable characters whenever a layout is performed, for example, when text is measured or drawn. When ATSUI finds a character that cannot be drawn with the currently assigned font, it identifies a valid font for the character and draws the character. ATSUI continues scanning the text range for characters in need of substitute fonts, replacing the font and redrawing the characters as needed. ATSUI stops scanning when it reaches the end of the text range associated with the text layout object.

ATSUI's default behavior for finding a substitute font is to use the first valid font that it finds when sequentially scanning the fonts in the user's system. However, you can alter this behavior by calling the function ATSUCreateFontFallbacks (page 39) and defining your own font fallback settings for the text layout object. If ATSUI cannot find any suitable replacement fonts, it substitutes the missing-character glyph—that is, a glyph representing an empty box—to indicate to the user that a valid font is not installed on their system.

Note that when ATSUSetTransientFontMatching performs font substitution, it does not change the font attribute in the associated style object. That is, the font attribute for the style object associated with the redrawn character(s) remains set to the invalid font—not the valid substitute font— just as it was prior to calling ATSUSetTransientFontMatching.

If you want ATSUI to identify a substitute font, but you do not want ATSUI to automatically perform the font substitution, you can call the function ATSUMatchFontsToText (page 106).

Availability

Available in Mac OS X v10.0 and later.

Declared In

ATSUnicodeObjects.h

ATSUSetVariations

Sets font variation axes and values in a style object.

```
OSStatus ATSUSetVariations (
   ATSUStyle iStyle,
   ItemCount iVariationCount,
   const ATSUFontVariationAxis iAxes[],
   const ATSUFontVariationValue iValue[]
);
```

Parameters

iStyle

An `ATSUStyle` value specifying the style object for which to set font variation values.

iVariationCount

An `ItemCount` value specifying the number of font variation values to set. This value should correspond to the number of elements in the `iAxes` and `iValue` arrays.

iAxes

A pointer to the initial `ATSUFontVariationAxis` value in an array of font variation axes. Each element in the array must represent a valid variation axis tag that corresponds to a variation value in the *iValue* array. To obtain a valid variation axis tag for a font, you can call the functions `ATSUGetIndFontVariation` (page 81) or `ATSUGetFontInstance` (page 72).

iValue

A pointer to the initial `ATSUFontVariationValue` value in an array of font variation values. Each element in the array must contain a value that is valid for the corresponding variation axis in the `iAxes` parameter. You can obtain a font's maximum, minimum, and default values for a given variation axis by calling the function `ATSUGetIndFontVariation` (page 81). You can obtain the font variation axis values for a font instance by calling `ATSUGetFontInstance` (page 72).

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

If you supply font variation axes and values to the `ATSUSetVariations` function, you can change the appearance of a style object's font accordingly. You may specify any number of variation axes and values in a style object. Any of the font's variations that you do not set retain their font-defined default values.

You can also use the `ATSUSetVariations` function to supply your own value within any variation axes defined for the font. However, if the font does not support the variation axis you specify, your custom variation has no visual effect.

By calling the function `ATSUGetIndFontVariation` (page 81), you can obtain a variation axis and its maximum, minimum, and default values for a font.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFonts.h`

ATSUStyleIsEmpty

Indicates whether a style object contains only default values.

```
OSStatus ATSUStyleIsEmpty (
   ATSUStyle iStyle,
   Boolean *oIsClear
);
```

Parameters

iStyle

An `ATSUStyle` value specifying the style object to examine.

oIsClear

A pointer to a `Boolean` value. On return, the value is set to `true` if the style object contains only default values for style attributes, font features, and font variations. If `false`, the style object contains one or more nondefault values for style attributes, font features, or font variations.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

You can call the `ATSUStyleIsEmpty` function to determine whether a style object contains only default values for style attributes, font features, and font variations. `ATSUStyleIsEmpty` does not consider reference constants in its evaluation.

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUTextDeleted

Informs ATSUI of the location and length of a text deletion.

```
OSStatus ATSUTextDeleted (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iDeletedRangeStart,
   UniCharCount iDeletedRangeLength
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value specifying the text layout object containing the deleted text.

iDeletedRangeStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the memory location of the deleted text. To specify a deletion point at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify that the entire text buffer has been deleted, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iDeletedRangeLength` parameter.

iIDeletedRangeLength

A `UniCharCount` value specifying the length of the deleted text. To specify a deletion length extending to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion
When you call the `ATSUTextDeleted` function to inform ATSUI of a text deletion, it shortens the style run(s) containing the deleted text by the amount of the deletion. If a style run corresponds entirely to a range of deleted text, that style run is removed. If the deletion point is between two style runs, the first style run is shortened (or removed).

The `ATSUTextDeleted` function also shortens the total length of the text buffer containing the deleted text by the amount of the deletion. That is, it shifts the memory location of the text following the deleted text by `iDeletedRangeLength`. `ATSUTextDeleted` also removes any soft line breaks that fall within the deleted text and updates affected drawing caches.

The `ATSUTextDeleted` function does not change the actual memory location of the affected text. You are responsible for deleting the corresponding text is from the text buffer. You are also responsible for calling the function `ATSUDisposeStyle` (page 49) to dispose of the memory associated with any style runs that have been removed.

Note that calling the function `ATSUTextDeleted` automatically removes previously-set soft line breaks if the line breaks are within the range of text that is deleted.

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

Declared In
`ATSUnicodeObjects.h`

ATSUTextInserted

Informs ATSUI of the location and length of a text insertion.

```
OSStatus ATSUTextInserted (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iInsertionLocation,
   UniCharCount iInsertionLength
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object containing the inserted text.

`iInsertionLocation`
A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the memory location of the inserted text. To specify an insertion point at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`.

`iInsertionLength`
A `UniCharCount` value specifying the length of the inserted text.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
When you call the `ATSUTextInserted` function to inform ATSUI of a text insertion, it extends the style run containing the insertion point by the amount of the inserted text. If the insertion point is between two style runs, the first style run is extended to include the new text.

The `ATSUTextInserted` function also extends the total length of the text buffer containing the inserted text by the amount of the inserted text. That is, it shifts the memory location of the text following the inserted text by `iInsertionLength`. `ATSUTextInserted` then updates drawing caches.

Note that the `ATSUTextInserted` function does not change the actual memory location of the inserted text. You are responsible for placing the inserted text into the text buffer at the appropriate location.

The `ATSUTextInserted` function does not insert style runs or line breaks; to do so, call the functions `ATSUSetRunStyle` (page 127) and `ATSUSetSoftLineBreak` (page 128), respectively. Break line operations should be redone after you call `ATSUTextInserted`.

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

Declared In
`ATSUnicodeObjects.h`

ATSUTextMoved

Informs ATSUI of the new memory location of relocated text.

```
OSStatus ATSUTextMoved (
   ATSUTextLayout iTextLayout,
   ConstUniCharArrayPtr iNewLocation
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value identifying the text layout object associated with the relocated text.

`iNewLocation`
A `ConstUniCharArrayPtr` specifying the new memory location of the moved text.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should call the `ATSUTextMoved` function when a range of text consisting of less than an entire text buffer has been moved. The `ATSUTextMoved` function informs ATSUI of the new memory location of the text. You are responsible for moving the text. The text buffer should remain otherwise unchanged.

When a range of text consisting of an entire text buffer has been moved, you should:

- Call the function `ATSUSetTextPointerLocation` (page 131) to update the text buffer's location.
- Call the function `ATSUSetRunStyle` (page 127) to update the corresponding style runs for the text buffer.
- Call the function `ATSUDrawText` (page 50) to display the updated text.

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

Declared In
`ATSUnicodeObjects.h`

ATSUUnderwriteAttributes

Copies to a destination style object only those nondefault style attribute settings of a source style object that are at default settings in the destination object.

```
OSStatus ATSUUnderwriteAttributes (
   ATSUStyle iSourceStyle,
   ATSUStyle iDestinationStyle
);
```

Parameters

iSourceStyle

An `ATSUStyle` value specifying the style object from which to copy nondefault style attributes.

iDestinationStyle

An `ATSUStyle` value specifying the style object containing style attribute values to be set.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUUnderwriteAttributes` function copies to a destination style object only those nondefault style attribute values of a source style object that are not currently set in a destination style object. Note that the corresponding value in the destination object must not be set in order for a copied value to be applied. All other quantities in the destination style object are left unchanged.

`ATSUUnderwriteAttributes` does not copy the contents of memory referenced by pointers within custom style attributes or within reference constants. You are responsible for ensuring that this memory remains valid until both the source and destination style objects are disposed of.

To create a style object that contains all the contents of another style object, call the function `ATSUCreateAndCopyStyle` (page 38). To copy all the style attributes (including any default settings) of a style object into an existing style object, call the function `ATSUCopyAttributes` (page 32). To copy style attributes that are set in the source whether or not they are set in the destination style object, call the function `ATSUOverwriteAttributes` (page 113).

Availability

Available in Mac OS X v10.0 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

ATSUUnflattenStyleRunsFromStream

Unflattens previously-flattened ATSUI style run data so that it can be read from disk or accepted (through the pasteboard) from another application.

```
OSStatus ATSUUnflattenStyleRunsFromStream (
   ATSUFlattenedDataStreamFormat iStreamFormat,
   ATSUUnFlattenStyleRunOptions iUnflattenOptions,
   ByteCount iStreamBufferSize,
   const void *iStreamBuffer,
   ItemCount iNumberOfRunInfo,
   ItemCount iNumberOfStyleObjects,
   ATSUStyleRunInfo oRunInfoArray[],
   ATSUStyle oStyleArray[],
   ItemCount *oActualNumberOfRunInfo,
   ItemCount *oActualNumberOfStyleObjects
);
```

Parameters

iStreamFormat

The format of the flattened data. There is only one format supported at this time (`'ustl'`) so you must pass the constant `kATSUDataStreamUnicodeStyledText`.

iUnflattenOptions

The options you want to use to unflatten the data. There are no options supported at this time, so you must pass the constant `kATSUUnflattenOptionsNoOptionsMask`.

iStreamBufferSize

The size of the buffer pointed to by the `iStreamBuffer` parameter. You must pass a value greater than `0`.

iStreamBuffer

A pointer to the buffer that contains the flattened data. The data must be of the format specified by the `iStreamFormat` parameter and must be of size specified by the `iStreamBufferSize` parameter. You cannot pass `NULL`.

iNumberOfRunInfo

The number of style run information structures passed in the `iRunInfoArray` parameter. If you are uncertain of the number of style run information structures, see the Discussion.

iNumberOfStyleObjects

The number of `ATSUStyle` objects in the array passed into the `iStyleArray` parameter. If you are uncertain of the number of `ATSUStyle` objects, see the Discussion.

oRunInfoArray[]

On return, points to an array of style run information structures. Each structure contains a style run length and index into the `oStyleArray` array. If you are uncertain of how much memory to allocate for this array, see the Discussion. You are responsible for disposing of the array when you no longer need it.

oStyleArray[]

On return, a pointer to an array of the unique ATSUI style objects (`ATSUStyle`) obtained from the flattened data. The indices returned in the array `oRunInfoArray` are indices into this array. If you are uncertain of how much memory to allocate for this array, see the Discussion. You are responsible for disposing of the array and the ATSUI style objects in the array when you no longer need the array.

oActualNumberOfRunInfo

On return, points to the actual number of `ATSUStyleRunInfo` structures obtained from the flattened data. The actual number of structures is the number of entries added to the array `oRunInfoArray`. You can pass `NULL` if you to not want to obtain this value.

oActualNumberOfStyleObjects

On return, points to the actual number of unique ATSUI style objects (`ATSUStyle`) obtained from the flattened data. The actual number is the number of entries added to the `oStyleArray` array. You can pass `NULL` if you do no want to obtain this value.

Return Value

A result code. See "ATSUI Result Codes" (page 234). This function can also return `paramErr` if you pass invalid values for any of the parameters.

Discussion

The function `ATSUUnflattenStyleRunsFromStream` extracts the ATSUI style run information from previously-flattened data. The style objects and style run information structures are returned in two separate arrays—the array `oStyleArray` and the array `oRunInfoArray`. These arrays are not parallel. Each `ATSUStyle` object in the `oStyleArray` is a unique `ATSUStyle` object. To figure out which `ATSUStyle` object belongs to which text run, the caller must parse the array of `ATSUStyleRunInfo` structures. These structures contain the style run lengths and an index into the `oStyleArray`.

Typically you use the function `ATSUUnflattenStyleRunsFromStream` by calling it twice, as follows:

1. Provide appropriate values for the `iStreamFormat`, `iUnflattenOptions`, and `iStreamBuffer` parameters. Pass `0` for the `iNumberOfRunInfo` and `iNumberOfStyleObjects` parameters, `NULL` for the `oRunInfoArray` and `oStyleArray`, parameters and valid `ItemCount` references for the `oActualNumberOfRunInfo` and `oActualNumberOfStyleObjects` parameters. On return, `oActualNumberOfRunInfo` and `oActualNumberOfStyleObjects` point to the sizes needed to allocate these arrays.

2. Allocate appropriately-sized arrays of `ATSUStyleRunInfo` data structures and `ATSUStyle` objects. Call the function `ATSUUnflattenStyleRunsFromStream` a second time, passing the newly allocated arrays in the `oRunInfoArray` and `oStyleArray` parameters, with the `iNumberOfRunInfo` and `iNumberOfStyleObjects` parameters set to the values you obtained from the first call.

Availability

Available in Mac OS X v10.2 and later.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFlattening.h`

ATSUUnhighlightText

Renders a previously highlighted range of text in an unhighlighted state.

```
OSStatus ATSUUnhighlightText (
   ATSUTextLayout iTextLayout,
   ATSUTextMeasurement iTextBasePointX,
   ATSUTextMeasurement iTextBasePointY,
   UniCharArrayOffset iHighlightStart,
   UniCharCount iHighlightLength
);
```

Parameters

iTextLayout

An `ATSUTextLayout` value identifying the text layout object for which to render unhighlighted text.

iTextBasePointX

An `ATSUTextMeasurement` value specifying the x-coordinate of the origin (in either the current graphics port or in a Quartz graphics context) of the line containing the text range. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to draw relative to the current pen location in the current graphics port.

iTextBasePointY

An `ATSUTextMeasurement` value specifying the y-coordinate of the origin (in either the current graphics port or in a Quartz graphics context) of the line containing the text range. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Convenience Constants" (page 209), to draw relative to the current pen location in the current graphics port.

iHighlightStart

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the text range. If the text range spans multiple lines, you should call `ATSUUnhighlightText` for each line, passing the offset corresponding to the beginning of the new line to draw with each call. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iHighlightLength` parameter.

iHighlightLength

A `UniCharCount` value specifying the length of the text range. To indicate that the text range extends to the end of the text buffer, pass the constant `kATSUToTextEnd`.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

The `ATSUUnhighlightText` function renders a previously highlighted range of text in an unhighlighted state. You should always call `ATSUUnhighlightText` after calling the function `ATSUHighlightText` (page 101), to properly redraw the unhighlighted text and background.

If the inversion method of highlighting was used, when you call `ATSUUnhighlightText`, it merely undoes the inversion and renders the text.

If the redraw method of highlighting was used, `ATSUUnhighlightText` turns off the highlighting and restores the desired background. Depending on the complexity of the background, ATSUI restores the background in one of two ways:

- When the background is a single color (such as white), ATSUI can readily unhighlight the background. In such a case, you specify the background color that ATSUI uses by calling the function `ATSUSetHighlightingMethod` (page 121) and setting `iMethod` to `kRedrawHighlighting` and `iUnhighlightData.dataType` to `kATSUBackgroundColor` and providing the background color in `iUnhighlightData.unhighlightData`. With these settings defined, when you call `ATSUUnhighlightText`, ATSUI simply calculates the previously highlighted area, repaints it with the specified background color, and then redraws the text.
- When the background is more complex (containing, for example, multiple colors, patterns, or pictures), you must provide a redraw background callback function when you call `ATSUSetHighlightingMethod`. You do this by setting `iUnhighlightData.dataType` to `kATSUBackgroundCallback` and providing a `RedrawBackgroundUPP` in `iUnhighlightData.unhighlightData`. When ATSUI calls your callback, you are responsible for redrawing the background of the unhighlighted area. If you choose to also redraw the text, then your callback should return `false` as a function result. If your callback returns `true` ATSUI redraws any text that needs to be redrawn. See `RedrawBackgroundProcPtr` (page 165) for additional information.

Before calculating the dimensions of the area to unhighlight, `ATSUUnhighlightText` examines the text layout object to ensure that each of the characters in the range is assigned to a style run. If there are gaps between style runs, ATSUI assigns the characters in the gap to the style run that precedes (in storage order) the gap. If there is no style run at the beginning of the text range, ATSUI assigns these characters to the first style run it finds. If there is no style run at the end of the text range, ATSUI assigns the remaining characters to the last style run it finds.

The `ATSUUnhighlightText` function uses the previously set line ascent and descent values to calculate the height of the region to unhighlight. If these values have not been set for the line, `ATSUUnhighlightText` uses the line ascent and descent values set for the text layout object containing the line. If these are not set, it uses the default values.

If you want to remove highlighting from a text range that spans multiple lines, you should call `ATSUUnhighlightText` for each line of text that is being unhighlighted, even if all the lines belong to the same text layout object. You should adjust the `iHighlightStart` parameter to reflect the beginning of each line to be unhighlighted.

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

Declared In
`ATSUnicodeDrawing.h`

DisposeATSCubicClosePathUPP

Disposes of a universal procedure pointer (UPP) to a cubic close-path callback.

```
void DisposeATSCubicClosePathUPP (
   ATSCubicClosePathUPP userUPP
);
```

Parameters

`userUPP`
    The universal procedure pointer.

Discussion
See the callback `ATSCubicClosePathProcPtr` (page 156) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

DisposeATSCubicCurveToUPP

Disposes of a universal procedure pointer (UPP) to a cubic curve-to callback.

```
void DisposeATSCubicCurveToUPP (
   ATSCubicCurveToUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion

See the callback `ATSCubicCurveToProcPtr` (page 157) for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeGlyphs.h`

DisposeATSCubicLineToUPP

Disposes of a universal procedure pointer (UPP) to a cubic line-to callback.

```
void DisposeATSCubicLineToUPP (
   ATSCubicLineToUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion

See the callback `ATSCubicLineToProcPtr` (page 158) for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeGlyphs.h`

DisposeATSCubicMoveToUPP

Disposes of a universal procedure pointer (UPP) to a cubic move-to callback.

```
void DisposeATSCubicMoveToUPP (
   ATSCubicMoveToUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion

See the callback `ATSCubicMoveToProcPtr` (page 159) for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In
`ATSUnicodeGlyphs.h`

DisposeATSQuadraticClosePathUPP

Disposes of a universal procedure pointer (UPP) to a quadratic close-path callback.

```
void DisposeATSQuadraticClosePathUPP (
   ATSQuadraticClosePathUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion
See the callback `ATSQuadraticClosePathProcPtr` (page 160) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

DisposeATSQuadraticCurveUPP

Disposes of a universal procedure pointer (UPP) to a quadratic curve callback.

```
void DisposeATSQuadraticCurveUPP (
   ATSQuadraticCurveUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion
See the callback `ATSQuadraticCurveProcPtr` (page 161) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

DisposeATSQuadraticLineUPP

Disposes of a universal procedure pointer (UPP) to a quadratic line callback.

```
void DisposeATSQuadraticLineUPP (
   ATSQuadraticLineUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion

See the callback `ATSQuadraticLineProcPtr` (page 162) for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeGlyphs.h`

DisposeATSQuadraticNewPathUPP

Disposes of a universal procedure pointer (UPP) to a quadratic new-path callback.

```
void DisposeATSQuadraticNewPathUPP (
   ATSQuadraticNewPathUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion

See the callback `ATSQuadraticNewPathProcPtr` (page 163) for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeGlyphs.h`

DisposeATSUDirectLayoutOperationOverrideUPP

Disposes of a universal procedure pointer (UPP) to a layout operation override callback.

```
void DisposeATSUDirectLayoutOperationOverrideUPP (
   ATSUDirectLayoutOperationOverrideUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion

See the callback `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) for more information.

Availability

Available in Mac OS X v10.2 and later.

Declared In
`ATSLayoutTypes.h`

DisposeRedrawBackgroundUPP

Disposes of a new universal procedure pointer (UPP) to a redraw background callback.

```
void DisposeRedrawBackgroundUPP (
   RedrawBackgroundUPP userUPP
);
```

Parameters

`userUPP`
  The universal procedure pointer.

Discussion
See the callback `RedrawBackgroundProcPtr` (page 165) for more information.

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

Declared In
`ATSUnicodeTypes.h`

InvokeATSCubicClosePathUPP

Calls your cubic close-path callback.

```
OSStatus InvokeATSCubicClosePathUPP (
   void *callBackDataPtr,
   ATSCubicClosePathUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSCubicClosePathUPP`, as ATSUI calls your cubic close-path callback for you. See the callback `ATSCubicClosePathProcPtr` (page 156) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSCubicCurveToUPP

Calls your cubic curve-to callback.

```
OSStatus InvokeATSCubicCurveToUPP (
   const Float32Point *pt1,
   const Float32Point *pt2,
   const Float32Point *pt3,
   void *callBackDataPtr,
   ATSCubicCurveToUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSCubicCurveToUPP`, as ATSUI calls your cubic curve-to callback for you. See the callback `ATSCubicCurveToProcPtr` (page 157) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSCubicLineToUPP

Calls your cubic line-to callback.

```
OSStatus InvokeATSCubicLineToUPP (
   const Float32Point *pt,
   void *callBackDataPtr,
   ATSCubicLineToUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSCubicLineToUPP`, as ATSUI calls your cubic line-to callback for you. See the callback `ATSCubicLineToProcPtr` (page 158) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSCubicMoveToUPP

Calls your cubic move-to callback.

```
OSStatus InvokeATSCubicMoveToUPP (
   const Float32Point *pt,
   void *callBackDataPtr,
   ATSCubicMoveToUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSCubicMoveToUPP`, as ATSUI calls your cubic move-to callback for you. See the callback `ATSCubicMoveToProcPtr` (page 159) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSQuadraticClosePathUPP

Calls your quadratic close-path callback.

```
OSStatus InvokeATSQuadraticClosePathUPP (
   void *callBackDataPtr,
   ATSQuadraticClosePathUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSQuadraticClosePathUPP`, as ATSUI calls your quadratic close-path callback for you. See the callback `ATSQuadraticClosePathProcPtr` (page 160) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSQuadraticCurveUPP

Calls your quadratic curve callback.

```
OSStatus InvokeATSQuadraticCurveUPP (
   const Float32Point *pt1,
   const Float32Point *controlPt,
   const Float32Point *pt2,
   void *callBackDataPtr,
   ATSQuadraticCurveUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSQuadraticCurveUPP`, as ATSUI calls your quadratic curve callback for you. See the callback `ATSQuadraticCurveProcPtr` (page 161) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSQuadraticLineUPP

Calls your quadratic line callback.

```
OSStatus InvokeATSQuadraticLineUPP (
   const Float32Point *pt1,
   const Float32Point *pt2,
   void *callBackDataPtr,
   ATSQuadraticLineUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSQuadraticLineUPP`, as ATSUI calls your quadratic line callback for you. See the callback `ATSQuadraticLineProcPtr` (page 162) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSQuadraticNewPathUPP

Calls your quadratic new-path callback.

```
OSStatus InvokeATSQuadraticNewPathUPP (
   void *callBackDataPtr,
   ATSQuadraticNewPathUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSQuadraticNewPathUPP`, as ATSUI calls your quadratic new-path callback for you. See the callback `ATSQuadraticNewPathProcPtr` (page 163) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

InvokeATSUDirectLayoutOperationOverrideUPP

Calls your layout operation override callback.

```
OSStatus InvokeATSUDirectLayoutOperationOverrideUPP (
   ATSULayoutOperationSelector iCurrentOperation,
   ATSULineRef iLineRef,
   URefCon iRefCon,
   void *iOperationCallbackParameterPtr,
   ATSULayoutOperationCallbackStatus *oCallbackStatus,
   ATSUDirectLayoutOperationOverrideUPP userUPP
);
```

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not need to use the function `InvokeATSUDirectLayoutOperationOverrideUPP`, as ATSUI calls your layout operation override callback for your. See the callback `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) for more information.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSLayoutTypes.h`

InvokeRedrawBackgroundUPP

Invokes your redraw background callback.

```
Boolean InvokeRedrawBackgroundUPP (
   ATSUTextLayout iLayout,
   UniCharArrayOffset iTextOffset,
   UniCharCount iTextLength,
   ATSTrapezoid iUnhighlightArea[],
   ItemCount iTrapezoidCount,
   RedrawBackgroundUPP userUPP
);
```

Return Value
A `Boolean` value that indicates whether or not the callback was invoked successfully .

Discussion
You should not need to use the function `InvokeRedrawBackgroundUPP`, as ATSUI calls your redraw background callback for you. See the callback `RedrawBackgroundProcPtr` (page 165) for more information.

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

Declared In
`ATSUnicodeTypes.h`

NewATSCubicClosePathUPP

Creates a new universal procedure pointer (UPP) to a cubic close-path callback.

```
ATSCubicClosePathUPP NewATSCubicClosePathUPP (
   ATSCubicClosePathProcPtr userRoutine
);
```

Parameters

`userRoutine`
A pointer to your cubic close-path callback.

Return Value
On return, a UPP to the cubic close-path callback.

Discussion
See the callback `ATSCubicClosePathProcPtr` (page 156) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSCubicCurveToUPP

Creates a new universal procedure pointer (UPP) to a cubic curve-to callback.

```
ATSCubicCurveToUPP NewATSCubicCurveToUPP (
   ATSCubicCurveToProcPtr userRoutine
);
```

Parameters

userRoutine
A pointer to your cubic curve-to callback.

Return Value
On return, a UPP to the cubic curve-to callback.

Discussion
See the callback `ATSCubicCurveToProcPtr` (page 157) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSCubicLineToUPP

Creates a new universal procedure pointer (UPP) to a cubic line-to callback.

```
ATSCubicLineToUPP NewATSCubicLineToUPP (
   ATSCubicLineToProcPtr userRoutine
);
```

Parameters

userRoutine
A pointer to your cubic line-to callback.

Return Value
On return, a UPP to the cubic line-to callback.

Discussion
See the callback `ATSCubicLineToProcPtr` (page 158) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSCubicMoveToUPP

Creates a new universal procedure pointer (UPP) to a cubic move-to callback.

```
ATSCubicMoveToUPP NewATSCubicMoveToUPP (
   ATSCubicMoveToProcPtr userRoutine
);
```

Parameters

userRoutine
A pointer to your cubic move-to callback.

Return Value
On return, a UPP to the cubic move-to callback.

Discussion
See the callback `ATSCubicMoveToProcPtr` (page 159) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSQuadraticClosePathUPP

Creates a new universal procedure pointer (UPP) to a quadratic close-path callback.

```
ATSQuadraticClosePathUPP NewATSQuadraticClosePathUPP (
   ATSQuadraticClosePathProcPtr userRoutine
);
```

Parameters

userRoutine

A pointer to your quadratic close-path callback.

Return Value
On return, a UPP to the quadratic close-path callback.

Discussion
See the callback `ATSQuadraticClosePathProcPtr` (page 160) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSQuadraticCurveUPP

Creates a new universal procedure pointer (UPP) to a quadratic curve callback.

```
ATSQuadraticCurveUPP NewATSQuadraticCurveUPP (
   ATSQuadraticCurveProcPtr userRoutine
);
```

Parameters

userRoutine

A pointer to your quadratic curve callback.

Return Value
On return, a UPP to the quadratic curve callback.

Discussion
See the callback `ATSQuadraticCurveProcPtr` (page 161) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSQuadraticLineUPP

Creates a new universal procedure pointer (UPP) to a quadratic line callback.

```
ATSQuadraticLineUPP NewATSQuadraticLineUPP (
   ATSQuadraticLineProcPtr userRoutine
);
```

Parameters

`userRoutine`
A pointer to your quadratic line callback.

Return Value
On return, a UPP to the quadratic line callback.

Discussion
See the callback `ATSQuadraticLineProcPtr` (page 162) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSQuadraticNewPathUPP

Creates a new universal procedure pointer (UPP) to a quadratic new-path callback.

```
ATSQuadraticNewPathUPP NewATSQuadraticNewPathUPP (
   ATSQuadraticNewPathProcPtr userRoutine
);
```

Parameters

`userRoutine`
A pointer to your quadratic new-path callback.

Return Value
On return, a UPP to the quadratic new-path callback.

Discussion
See the callback `ATSQuadraticNewPathProcPtr` (page 163) for more information.

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

Declared In
`ATSUnicodeGlyphs.h`

NewATSUDirectLayoutOperationOverrideUPP

Creates a new universal procedure pointer (UPP) to a layout operation override callback.

```
ATSUDirectLayoutOperationOverrideUPP NewATSUDirectLayoutOperationOverrideUPP (
   ATSUDirectLayoutOperationOverrideProcPtr userRoutine
);
```

Parameters

userRoutine

A pointer to your layout operation override callback.

Return Value

On return, a UPP to the layout operation override callback.

Discussion

See the callback `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) for more information.

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSLayoutTypes.h`

NewRedrawBackgroundUPP

Creates a new universal procedure pointer (UPP) to a redraw background callback.

```
RedrawBackgroundUPP NewRedrawBackgroundUPP (
   RedrawBackgroundProcPtr userRoutine
);
```

Parameters

userRoutine

A pointer to your redraw background callback.

Return Value

On return, a UPP to the redraw background callback.

Discussion

See the callback `RedrawBackgroundProcPtr` (page 165) for more information.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeTypes.h`

Callbacks

ATSCubicClosePathProcPtr

Defines a pointer to a cubic close-path callback for drawing glyphs that overrides ATSUI's cubic close-path operation for drawing glyphs.

```
typedef OSStatus(* ATSCubicClosePathProcPtr)
(
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSCubicClosePathCallback (
    void *callBackDataPtr
);
```

Parameters

`callBackDataPtr`

A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetCurvePaths` (page 96). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value

A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetCubicPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion

You supply a pointer to your customized cubic close-path callback as a parameter to the function `ATSUGlyphGetCubicPaths` (page 95).

To provide a pointer to your cubic close-path callback, you create a universal procedure pointer (UPP) of type `ATSCubicClosePathUPP`, using the function `NewATSCubicClosePathUPP` (page 151). You can do so with code similar to the following:

```
ATSCubicClosePathUPP MyCubicClosePathUPP;
MyCubicClosePathUPP = NewATSCubicClosePathUPP (&MyATSCubicClosePathCallback);
```

When you no longer need to use your cubic close-path callback, you should use the function `DisposeATSCubicClosePathUPP` (page 142) to dispose of the universal procedure pointer associated with the callback.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeGlyphs.h`

ATSCubicCurveToProcPtr

Defines a pointer to a cubic curve-to callback for drawing glyphs that overrides ATSUI's cubic curve-to operation for drawing glyphs.

```
typedef OSStatus(* ATSCubicCurveToProcPtr)
(
    const Float32Point *pt1,
    const Float32Point *pt2,
    const Float32Point *pt3,
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSCubicCurveToCallback (
    const Float32Point *pt1,
    const Float32Point *pt2,
    const Float32Point *pt3,
    void *callBackDataPtr
);
```

Parameters

pt1

A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the first off-curve point for this segment of the glyph.

pt2

A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the second off-curve point for this segment of the glyph.

pt3

A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the end of the curve (an on-curve point) for this segment of the glyph.

callBackDataPtr

A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetCurvePaths` (page 96). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value

A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetCubicPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion

You supply a pointer to your customized cubic curve-to function as a parameter to the function `ATSUGlyphGetCubicPaths` (page 95).

To provide a pointer to your cubic curve-to callback, you create a universal procedure pointer (UPP) of type `ATSCubicCurveToUPP`, using the function `NewATSCubicCurveToUPP` (page 151). You can do so with code similar to the following:

```
ATSCubicCurveToUPP MyCubicCurveToUPP;
MyCubicCurveToUPP = NewATSCubicCurveToUPP (&MyATSCubicCurveToCallback);
```

When you no longer need to use your cubic curve-to callback, you should use the function `DisposeATSCubicCurveToUPP` (page 142) to dispose of the universal procedure pointer associated with the callback.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSCubicLineToProcPtr

Defines a pointer to a cubic line-to callback for drawing glyphs that overrides ATSUI's cubic line-to operation for drawing glyphs.

```
typedef OSStatus(* ATSCubicLineToProcPtr)
(
    const Float32Point *pt,
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSCubicLineToCallback (
    const Float32Point *pt,
    void *callBackDataPtr
);
```

Parameters

`pt`
: A `Float32Point` data structure that contains the x and y coordinates for the relative point to which the pen should draw a line.

`callBackDataPtr`
: A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetCurvePaths` (page 96). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value
A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetCubicPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion
You supply a pointer to your customized cubic line-to callback as a parameter to the function `ATSUGlyphGetCubicPaths` (page 95).

To provide a pointer to your cubic line-to callback, you create a universal procedure pointer (UPP) of type `ATSCubicLineToUPP`, using the function `NewATSCubicLineToUPP` (page 152). You can do so with code similar to the following:

```
ATSCubicLineToUPP MyCubicLineToUPP;
MyCubicLineToUPP = NewATSCubicLineToUPP (&MyATSCubicLineToCallback);
```

When you no longer need to use your cubic line-to callback, you should use the function `DisposeATSCubicLineToUPP` (page 143) to dispose of the universal procedure pointer associated with the callback.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSCubicMoveToProcPtr

Defines a pointer to a cubic move-to function for drawing glyphs that overrides ATSUI's cubic move-to operation for drawing glyphs.

```
typedef OSStatus(* ATSCubicMoveToProcPtr)
(
    const Float32Point *pt,
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSCubicMoveToCallback (
    const Float32Point *pt,
    void *callBackDataPtr
);
```

Parameters

`pt`
A `Float32Point` data structure that contains the x and y coordinates for the relative point to which the pen should move before it begins drawing this segment of the glyph.

`callBackDataPtr`
A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetCurvePaths` (page 96). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value
A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetCubicPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion
You supply a pointer to your customized cubic move-to callback as a parameter to the function `ATSUGlyphGetCubicPaths` (page 95).

To provide a pointer to your cubic move-to callback, you create a universal procedure pointer (UPP) of type `ATSCubicMoveToUPP`, using the function `NewATSCubicMoveToUPP` (page 152). You can do so with code similar to the following:

```
ATSCubicMoveToUPP MyCubicMoveToUPP;
MyCubicMoveToUPP = ATSCubicMoveToUPP (&MyATSCubicMoveToCallback);
```

When you no longer need to use your cubic move-to callback, you should use the function `DisposeATSCubicMoveToUPP` (page 143) to dispose of the universal procedure pointer associated with the callback.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticClosePathProcPtr

Defines a pointer to a quadratic close-path callback for drawing glyphs that overrides ATSUI's quadratic close-path operation for drawing glyphs.

```
typedef OSStatus(* ATSQuadraticClosePathProcPtr)
(
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSQuadraticClosePathCallback
(
    void *callBackDataPtr
);
```

Parameters

`callBackDataPtr`
A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetQuadraticPaths` (page 98). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value
A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetQuadraticPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion
You supply a pointer to your customized quadratic close-path callback as a parameter to the function `ATSUGlyphGetQuadraticPaths` (page 98).

To provide a pointer to your quadratic close-path callback, you create a universal procedure pointer (UPP) of type `ATSQuadraticClosePathUPP`, using the function `NewATSQuadraticClosePathUPP` (page 153). You can do so with code similar to the following:

```
ATSQuadraticClosePathUPP MyQuadraticClosePathUPP;
MyQuadraticClosePathUPP = NewATSQuadraticClosePathUPP
(&MyATSQuadraticClosePathCallback);
```

When you no longer need to use your quadratic close-path callback, you should use the function `DisposeATSQuadraticClosePathUPP` (page 144) to dispose of the universal procedure pointer associated with the callback.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticCurveProcPtr

Defines a pointer to a quadratic curve callback for drawing glyphs that overrides ATSUI's quadratic curve operation for drawing glyphs.

```
typedef OSStatus(* ATSQuadraticCurveProcPtr)
(
    const Float32Point *pt1,
    const Float32Point *controlPt,
    const Float32Point *pt2,
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSQuadraticCurveCallback (
    const Float32Point *pt1,
    const Float32Point *controlPt,
    const Float32Point *pt2,
    void *callBackDataPtr
);
```

Parameters

pt1

A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the start of the curve (an on-curve point) for this segment of the glyph.

controlPt

A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the control point (an off-curve point) for this segment of the glyph.

pt2

A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the end of the curve (an on-curve point) for this segment of the glyph.

callBackDataPtr

A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetQuadraticPaths` (page 98). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value
A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetQuadraticPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion
You supply a pointer to your customized quadratic curve callback as a parameter to the function `ATSUGlyphGetQuadraticPaths` (page 98).

To provide a pointer to your quadratic curve callback, you create a universal procedure pointer (UPP) of type `ATSQuadraticCurveUPP`, using the function `NewATSQuadraticCurveUPP` (page 153). You can do so with code similar to the following:

```
ATSQuadraticCurveUPP MyQuadraticCurveUPP;
MyQuadraticCurveUPP = NewATSQuadraticCurveUPP (&MyATSQuadraticCurveCallback);
```

When you no longer need to use your quadratic curve callback, you should use the function `DisposeATSQuadraticCurveUPP` (page 144) to dispose of the universal procedure pointer associated with the callback.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticLineProcPtr

Defines a pointer to a quadratic line callback for drawing glyphs that overrides ATSUI's quadratic line operation for drawing glyphs.

```
typedef OSStatus(* ATSQuadraticLineProcPtr)
(
    const Float32Point *pt1,
    const Float32Point *pt2,
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSQuadraticLineCallback (
    const Float32Point *pt1,
    const Float32Point *pt2,
    void *callBackDataPtr
);
```

Parameters

`pt1`
: A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the start of the line for this segment of the glyph.

`pt2`
: A `Float32Point` data structure that contains the x and y coordinates for the relative point that defines the end of the line for this segment of the glyph.

`callBackDataPtr`
: A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetQuadraticPaths` (page 98). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value
A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetQuadraticPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion
You supply a pointer to your customized quadratic line callback as a parameter to the function `ATSUGlyphGetQuadraticPaths` (page 98).

To provide a pointer to your quadratic line callback, you create a universal procedure pointer (UPP) of type `ATSQuadraticLineUPP`, using the function `NewATSQuadraticLineUPP` (page 154). You can do so with code similar to the following:

```
ATSQuadraticLineUPP MyQuadraticLineUPP;
MyQuadraticLineUPP = NewATSQuadraticLineUPP (&MyATSQuadraticLineCallback);
```

When you no longer need to use your quadratic line callback, you should use the function `DisposeATSQuadraticLineUPP` (page 144) to dispose of the universal procedure pointer associated with the callback.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticNewPathProcPtr

Defines a pointer to a quadratic new-path callback for drawing glyphs that overrides ATSUI's quadratic new-path operation for drawing glyphs.

```
typedef OSStatus(* ATSQuadraticNewPathProcPtr)
(
    void *callBackDataPtr
);
```

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

```
OSStatus MyATSQuadraticNewPathCallback
(
    void *callBackDataPtr
);
```

Parameters

`callBackDataPtr`
A pointer to any data your callback function needs. You pass this pointer to the function `ATSUGlyphGetQuadraticPaths` (page 98). Then, ATSUI passes the pointer through to your callback function when your callback function is invoked.

Return Value
A value that indicates the status of your callback function. When a callback function returns any value other than `0`, the `ATSGlyphGetQuadraticPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

Discussion
You supply a pointer to your customized quadratic new-path callback as a parameter to the function `ATSUGlyphGetQuadraticPaths` (page 98).

To provide a pointer to your quadratic new-path callback, you create a universal procedure pointer (UPP) of type `ATSQuadraticNewPathUPP`, using the function `NewATSQuadraticNewPathUPP` (page 154). You can do so with code similar to the following:

```
ATSQuadraticNewPathUPP MyQuadraticNewPathUPP;
MyQuadraticNewPathUPP = NewATSQuadraticNewPathUPP
(&MyATSQuadraticNewPathCallback);
```

When you no longer need to use your quadratic new-path callback, you should use the function `DisposeATSQuadraticNewPathUPP` (page 145) to dispose of the universal procedure pointer associated with the callback.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSUDirectLayoutOperationOverrideProcPtr

Defines a pointer to a layout operation callback that overrides an ATSUI layout operation.

```
typedef CALLBACK_API_C (OSStatus, ATSUDirectLayoutOperationOverrideProcPtr
)
    ATSULayoutOperationSelector iCurrentOperation,
    ATSULineRef iLineRef,
    UInt32 iRefCon,
    void *iOperationCallbackParameterPtr,
    ATSULayoutOperationCallbackStatus *oCallbackStatus
);
```

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

```
OSStatus MyLayoutOperationOverrideCallback
(
    ATSULayoutOperationSelector iCurrentOperation,
    ATSULineRef iLineRef,
    UInt32 iRefCon,
    void *iOperationCallbackParameterPtr,
    ATSULayoutOperationCallbackStatus *oCallbackStatus
);
```

Parameters

`iCurrentOperation`
: The operation that triggered the callback. This value is passed to your callback by ATSUI. If you write a callback that handles more than one layout operation, you can use this value to determine which operation you should handle.

`iLineRef`
: An `ATSULineRef` value that specifies the line of text on which your callback will operation. Your callback gets called for each line of text associated with the text layout object on which you installed the callback.

iRefCon
: An unsigned 32-bit integer. This is an optional value. You can use this value to specify any data your application needs, such as user preference data.

iOperationCallbackParameterPtr
: A pointer. This is currently unused and should be set to `NULL`.

oCallbackStatus
: A layout callback status value. On output, you must supply a status value to indicate to ATSUI whether or not your callback handled the operation. See "Layout Callback Status Values" (page 221) for a list of the constants you can supply.

Discussion

ATSUI calls your layout operation override function each time the layout operation you specify is invoked. You associate a universal procedure pointer with a text layout object by treating the callback as a layout attribute. That is, you set up a triple (tag, size, value) to specify the layout operation your callback handles, then you call the function `ATSUSetLayoutControls` (page 122) to associate the triple with the text layout object whose layout operation you want to override. The attribute tag you specify is `kATSULayoutOperationOverrideTag`. The attribute value you specify is an `ATSULayoutOperationOverrideSpecifier` structure that contains a selector for a layout operation and a pointer to your callback function.

To provide a pointer to your layout operation override callback, you create a universal procedure pointer (UPP) of type `ATSUDirectLayoutOperationOverrideUPP`, using the function `NewATSUDirectLayoutOperationOverrideUPP` (page 155). You can do so with code similar to the following:

```
ATSUDirectLayoutOperationOverrideUPP MyLayoutOperationOverrideUPP;
MyLayoutOperationOverrideUPP = NewATSUDirectLayoutOperationOverrideUPP
                                (&MyLayoutOperationOverrideCallback);
```

When your layout operation is completed, you should use the function `DisposeATSUDirectLayoutOperationOverrideUPP` (page 145) to dispose of the universal procedure pointer associated with your layout operation override function. However, if you plan to use the same layout operation override function in subsequent layout operations, you can reuse the same UPP, rather than dispose of it and later create a new UPP.

You are limited to the ATSUI functions you can call from within your callback. You can call only those functions that have do not trigger ATSUI to perform the layout operation again. Otherwise, you run the risk of causing infinite recursion. Most functions that use "create", "get", or "copy" semantics are safe to use within your callback. If you call one of the restricted functions, the function returns immediately with the error `kATSUInvalidCallInsideCallbackErr`.

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSLayoutTypes.h`

RedrawBackgroundProcPtr

Defines a pointer to a redraw-background callback that overrides ATSUI's highlighting method for drawing backgrounds.

```
typedef Boolean (* RedrawBackgroundProcPtr)
(
    ATSUTextLayout iLayout,
    UniCharArrayOffset iTextOffset,
    UniCharCount iTextLength,
    ATSTrapezoid *iUnhighlightArea,
    ItemCount iTrapezoidCount
);
```

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

```
Boolean MyRedrawBackgroundCallback (
    ATSUTextLayout iLayout,
    UniCharArrayOffset iTextOffset,
    UniCharCount iTextLength,
    ATSTrapezoid *iUnhighlightArea,
    ItemCount iTrapezoidCount
);
```

Parameters

iLayout
: An `ATSUTextLayout` value that specifies the text layout object on which your callback will operate.

iTextOffset
: The offset of the text to be highlighted.

iTextLength
: The length of the text to be highlighted.

iUnhighlightArea
: An array of `ATSTrapezoid` data structures that describe the boundaries of the highlight area. The boundary values in this array are always specified in QuickDraw coordinates.

iTrapezoidCount
: The number of `ATSTrapezoid` data structures in the `iUnhighlightArea` array.

Return Value

A `Boolean` value that indicates whether ATSUI should redraw the text. If your function redraws the text, your callback should return `false`, otherwise you callback should return `true` to have ATSUI redraw any text that needs to be redrawn.

Discussion

ATSUI calls your customized redraw-background callback when it needs to redraw complex backgrounds (and optionally the text as well). For ATSUI to use your callback, you must first call the `ATSUSetHighlightingMethod` (page 121) function with the `iMethod` parameter set to `kRedrawHighlighting`. You must also pass an `ATSUUnhighlightData` data structure as a parameter to the `ATSUSetHighlightingMethod` function. This structure should contain a pointer to your redraw background callback.

To provide a pointer to your redraw background callback, you create a universal procedure pointer (UPP) of type `RedrawBackgroundUPP`, using the function `NewRedrawBackgroundUPP` (page 155). You can do so with code similar to the following:

```
RedrawBackgroundUPP gMyRedrawBackgroundUPP;
gMyRedrawBackgroundUPP = NewRedrawBackgroundUPP
                             (&MyRedrawBackgroundCallback);
```

For ATSUI to invoke your callback function, you must also pass the `RedrawBackgroundUPP` in the `unhighlightData.backgroundUPP` field of the `iUnhighlightData` parameter for the function `ATSUSetHighlightingMethod`. When finished, you must call the function `DisposeRedrawBackgroundUPP` (page 146) to dispose of the `RedrawBackgroundUPP`.

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

Declared In
`ATSUnicodeTypes.h`

Data Types

Core Data Types

ATSUAttributeInfo

Contains an attribute tag and the size of the attribute.

```
struct ATSUAttributeInfo {
    ATSUAttributeTag fTag;
    ByteCount fValueSize;
};
```

Fields

`fTag`
: Identifies a particular style run or text attribute value. For a description of the Apple-defined style run and text layout attribute tag constants, see "Attribute Tags" (page 196).

`fValueSize`
: The size (in bytes) of the style run or text layout attribute value.

Discussion
Several ATSUI functions pass back an array of structures of this type. The function `ATSUGetAllAttributes` (page 59) passes back an array of `ATSUAttributeInfo` structures to represent the data sizes of all previously set style run attribute values and the corresponding style run attribute tags that identify those style run attribute values. The function `ATSUGetAllLayoutControls` (page 62) passes back an array of `ATSUAttributeInfo` structures to represent the data sizes of all previously set text layout attribute values for an entire text layout object and the corresponding text layout attribute tags that identify those text layout attribute values. The function `ATSUGetAllLineControls` (page 63) passes back an array of `ATSUAttributeInfo` structures to represent the data sizes of all previously set text layout attribute values for a single line in a text layout object and the corresponding text layout attribute tags that identify those text layout attribute values.

ATSLayoutRecord

Contains basic layout information for a single glyph.

```
struct ATSLayoutRecord {
    ATSGlyphRef         glyphID;
    ATSGlyphInfoFlags   flags;
    ByteCount           originalOffset;
    Fixed               realPos;
};
typedef struct ATSLayoutRecord ATSLayoutRecord;
```

Fields

`glyphID`

A reference to a glyph ID.

`flags`

A flag that specifies the glyph's properties. See "Glyph Property Flags" (page 217) for the constants you can use.

`originalOffset`

The byte offset of the character with which this glyph is associated.

`realPos`

A `Fixed` value that specifies the real position of the glyph. This is the x-coordinate of the glyph.

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSLayoutTypes.h`

ATSUStyleSettingRef

A reference to an opaque style setting object.

```
typedef struct LLCStyleInfo*        ATSUStyleSettingRef;
```

Discussion

You can obtain a style setting reference by calling the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) with the selector set to `kATSUDirectDataStyleSettingATSUStyleSettingRefArray`. You can move a style setting reference from one text layout object to another by calling the function `ATSUDirectAddStyleSettingRef` (page 44).

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeDirectAccess.h`

ATSUAttributeValuePtr

Represents a pointer to a style run or text layout attribute value of unknown size.

```
typedef void* ATSUAttributeValuePtr;
```

Discussion
Each attribute value pointed to by `ATSUAttributeValuePtr` is identified by an attribute tag and the size (in bytes) of the attribute value.

You pass the `ATSUAttributeValuePtr` type to functions that set or clear attribute values in style and text layout objects. The `ATSUAttributeValuePtr` type is passed back by functions that query style and text layout objects for their attribute values. You must dereference this pointer and cast it to the appropriate data type to obtain the actual attribute value.

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

Declared In
`ATSUnicodeTypes.h`

ConstATSUAttributeValuePtr

A pointer to a constant attribute value pointer (`ATSUAttributeValuePtr`).

```
typedef const void* ConstATSUAttributeValuePtr;
```

Discussion
An `ATSUAttributeValuePtr` data type provides generic access to storage of attribute values which vary in size.

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

Declared In
`ATSUnicodeTypes.h`

ATSURGBAlphaColor

Contains color information that includes alpha channel information.

```
struct ATSURGBAlphaColor {
    float               red;
    float               green;
    float               blue;
    float               alpha;
};
typedef struct ATSURGBAlphaColor        ATSURGBAlphaColor;
```

Fields

`red`
: A value that specifies the red component of the background color.

`green`
: A value that specifies the green component of the background color.

`blue`
: A value that specifies the blue component of the background color.

`alpha`
: A value that specifies thee alpha channel component of the background color.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeTypes.h`

ATSUBackgroundColor

Redefines the `ATSUBackgroundColor` data type to be an `ATSURGBAlphaColor` data type.

```
typedef ATSURGBAlphaColor ATSUBackgroundColor;
```

Discussion
Prior to Mac OS X version 10.2, the `ATSUBackgroundColor` data type did not include an alpha channel.

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

Declared In
`ATSUnicodeTypes.h`

ATSUBackgroundData

A union that contains a background color or a universal procedure pointer to a callback that redraws the background.

```
union ATSUBackgroundData {
    ATSUBackgroundColor backgroundColor;
    RedrawBackgroundUPP backgroundUPP;
};
```

Fields

`backgroundColor`
: A structure that specifies the background color.

`backgroundUPP`
: A universal procedure pointer to a callback function for redrawing complex backgrounds. See `RedrawBackgroundUPP` (page 196) for more information.

ATSUCaret

Contains the coordinates needed to draw a caret.

```
struct ATSUCaret {
    Fixed fX;
    Fixed fY;
    Fixed fDeltaX;
    Fixed fDeltaY;
};
```

Fields

`fX`
: Represents the x-coordinate of the caret's starting pen position relative to the position of the origin of the line in the current graphics port in which the hit occurred.

`fY`
: Represents the y-coordinate of the caret's starting pen position relative to the position of the origin of the line in the current graphics port in which the hit occurred.

`fDeltaX`
: Represents the x-coordinate of the caret's ending pen position relative to the position of the origin of the line in the current graphics port in which the hit occurred. This position takes into account line rotation. You do not have to rotate it yourself.

`fDeltaY`
: Represents the y-coordinate of the caret's ending pen position relative to the position of the origin of the line in the current graphics port in which the hit occurred. This position takes into account line rotation. You do not have to rotate it yourself.

Discussion

The function `ATSUOffsetToPosition` (page 111) passes back two structures of type `ATSUCaret` to represent the caret position relative to the origin of the line in the current graphics port, corresponding to a specified edge offset. If the edge offset is at a line boundary, the structure passed back in `oMainCaret` contains the starting and ending pen locations of the high caret, while `oSecondCaret` contains the low caret. If the offset is not at a line boundary, both parameters contain the same structure. This structure contains the starting and ending pen locations of the main caret.

You can use the information in this structure to draw a caret by calling the `MoveTo` and `LineTo` functions. For example.

```
MoveTo (fX, fY);
LineTo (fDeltaX, fDeltaY);
```

ATSUFontFeatureType

Represents the attributes of a particular font feature.

```
typedef UInt16 ATSUFontFeatureType;
```

Discussion

Font features are typographic and layout capabilities that you can select or deselect and which control many aspects of glyph selection, ordering, and positioning. Font features include fundamental controls such as whether your text is drawn with contextual forms, as well as details of appearance such as whether you want alternate forms of glyphs to be used at the beginning of a word. To a large extent, how text looks when it is laid out is a function of the number and kinds of font features you choose.

Font vendors create tables that implement the specific set of features which are included in a font by the font designer. Note that only a few feature types and selectors may be available with a given font. If you select features that are not available in a font, you won't see a change in the glyph's appearance. To determine the available features of a font, you can call the functions `ATSUGetFontFeatureTypes` (page 70) and `ATSUGetFontFeatureSelectors` (page 68).

For a complete discussion of font features, the selectors you use to access them, and illustrations of the features, see *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

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

Declared In
`ATSUnicodeTypes.h`

ATSUFontFeatureSelector

Represents the state (on or off) of a particular feature type.

```
typedef UInt16 ATSUFontFeatureSelector;
```

Discussion
You pass the `ATSUFontFeatureSelector` type to functions that set or clear font feature selectors in a style run. The `ATSUFontFeatureSelector` type is passed back by functions that obtain font feature selectors in a style run. For a complete discussion of font feature selectors, see *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

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

Declared In
`ATSUnicodeTypes.h`

ATSUFontVariationAxis

Represents a stylistic attribute and the range of values that the font can use to express this attribute.

```
typedef FourCharCode ATSUFontVariationAxis;
```

Discussion
Font variations allow your application to produce a range of type styles algorithmically. You can obtain a a variation axis and its maximum, minimum, and default values for a font by calling the function `ATSUGetIndFontVariation` (page 81). For a complete discussion of font variations, see *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

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

Declared In
`ATSUnicodeTypes.h`

ATSUFontVariationValue

Represents the range of values that the font can use for a particular font variation.

```
typedef Fixed ATSUFontVariationValue;
```

Discussion
You pass the `ATSUFontVariationValue` type to functions that set and clear font variations in a style run. The `ATSUFontVariationValue` type is passed back by functions that query a style run for font variations.

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

Declared In
`ATSUnicodeTypes.h`

ATSUFontFallbacks

An opaque structure that contains a font fallback list and font fallback cache information.

```
typedef struct OpaqueATSUFontFallbacks *ATSUFontFallbacks;
```

Availability
Available in Mac OS X v10.1 and later.

Declared In
`ATSUnicodeTypes.h`

ATSUFontID

Represents the unique identifier of a font to the font management system in ATSUI.

```
typedef FMFont ATSUFontID;
```

Discussion
You pass the `ATSUFontID` type with functions that set and obtain font information. The `ATSUFontID` type is passed back by functions that count fonts installed on a user's system. The `ATSUFontID` type can be also used to set and get the font in a style run; see "Attribute Tags" (page 196).

An `ATSUFontID` specifies a font family and instance. This value is not guaranteed to remain constant if the system is restarted. You should obtain the font's unique name and store that information in documents for which you need persistent font information.

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

Declared In
`ATSUnicodeTypes.h`

ATSUGlyphInfo

Contains information about a glyph.

```
struct ATSUGlyphInfo {
    GlyphID glyphID;
    UInt16 reserved;
    UInt32 layoutFlags;
    UniCharArrayOffset charIndex;
    ATSUStyle style;
    Float32 deltaY;
    Float32 idealX;
    SInt16 screenX;
    SInt16 caretX;
};
```

Fields

`glyphID`
: A glyph ID. This is unique to the associated font.

`reserved`
: Reserved for Apple's use.

`layoutFlags`
: The layout flags associated with this glyph.

`charIndex`
: The index of the character in the Unicode character stream from which this glyph is derived.

`style`
: An `ATSUStyle` value that specifies the style object associated with this glyph.

`deltaY`
: The cross-stream shift value for this glyph.

`idealX`
: The ideal with-stream offset from the origin of this layout.

`screenX`
: The device-adjusted with-stream offset from the origin of this layout.

`caretX`
: The position in device coordinates where a trailing caret for this glyph intersects the baseline.

Discussion

This data structure is used by ATSUI to return the glyph information associated with one glyph.

ATSUGlyphInfoArray

Contains text layout information for an array of glyphs.

```
struct ATSUGlyphInfoArray {
    ATSUTextLayout layout;
    ItemCount numGlyphs;
    ATSUGlyphInfo glyphs[1];
};
```

Fields

`layout`
: An `ATSUTextLayout` value that specifies the text layout object associated with the glyphs.

`numGlyphs`
: The number of glyphs associated with the text layout object.

`glyphs`
: An array of glyph information structures.

Discussion
This data structure is used by ATSUI to return the glyph information associated with the glyphs in a text layout object.

ATSUGlyphSelector

Contains information that directs ATSUI to use a specific glyph instead of the one ATSUI normally derives.

```
struct ATSUGlyphSelector {
    GlyphCollection     collection;
    GlyphID             glyphID;
};
typedef struct ATSUGlyphSelector        ATSUGlyphSelector;
```

Fields

`collection`
: A value that represents the collection of glyphs you want ATSUI to use. See "Glyph Collection Types " (page 216) for possible values you can supply.

`glyphID`
: A glyph ID value or a collection ID (CID) value. Supply a glyph ID when the collection type is `kGlyphCollectionGID`. Otherwise supply a CID.

Discussion
The `ATSUGlyphSelector` structure along with the attribute tag `kATSUGlyphSelectorTag` allow display of glyphs that do not have an explicit Unicode character. You can use the `kATSUGlyphSelectorTag` to access characters in fonts that otherwise would not be accessible. You can choose the variant glyph by font-specific glyph ID or CID. For more information on CID conventions, see go to http://www.adobe.com. You can get the variant glyph information from an input method through the Text Services Manager using the Carbon event key `kEventParamTextInputGlyphInfoArray`.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeTypes.h`

ATSJustPriorityWidthDeltaOverrides

Contains justification width delta override structures, one for each priority-level override.

```
typedef ATSJustWidthDeltaEntryOverride ATSJustPriorityWidthDeltaOverrides[4];
```

Discussion
For more information see `ATSJustWidthDeltaEntryOverride` (page 176).

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

Declared In
`ATSLayoutTypes.h`

ATSJustWidthDeltaEntryOverride

Contains values that specify the amount of space that can be added to or removed from the right and left sides of each of the glyphs of a given justification priority.

```
struct ATSJustWidthDeltaEntryOverride {
    Fixed beforeGrowLimit;
    Fixed beforeShrinkLimit;
    Fixed afterGrowLimit;
    Fixed afterShrinkLimit;
    JustificationFlags growFlags;
    JustificationFlags shrinkFlags;
};
typedef struct ATSJustWidthDeltaEntryOverride ATSJustWidthDeltaEntryOverride;
```

Fields

`beforeGrowLimit`

The proportion by which a glyph can expand on the left side (top side for vertical text). For example, a value of 0.2 means that a 24-point glyph can have by no more than 4.8 points (0.2 x 24 = 4.8) of extra space added on the left side (top side for vertical text).

`beforeShrinkLimit`

The proportion by which a glyph can shrink on the left side (top side for vertical text). If specified, this value should be negative.

`afterGrowLimit`

The proportion by which a glyph can expand on the right side (bottom side for vertical text).

`afterShrinkLimit`

The proportion by which a glyph can shrink on the right side (bottom side for vertical text). If specified, this value should be negative.

`growFlags`

Mask constants that indicate whether ATSUI should apply the limits defined in the `beforeGrowLimit` and `afterGrowLimit` fields. See "Justification Override Mask Constants" in the Font Manager for a description of possible values. These mask constants also control whether unlimited gap absorption should be applied to the priority of glyphs specified in the given width delta override structure. You can use these mask constants to selectively override the grow case only, while retaining default behavior for other cases.

`shrinkFlags`

Mask constants that indicate whether ATSUI should apply the limits defined in the `beforeShrinkLimit` and `afterShrinkLimit` fields. See "Justification Override Mask Constants" in the Font Manager for a description of possible values. These mask constants also control whether unlimited gap absorption should be applied to the priority of glyphs specified in the given width delta override structure. You can use these mask constants to selectively override the shrink case only, while retaining default behavior for other cases.

Discussion

The `JustWidthDeltaEntryOverride` structure specifies proportions for justification growth and shrinkage, both on the left and the right sides. The growth and shrinkage values override the font-specified widths, such as those specified by the font for kashidas.

It also contains justification flags. The `ATSJustWidthDeltaEntryOverride` data type can be used to set and get justification behavior and priority override weighting; see "Attribute Tags" (page 196).

If you need to access other `'just'` table constants and structures from the `'sfnt'` resource, see the header file SFNTLayoutTypes.h.

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

Declared In
`ATSLayoutTypes.h`

ATSULayoutOperationOverrideSpecifier

Contains an layout operation selector and a pointer to a layout operation override callback.

```
struct ATSULayoutOperationOverrideSpecifier {
    ATSULayoutOperationSelector  operationSelector;
    ATSUDirectLayoutOperationOverrideUPP  overrideUPP;
};
typedef struct ATSULayoutOperationOverrideSpecifier
ATSULayoutOperationOverrideSpecifier;
```

Fields

`operationSelector`
A layout operation selector that specifies the operation for which the callback should be invoked. See "Layout Operation Selectors" (page 221) for the selectors you can specify.

`overrideUPP`
A universal procedure pointer to a layout operation override callback.

Discussion
You can pass this structure as an attribute value for the layout attribute tag `kATSULayoutOperationOverrideTag`.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSLayoutTypes.h`

ATSULineRef

Represents a reference to a structure that specifies a line of text.

```
typedef struct ATSGlyphVector *ATSULineRef;
```

Discussion
You get an ATSUI line reference from ATSUI when your layout operation override callback is invoked. The line reference refers to the line that ATSUI is in the process of laying out.

From within your callback, you pass an ATSUI line reference to the function `ATSUDirectGetLayoutDataArrayPtrFromLineRef` to obtain layout data for that line. The only way you can obtain an ATSUI line reference is from inside your layout operation override callback. An ATSUI line reference is not valid is outside of the callback.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSLayoutTypes.h`

ATSUStyle

Represents a reference to an opaque structure that contains information about a style object.

```
typedef struct OpaqueATSUStyle *ATSUStyle;
```

Discussion
A style object is an opaque structure encapsulating the following character-level style settings

- style attributes: including font ID, font size, font color, kerning control, optical alignment, verticality, and with-stream (left-right) and cross-stream (up-down) shifting (as for superscripts and subscripts)
- font features: including ligatures, swashes, and alternate glyph forms
- font variations: such as continually varying font weight, width, or slant

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

Declared In
`ATSUnicodeTypes.h`

ATSUStyleRunInfo

Contains information for a style run.

```
struct ATSUStyleRunInfo {
    UniCharCount        runLength;
    ItemCount           styleObjectIndex;
};
typedef struct ATSUStyleRunInfo         ATSUStyleRunInfo;
```

Fields

`runLength`
: The length of the style run.

`styleObjectIndex`
: An index into an array of unique style objects.

Discussion
This structure is used by the function `ATSUUnflattenStyleRunsFromStream` (page 138) to return style run information.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeFlattening.h`

ATSUTab

Contains tab settings.

```
struct ATSUTab {
    ATSUTextMeasurement  tabPosition;
    ATSUTabType                          tabType;
};
typedef struct ATSUTab ATSUTab;
```

Fields

`tabPosition`

Specifies a tab position.

`tabType`

Specifies a type of tab stop. See "Tab Positioning Options" (page 232).

Discussion

You can set tabs for a text layout object by calling the function `ATSUSetTabArray` (page 129). You can obtain tab settings by calling the function `ATSUGetTabArray` (page 89).

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeTypes.h`

ATSUTextLayout

Represents a reference to an opaque text layout structure that contains information about a text layout.

```
typedef struct OpaqueATSUTextLayout* ATSUTextLayout;
```

Discussion

The basic building block upon which ATSUI operates is a text layout object (`ATSUTextLayout`). A text layout object ties one or more paragraphs of text together with style attributes that may apply to characters, lines, or the entire layout. The text layout object itself contains information about line and layout attributes, including justification, rotation, direction, and others. Character style information is contained in a style object, which is only associated with, not contained by, a text layout object. For more information on text layout objects, see *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`ATSUnicodeTypes.h`

ATSUTextMeasurement

Represents measurements needed by ATSUI to lay out text, such as outline metrics and line width, ascent, descent.

```
typedef Fixed ATSUTextMeasurement;
```

Discussion

The `ATSUTextMeasurement` type is defined as a `Fixed` value, with a limit of 32K. You must ensure that your measurements are converted to `Fixed` values before passing them to ATSUI functions that use this type.

ATSUI uses fractional `Fixed` values instead of `short` values used in QuickDraw Text. Fractional `Fixed` values provide exact outline metrics and line specifications such as line width, ascent, descent, and so on.

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

Declared In
`ATSUnicodeTypes.h`

ATSTrapezoid

Contains the coordinates of the typographic bounding trapezoid for the final layout of a line a text.

```
struct ATSTrapezoid {
    FixedPoint upperLeft;
    FixedPoint upperRight;
    FixedPoint lowerRight;
    FixedPoint lowerLeft;
};
```

Fields

`upperLeft`
: A structure of type `FixedPoint` that contains the upper left coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

`upperRight`
: A structure of type `FixedPoint` that contains the upper right coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

`lowerRight`
: A structure of type `FixedPoint` that identifies the lower right coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

`lowerLeft`
: A structure of type `FixedPoint` that identifies the lower left coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

Discussion
The dimensions of the resulting trapezoid are relative to the coordinates specified in the `iTextBasePointX` and `iTextBasePointY` parameters. The width of the glyph bounds is determined based on the value passed in the `iTypeOfBounds` parameter.

The function `ATSUGetGlyphBounds` (page 75) passes back an array of structures of type `ATSTrapezoid` to specify the enclosing trapezoid(s) of a final laid-out line of text. If the range of text spans directional boundaries, `ATSUGetGlyphBounds` produces multiple trapezoids defining these regions.

Version Notes
In ATSUI 1.1, the function `ATSUGetGlyphBounds` can pass back a maximum of 31 bounding trapezoids. In ATSUI 1.2, `ATSUGetGlyphBounds` can pass back as many as 127 bounding trapezoids.

ATSUUnhighlightData

Contains data needed to redraw the background.

```
struct ATSUUnhighlightData {
    ATSUBackgroundDataType dataType;
    ATSUBackgroundData unhighlightData;
};
```

Fields

`dataType`

The data type of the background—a color or a callback.

`unhighlightData`

A background color or a universal procedure pointer to a callback that redraws the background.

USTL Data Structure Data Types

The data types in this section define the `'ustl'` data structure, which is the data structure used by ATSUI to contain flattened data. The `'ustl'` data structure has four blocks. The Block 1 structure defines is a header for the entire `'ustl'` data structure. Block 2 structures define flattened text layout data. (Note that Block 2 structures are not currently used by the functions `ATSUFlattenStyleRunsToStream` (page 55) and `ATSUUnflattenStyleRunsFromStream` (page 138).) Block 3 structures define flattened style run data. Block 4 structures define flattened style data.

The `'ustl'` data structure can accommodate any ATSUI text layout and style run data associated with a document. That is, the `'ustl'` data structure can contain data for multiple text layout objects, multiple style runs, and multiple style objects. Within each block (text layout, style run, and style) you must specify the number structures in that block.

ATSFlatDataMainHeaderBlock

Contains the `'ustl'` data structure version and size and provides offsets to the text layout, style run, and style list data blocks.

```
struct ATSFlatDataMainHeaderBlock {
    UInt32              version;
    ByteCount           sizeOfDataBlock;
    ByteCount           offsetToTextLayouts;
    ByteCount           offsetToStyleRuns;
    ByteCount           offsetToStyleList;
};
typedef struct ATSFlatDataMainHeaderBlock ATSFlatDataMainHeaderBlock;
```

Fields

`version`

The version number of the `'ustl'` data structure. You must make sure this number is the first item in the data block, otherwise the data may not be readable by code written to parse earlier versions of `'ustl'` data.

`sizeOfDataBlock`

The total size of the data in bytes, including the four bytes needed for the version number.

`offsetToTextlayouts`

The offset from the beginning of the data block to the flattened text layout data. You can set this value to `0` if there is no text layout data. This value specifies the offset to the `ATSFlatDataTextLayoutDataHeader` (page 182) structure.

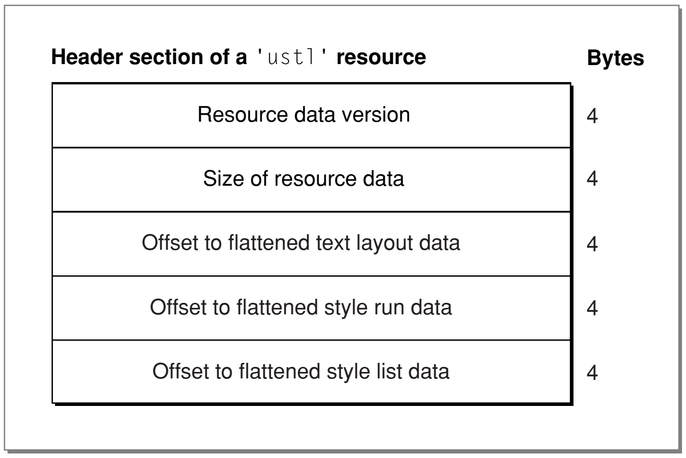
`offsetToStyleRuns`
: The offset from the beginning of the data to the flattened style run data. You can set this value to `0` if there is no flattened style run data. This value specifies the offset to the `ATSFlatDataStyleRunDataHeader` (page 186) structure.

`offsetToStyleList`
: The offset to the flattened style list data. You can set this value to `0` if there is no flattened style list data. This value specifies the offset to the `ATSFlatDataStyleListHeader` (page 187) structure.

Discussion

The structure `ATSFlatDataMainHeaderBlock` is Block 1 of the `'ustl'` data structure. This structure contains information about the rest of the `'ustl'` data structure and provides offsets to each of the other three data blocks. Figure 1 illustrates the main header structure.

Figure 1 The main header for the ustl data structure


Header section of a 'ustl' resource
Bytes
Resource data version
4
Size of resource data
4
Offset to flattened text layout data
4
Offset to flattened style run data
4
Offset to flattened style list data
4

Per the `'ustl'` specification, all data blocks with the `'ustl'` data structure must maintain 4-byte alignment. For such items as font names, which have a variable width, you must add padding bytes to ensure the 4-byte alignment is always maintained.

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeFlattening.h`

ATSFlatDataTextLayoutDataHeader

Contains size, length, and offset information for a text layout data block.

```
struct ATSFlatDataTextLayoutDataHeader {
        ByteCount           sizeOfLayoutData;
        ByteCount           textLayoutLength;
        ByteCount           offsetToLayoutControls;
        ByteCount           offsetToLineInfo;
};
typedef struct ATSFlatDataTextLayoutDataHeader ATSFlatDataTextLayoutDataHeader;
```

Fields

`sizeOfLayoutData`

The size of the flattened text layout data. This value must include any bytes that have been added to maintain the required 4-byte alignment.

`textLayoutLength`

The number of characters to which the flattened text layout data applies.

`offsetLayoutControls`

The offset to the flattened layout control data. This offset is relative to the start of the text layout data block, and specifies the offset to the `ATSFlatDataLayoutControlsDataHeader` (page 185) structure. The offset can be set to zero if there are no layout controls.

`offsetToLineLength`

The offset to the flattened line info data. This offset is relative to the start of the text layout data block, and specifies the offset to the `ATSFlatDataLineInfoHeader` (page 185) structure. The offset can be set to zero if there is no line info in this layout.

Discussion

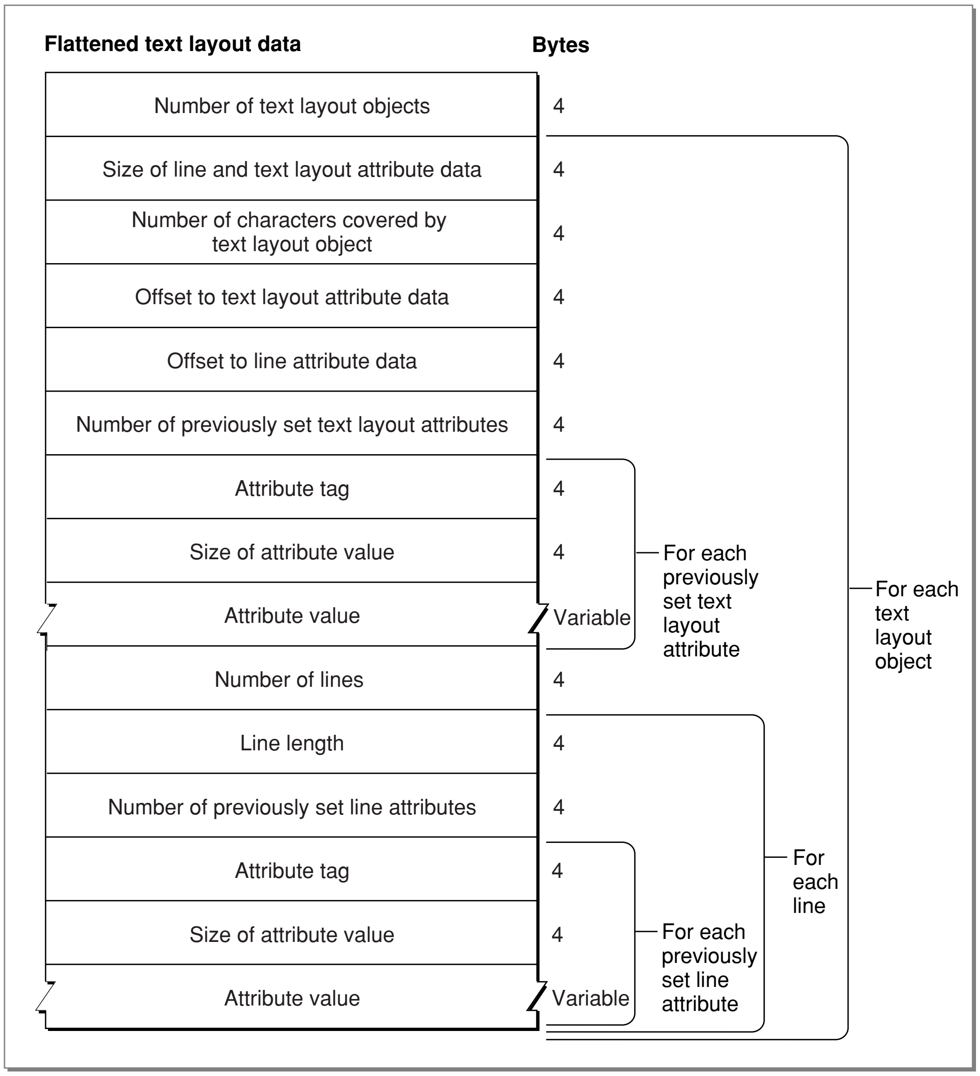
The `ATSFlatDataTextLayoutDataHeader` structure is a block 2 data structure and it is the main header for text layout data. If you have text layout data to flatten or unflatten, you must have one of these structures. for each text layout object whose data you want to flatten.

Note that the `ATSFlatDataTextLayoutDataHeader` data structure(s) must be preceded by an `ItemCount` value that specifies the number of `ATSFlatDataTextLayoutDataHeader` data structures included in the flattened data. Although the `ItemCount` value is not part of any `'ustl'` data structure, you need to include this 4-byte value when you flatten your text layout data so that you can successfully parse the flattened data at a later time.

The `offsetToTextLayouts` field in the `ATSFlatDataMainHeaderBlock` (page 181) structure specifies the offset to the structure `ATSFlatDataTextLayoutDataHeader`.

Figure 2 depicts the flattened text layout data. At the top of the figure is the information contained in the data header (`ATSFlatDataTextLayoutDataHeader`). Following the header are layout controls data (see `ATSFlatDataLayoutControlsDataHeader` (page 185)) and line length data (see `ATSFlatDataLineInfoHeader` (page 185) and `ATSFlatDataLineInfoData` (page 186)).

Figure 2 Flattened text layout data


Flattened text layout data
Bytes
Number of text layout objects
4
Size of line and text layout attribute data
4
Number of characters covered by text layout object
4
Offset to text layout attribute data
4
Offset to line attribute data
4
Number of previously set text layout attributes
4
Attribute tag
4
Size of attribute value
4
Attribute value
Variable
For each previously set text layout attribute
For each text layout object
Number of lines
4
Line length
4
Number of previously set line attributes
4
Attribute tag
4
Size of attribute value
4
Attribute value
Variable
For each previously set line attribute
For each line

If the `offsetToLayoutControls` value is not zero, there must be a `ATSFlatDataLayoutControlsDataHeader` (page 185) structure that contains a count of the number of layout controls and an array of layout control attribute data.

If the `offsetToLineInfo` is not zero, then following the flattened layout controls data you must have an `ATSFlatDataLineInfoHeader` (page 185) structure.

This and other Block 2 structures are not currently used by the functions `ATSUFlattenStyleRunsToStream` (page 55) and `ATSUUnflattenStyleRunsFromStream` (page 138).

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeFlattening.h`

ATSFlatDataLayoutControlsDataHeader

Contains the number of flattened layout controls and an array of layout control attribute data.

```
struct ATSFlatDataLayoutControlsDataHeader {
        ItemCount           numberOfLayoutControls;
        ATSUAttributeInfo   controlArray[1];
};typedef struct ATSFlatDataLayoutControlsDataHeader
ATSFlatDataLayoutControlsDataHeader;
```

Fields

`numberOfLayoutControls`

The number of flattened layout controls. There should be at least one layout control that specifies the line direction of the layout.

`controlArray[1]`

The first entry in an array of ATSUI attribute information. There should be `numberOfLayoutControls` entries in this array. If necessary, each ATSUI attribute info structure in the array should be followed by padding bytes to maintain the required 4-byte alignment. The value in the `fValueSize` field of each `ATSUAttributeInfo` structure must specify the size of the attribute value, and must not reflect any padding bytes you added.

Discussion

The `ATSFlatDataLayoutControlsDataHeader` structure is the header for the flattened layout controls structure. The `offsetToLayoutControls` field in the `ATSFlatDataTextLayoutDataHeader` (page 182) structure specifies the offset to the structure `ATSFlatDataLayoutControlsDataHeader`. If there are no layout controls, you do not need the `ATSFlatDataLayoutControlsDataHeader` structure.

This and other Block 2 structures are not currently used by the functions `ATSUFlattenStyleRunsToStream` (page 55) and `ATSUUnflattenStyleRunsFromStream` (page 138).

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeFlattening.h`

ATSFlatDataLineInfoHeader

Contains the number of lines and an array of line information data.

```
struct ATSFlatDataLineInfoHeader {
        ItemCount           numberOfLines;
        ATSFlatDataLineInfoData  lineInfoArray[1];
};
typedef struct ATSFlatDataLineInfoHeader ATSFlatDataLineInfoHeader;
```

Fields

`numberOfLines`

The number of flattened line info structures that are stored in this block. This value should be greater than zero and equal to the number of soft line breaks in the layout plus one.

`lineInfoArray[1]`

The first entry in a array of `ATSFlatDataLineInfoData` (page 186) structures. There should be `numberOfLines` entries in this array.

Discussion
The ATSFlatDataLineInfoHeader structure is the main data header for the flattened line info data. The value offsetToLineInfo in the ATSFlatDataTextLayoutDataHeader (page 182) specifies the offset to the ATSFlatDataLineInfoHeader structure.

This and other Block 2 structures are not currently used by the functions ATSUFlattenStyleRunsToStream (page 55) and ATSUUnflattenStyleRunsFromStream (page 138).

Availability
Available in Mac OS X v10.2 and later.

Declared In
ATSUnicodeFlattening.h

ATSFlatDataLineInfoData

Contains a line length and the number of line controls for a line of flattened text.

```
struct ATSFlatDataLineInfoData {
        UniCharCount        lineLength;
        ItemCount           numberOfLineControls;
};
typedef struct ATSFlatDataLineInfoData  ATSFlatDataLineInfoData;
```

Fields

lineLength
: The number of UniChars characters in the line.

numberOfLineControls
: The number of line controls applied to the line. You can set this value to zero if there are no line controls applied to this line.

Discussion
If the numberOfLineControls is not zero, then you must supply an array of ATSUAttributeInfo structures that contains numberOfLineControls elements.

This and other Block 2 structures are not currently used by the functions ATSUFlattenStyleRunsToStream (page 55) and ATSUUnflattenStyleRunsFromStream (page 138).

Availability
Available in Mac OS X v10.2 and later.

Declared In
ATSUnicodeFlattening.h

ATSFlatDataStyleRunDataHeader

Contains the number of style runs and style run information for the style run data block.

```
struct ATSFlatDataStyleRunDataHeader {
    ItemCount           numberOfStyleRuns;
    ATSUStyleRunInfo    styleRunArray[1];
};
typedef struct ATSFlatDataStyleRunDataHeader ATSFlatDataStyleRunDataHeader;
```

Fields

`numberOfStyleRuns`

The number of style run data structures stored in this block.

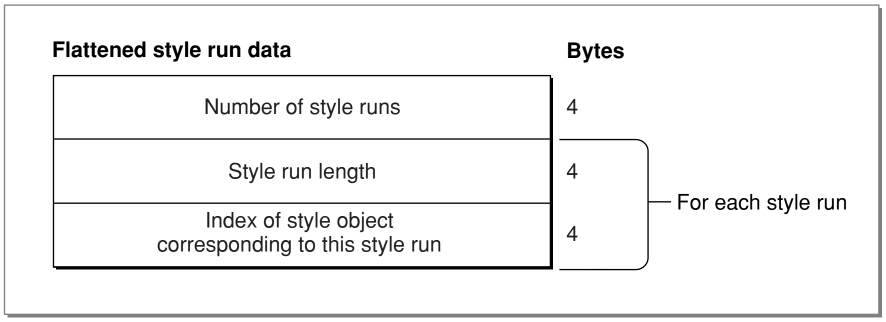
`styleRunArray[1]`

The first entry in a array of `ATSUStyleRunInfo` structures. There should be `numberOfStyleRuns` entries in this array.

Discussion

The `ATSFlatDataStyleRunDataHeader` structure precedes style run data structures. The `offsetToStyleRuns` field in the `ATSFlatDataMainHeaderBlock` (page 181) specifies the offset to the structure `ATSFlatDataStyleRunDataHeader`.

Figure 3 Flattened style run data



This is a Block 3 structure. Block 3 structures are used by ATSUI style run flattening and parsing functions, `ATSUFlattenStyleRunsToStream` (page 55) and `ATSUUnflattenStyleRunsFromStream` (page 138), to represent flattened style run information. These structures work together with Block 4 structures.

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeFlattening.h`

ATSFlatDataStyleListHeader

Contains the number of styles and the first item in the style list style data header.

```
struct ATSFlatDataStyleListHeader {
        ItemCount           numberOfStyles;
        ATSFlatDataStyleListStyleDataHeader  styleDataArray[1];
};
typedef struct ATSFlatDataStyleListHeader ATSFlatDataStyleListHeader;
```

Fields

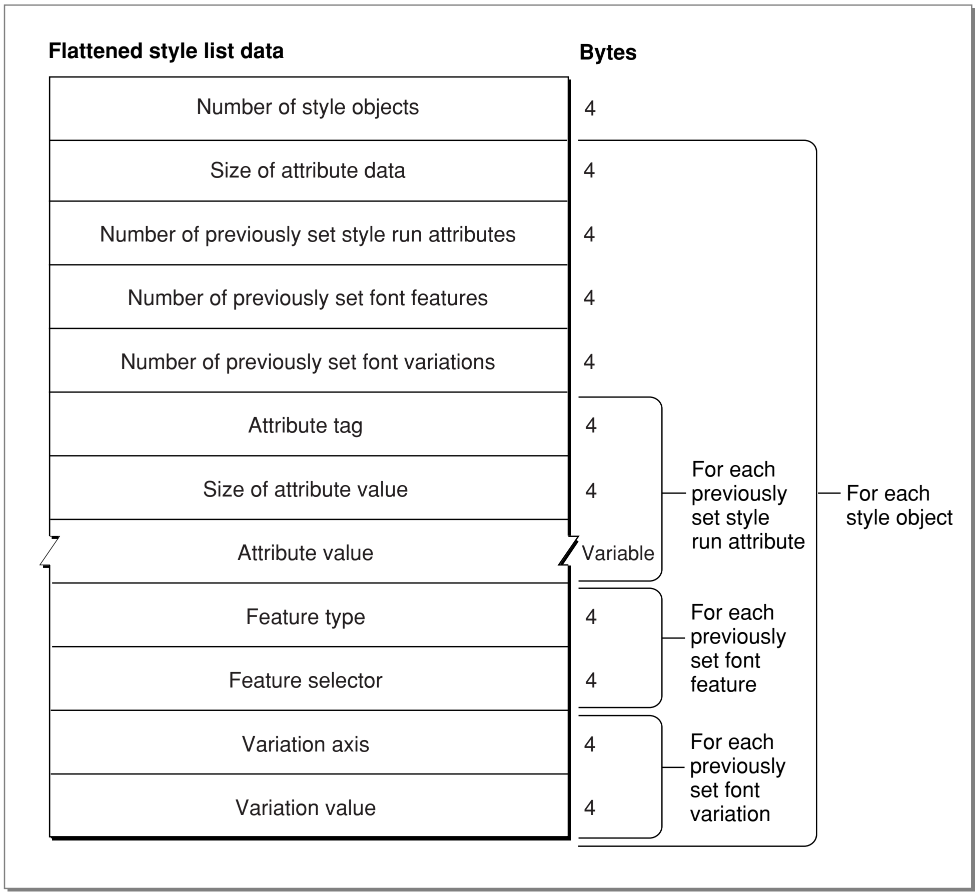
`numberOfStyles`
The number of flattened style objects in this block.

`styleDataArray[1]`
The first item in an array of `ATSFlatDataStyleListStyleDataHeader` (page 189) structures. There should be `numberOfStyles` entries in this array. Note that the data stored in these structures can be of variable sizes.

Discussion

The `ATSFlatDataStyleListHeader` structure is the main header for Block 4. The `offsetToStyleList` field in the `ATSFlatDataMainHeaderBlock` (page 181) specifies the offset to the structure `ATSFlatDataStyleListHeader`.

Figure 4 Flattened style list data


Flattened style list data
Bytes
Number of style objects
4
Size of attribute data
4
Number of previously set style run attributes
4
Number of previously set font features
4
Number of previously set font variations
4
Attribute tag
4
Size of attribute value
4
Attribute value
Variable
Feature type
4
Feature selector
4
Variation axis
4
Variation value
4
For each previously set style run attribute
For each previously set font feature
For each previously set font variation
For each style object

Availability

Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeFlattening.h`

ATSFlatDataStyleListStyleDataHeader

Contains size information and the number of attributes, features, and variations for the style list data block.

```
struct ATSFlatDataStyleListStyleDataHeader {
            ByteCount           sizeOfStyleInfo;
            ItemCount           numberOfSetAttributes;
            ItemCount           numberOfSetFeatures;
            ItemCount           numberOfSetVariations;
};
typedef struct ATSFlatDataStyleListStyleDataHeader
ATSFlatDataStyleListStyleDataHeader;
```

Fields

`sizeOfStyleInfo`
The size of the flattened style object. This value should include the four bytes for this field (`sizeOfStyleInfo`) and any padding bytes you add to end of the structure to maintain the required 4-byte alignment.

`numberOfSetAttributes`
The number of attributes in the flattened style object. You should have at least one attribute for the font data, although you can set this value to `0` if you do not want to specify font data.

`numberOfSetFeatures`
The number of font features in the flattened style object. You can set this value to `0` if there are no font features in the style object.

`numberOfSetVariations`
The number of font variations in the flattened style object. You can set this value to `0` if there are no font variations in the style object.

Discussion
The `ATSFlatDataStyleListStyleDataHeader` structure forms the beginning of an individually flattened `ATSUStyle` object. This structure precedes the following data:

1. If the value `numberOfSetAttributes` is non-zero, there must be an array of `ATSUAttributeInfo` structures immediately following the `ATSFlatDataStyleListStyleDataHeader` structure to store the style attributes. This is a variable-size array. The number of `ATSUAttributeInfo` structures must be equal to the value `numberOfSetAttributes`, one structure for each attribute.

   If the value `numberOfSetAttributes` is zero, you do not need an array of `ATSUAttributeInfo` structures.

2. If the value `numberOfSetFeatures` is non-zero, there must be an array of `ATSFlatDataStyleListFeatureData` structures. These structures must appear immediately following the `ATSUAttributeInfo` array above (if there is such an array). The number of `ATSFlatDataStyleListFeatureData` structures must be equal to the value `numberOfSetFeatures`, one structure for each feature.

   If the value `numberOfSetFeatures` is zero, you do not need an array of `ATSFlatDataStyleListFeatureData` structures.

3. If the value `numberOfSetVariations` is non-zero, there must be an array of `ATSFlatDataStyleListVariationData` structures immediately following the `ATSFlatDataStyleListFeatureData` array (if there is such an array). The number of `ATSFlatDataStyleListVariationData` structures must be equal to the value `numberOfSetVariations`, one structure for each variation.

This is a Block 4 structure. Block 4 structures store flattened `ATSUStyle` objects and are currently used by the ATSUI style run flattening and parsing functions, `ATSUFlattenStyleRunsToStream` (page 55) and `ATSUUnflattenStyleRunsFromStream` (page 138).

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeFlattening.h`

ATSFlatDataStyleListFeatureData

Contains flattened font feature data.

```
struct ATSFlatDataStyleListFeatureData {
            ATSUFontFeatureType  theFeatureType;
            ATSUFontFeatureSelector  theFeatureSelector;
};
typedef struct ATSFlatDataStyleListFeatureData ATSFlatDataStyleListFeatureData;
```

Fields

`theFeatureType`
    A font feature type.

`theFeatureSelector`
    A font feature selector.

Discussion
This is a Block 4 structure. The structure `ATSFlatDataStyleListFeatureData` stores flattened font feature data. If the value `numberOfSetFeatures` in the `ATSFlatDataStyleListStyleDataHeader` (page 189) structure is non-zero, an array of these structure must follow the array of font data attributes (if such an array exists) if the `numberOfSetFeatures` is non-zero. The number of `ATSFlatDataStyleListFeatureData` structures must be equal to the value `numberOfSetFeatures`.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeFlattening.h`

ATSFlatDataStyleListVariationData

Contains flattened font variation axis data.

```
struct ATSFlatDataStyleListVariationData {
            ATSUFontVariationAxis  theVariationAxis;
            ATSUFontVariationValue  theVariationValue;
};
typedef struct ATSFlatDataStyleListVariationData ATSFlatDataStyleListVariationData;
```

Fields

`theVariationAxis`

A font variation axis.

`theVariationValue`

A font variation value.

Discussion

This is a Block 4 structure. The structure `ATSFlatDataStyleListVariationData` stores flattened font variation data. If the value `numberOfSetVariations` in the `ATSFlatDataStyleListStyleDataHeader` (page 189) structure is non-zero, an array of these structure must follow the array of font features (if such an array exists) if the `numberOfSetVariations` is non-zero. The number of `ATSFlatDataStyleListVariationData` structures must be equal to the value `numberOfSetVariations`.

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeFlattening.h`

ATSFlatDataFontNameDataHeader

Contains font name information.

```
struct ATSFlatDataFontNameDataHeader {
        ATSFlatDataFontSpeciferType  nameSpecifierType;
        ByteCount           nameSpecifierSize;
};
typedef struct ATSFlatDataFontNameDataHeader ATSFlatDataFontNameDataHeader;
```

Fields

`nameSpecifierType`

A font specifier for the type of the font name data you plan to supply. See "Flattened Data Font Type Selectors" (page 212) for a list of the font specifiers you can supply. The font name data must follow the `ATSFlatDataFontNameDataHeader` structure.

`nameSpecifierSize`

The size of the flattened font name data. This value must not include any padding bytes that may be necessary to achieve the required 4-byte alignment, unless the padding bytes are specified as part of structure, such as with the `ATSFlatDataFontSpecRawNameData` structure.

Discussion

Font information can be recorded in an `ATSUStyle` object using the attribute tag `kATSUFontTag` and an attribute value that is of type `ATSUFontID`. Unfortunately, a font ID can vary between systems or system startups, which means you cannot ensure that the font used when the style is flattened is the same font that will be used with the style is unflattened. To preserve font information, you must flatten font name data. You specify font information using the structure `ATSFlatDataFontNameDataHeader`. You store this structure as a style attribute value. You must make sure this structure maintains the required 4-byte alignment.

Following the `ATSFlatDataFontNameDataHeader` structure must be the flattened font name data of the type specified by the `nameSpecifierType` field. For instance, if the value of the `nameSpecType` field is `kATSFlattenedFontNameSpecifierRawNameData`, the structure that immediately follows should be a `ATSFlatDataFontSpecRawNameDataHeader` (page 192) structure.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeFlattening.h`

ATSFlatDataFontSpecRawNameDataHeader

Contains raw font name data.

```
struct ATSFlatDataFontSpecRawNameDataHeader {
            ItemCount           numberOfFlattenedNames;
            ATSFlatDataFontSpecRawNameData  nameDataArray[1];
};
typedef struct ATSFlatDataFontSpecRawNameDataHeader
ATSFlatDataFontSpecRawNameDataHeader;
```

Fields

`numberOfFlattenedNames`
: The number of flattened font names. There must be at least one flattened font name, otherwise the structure is malformed.

`nameDataArray[1]`
: The first element in an array of raw font name data.

Discussion
The function `ATSUUnflattenStyleRunsFromStream` (page 138) searches for fonts that match the font data provides in the `nameDataArray`. ATSUI obtains matches for all the font name specifiers in the structure. You must supply at least one entry in the `nameDataArray`, but you may want to supply more than one entry to ensure a specific match. For example, the default ATSUI implementation is to use two name specifiers—the full name of the font (`kFontFullName`) and the font manufacturer's name (`kFontManufacturerName`).

The `ATSFlatDataFontSpecRawNameDataHeader` structure must be followed by one or more `ATSFlatDataFontSpecRawNameData` structures. The number of structures must match the value specified by the `numberOfFlattenedName` field.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSUnicodeFlattening.h`

ATSFlatDataFontSpecRawNameData

Contains data for a font name.

```
struct ATSFlatDataFontSpecRawNameData {
            FontNameCode        fontNameType;
            FontPlatformCode    fontNamePlatform;
            FontScriptCode      fontNameScript;
            FontLanguageCode    fontNameLanguage;
            ByteCount           fontNameLength;
};
typedef struct ATSFlatDataFontSpecRawNameData ATSFlatDataFontSpecRawNameData;
```

Fields

`fontNameType`
: The type of font name. You must supply this parameter.

`fontNamePlatform`
: The platform type of the font name. You should specify this if you know it (Unicode, Mac, and so forth). If you do not know the platform type, then specify `kFontNoPlatform`. In this case all matching is done by ATSUI based on the first font in the name table that matches the other parameters in this structure.

`fontNameScript`
: The script code of the font name based on the platform specified in the `fontNamePlatform` field. If you set this to `kFontNoScript`, the name is matched based on the first font in the name table that matches the other font name parameters in this structure.

`fontNameLanguage`
: The language of the font name. If you set this to `kFontNoLanguage`, the name is matched based on the first font in the name table that matches the other font name parameters in this structure.

`fontNameLength`
: The length of the font name. The length should include any padding bytes needed to maintain the required 4-byte alignment.

Discussion

The `ATSFlatDataFontSpecRawNameData` structure is the structure in which raw font name data is actually stored. This structure is used only when the value of the `nameSpecifierType` field in the `ATSFlatDataFontNameDataHeader` (page 191) structure is `kATSFlattenedFontSpecifierRawNameData`. The structure stores multiple font name table entries for the purposes of reconstructing an `ATSUFontID` value for the same font at some time in the future.

When the ATSUI parsing function `ATSUUnflattenStyleRunsFromStream` searches for fonts to match the font data in this structure, it obtains matches for all the font name specifiers in the structure. The default ATSUI implementation is to use two name specifiers—the full name of the font (`kFontFullName`) and the font manufacturer's name (`kFontManufacturerName`).

Availability

Available in Mac OS X v10.2 and later.

Declared In

`ATSUnicodeFlattening.h`

Universal Procedure Pointers

ATSUDirectLayoutOperationOverrideUPP

Defines a universal procedure pointer to a layout operation callback.

```
typedef ATSUDirectLayoutOperationOverrideProcPtr
ATSUDirectLayoutOperationOverrideUPP;
```

Discussion
For more information, see the description of the `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) callback function.

Availability
Available in Mac OS X v10.2 and later.

Declared In
`ATSLayoutTypes.h`

ATSCubicClosePathUPP

Defines a universal procedure pointer to a cubic close-path callback.

```
typedef ATSCubicClosePathProcPtr ATSCubicClosePathUPP;
```

Discussion
For more information, see the description of the `ATSCubicClosePathProcPtr` (page 156) callback function.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSCubicCurveToUPP

Defines a universal procedure pointer to a cubic curve-to callback.

```
typedef ATSCubicCurveToProcPtr ATSCubicCurveToUPP;
```

Discussion
For more information, see the description of the `ATSCubicCurveToProcPtr` (page 157) callback function.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSCubicLineToUPP

Defines a universal procedure pointer to a cubic line-to callback.

```
typedef ATSCubicLineToProcPtr ATSCubicLineToProcUPP;
```

Discussion
For more information, see the description of the `ATSCubicLineToProcPtr` (page 158) callback function.

ATSCubicMoveToUPP

Defines a universal procedure pointer to a cubic move-to callback.

```
typedef ATSCubicMoveToProcPtr ATSCubicMoveToUPP;
```

Discussion
For more information, see the description of the `ATSCubicMoveToProcPtr` (page 159) callback function.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticClosePathUPP

Defines a universal procedure pointer to a quadratic close-path callback.

```
typedef ATSQuadraticClosePathProcPtr ATSQuadraticClosePathUPP;
```

Discussion
For more information, see the description of the `ATSQuadraticClosePathProcPtr` (page 160) callback function.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticCurveUPP

Defines a universal procedure pointer to a quadratic curve callback.

```
typedef ATSQuadraticCurveProcPtr ATSQuadraticCurveUPP;
```

Discussion
For more information, see the description of the `ATSQuadraticCurveProcPtr` (page 161) callback function.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticLineUPP

Defines a universal procedure pointer to a quadratic line callback.

```
typedef ATSQuadraticLineProcPtr ATSQuadraticLineUPP;
```

Discussion
For more information, see the description of the `ATSQuadraticLineProcPtr` (page 162) callback function.

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

Declared In
`ATSUnicodeGlyphs.h`

ATSQuadraticNewPathUPP

Defines a universal procedure pointer to a quadratic new-path callback.

```
typedef ATSQuadraticNewPathProcPtr ATSQuadraticNewPathUPP;
```

Discussion
For more information, see the description of the `ATSQuadraticNewPathProcPtr` (page 163) callback function.

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

Declared In
`ATSUnicodeGlyphs.h`

RedrawBackgroundUPP

Defines a universal procedure pointer to a redraw-background callback.

```
typedef RedrawBackgroundProcPtr RedrawBackgroundUPP;
```

Discussion
For more information, see the description of the `RedrawBackgroundProcPtr` (page 165) callback function.

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

Declared In
`ATSUnicodeTypes.h`

Constants

Attribute Tags

Specify attributes that can be applied to a style object, a text layout object, or a line in a text layout object.

```
typedef UInt32 ATSUAttributeTag;
enum {
    kATSULineWidthTag = 1L,
    kATSULineRotationTag = 2L,
    kATSULineDirectionTag = 3L,
    kATSULineJustificationFactorTag = 4L,
    kATSULineFlushFactorTag = 5L,
    kATSULineBaselineValuesTag = 6L,
    kATSULineLayoutOptionsTag = 7L,
    kATSULineAscentTag = 8L,
    kATSULineDescentTag = 9L,
    kATSULineLangRegionTag = 10L,
    kATSULineTextLocatorTag = 11L,
    kATSULineTruncationTag = 12L,
    kATSULineFontFallbacksTag = 13L,
    kATSULineDecimalTabCharacterTag = 14L,
    kATSULayoutOperationOverrideTag = 15L,
    kATSULineHighlightCGColorTag  = 17L,
    kATSUMaxLineTag = 18L,
    kATSULineLanguageTag = 10L,
    kATSUCGContextTag = 32767L,
    kATSUQDBoldfaceTag = 256L,
    kATSUQDItalicTag = 257L,
    kATSUQDUnderlineTag = 258L,
    kATSUQDCondensedTag = 259L,
    kATSUQDExtendedTag = 260L,
    kATSUFontTag = 261L,
    kATSUSizeTag = 262L,
    kATSUColorTag = 263L,
    kATSULangRegionTag = 264L,
    kATSUVerticalCharacterTag = 265L,
    kATSUImposeWidthTag = 266L,
    kATSUBeforeWithStreamShiftTag = 267L,
    kATSUAfterWithStreamShiftTag = 268L,
    kATSUCrossStreamShiftTag = 269L,
    kATSUTrackingTag = 270L,
    kATSUHangingInhibitFactorTag = 271L,
    kATSUKerningInhibitFactorTag = 272L,
    kATSUDecompositionFactorTag = 273L,
    kATSUBaselineClassTag = 274L,
    kATSUPriorityJustOverrideTag = 275L,
    kATSUNoLigatureSplitTag = 276L,
    kATSUNoCaretAngleTag = 277L,
    kATSUSuppressCrossKerningTag = 278L,
    kATSUNoOpticalAlignmentTag = 279L,
    kATSUForceHangingTag = 280L,
    kATSUNoSpecialJustificationTag = 281L,
    kATSUStyleTextLocatorTag = 282L,
    kATSUStyleRenderingOptionsTag = 283L,
    kATSUAscentTag         = 284L,
    kATSUDescentTag = 285L,
    kATSULeadingTag = 286L,
    kATSUGlyphSelectorTag = 287L,
    kATSURGBAlphaColorTag     = 288L,
    kATSUFontMatrixTag = 289L,
    kATSUStyleUnderlineCountOptionTag = 290L,
    kATSUStyleUnderlineColorOptionTag = 291L,
    kATSUStyleStrikeThroughTag    = 292L,
    kATSUStyleStrikeThroughCountOptionTag = 293L,
    kATSUStyleStrikeThroughColorOptionTag = 294L,
    kATSUStyleDropShadowTag = 295L,
    kATSUStyleDropShadowBlurOptionTag = 296L,
    kATSUStyleDropShadowOffsetOptionTag = 297L,
    kATSUStyleDropShadowColorOptionTag = 298L,
    kATSUMaxStyleTag = 299L,
    kATSULanguageTag = 264L,
    kATSUMaxATSUITagValue = 65535L
};
```

Constants

`kATSULineWidthTag`

Specifies the desired width of a line of text, in typographic points, of the line when drawn as justified or right-aligned text. The associated value is of type `ATSUTextMeasurement` (page 179) and has a default value of `0`.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSULineRotationTag`

Specifies the angle by which the entire line should be rotated. The associated value is a `Fixed` value that specifies degrees in a right-hand coordinate system, and has a default value of `0`.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSULineDirectionTag`

Specifies a left-to-right or right-to-left direction for the glyphs in a text layout object, regardless of their natural direction as specified in the font. The associated value is `Boolean` (`kATSURightToLeftBaseDirection` or `kATSULeftToRightBaseDirection`) and has a default value of `GetSysDirection()`. See "Glyph Direction Selectors" (page 217) for more information on the values that can be associated with this tag.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSULineJustificationFactorTag`

Specifies how ATSUI should typographically fit a line of text to a given width (or height, in the case of vertical text). The associated value is a `Fract` value between 0 and 1 and has a default value of `kATSUNoJustification`. See "Line Justification Selectors" (page 224) for information on the values that can be associated with this tag.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSULineFlushFactorTag`

Specifies how ATSUI should place text in relation to one or both margins, which are the left and right sides (or top and bottom sides) of the text area. The associated value is a `Fract` value between 0 and 1 and has a default value of `kATSUStartAlignment`. See "Line Alignment Selectors " (page 223) for information on the values that can be associated with this tag.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSULineBaselineValuesTag`
: Specifies the positions of different baseline types with respect to one another in a line of text. The associated value is of type `BslnBaselineRecord` and contains default values all of which are `0`. The values are calculated from other style attributes such as font and point size.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineLayoutOptionsTag`
: Specifies how ATSUI should manipulate basic attributes of a line or the text layout object, such as whether a line should have optical hangers or whether the last line of a text layout object should be justified. The associated value is of type `ATSLineLayoutOptions` and has a default value of `kATSLineNoLayoutOptions`. See "Line Layout Attribute Tags" (page 224) for information on the values that can be associated with this tag.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineAscentTag`
: Specifies line ascent. The associated value is of type `ATSUTextMeasurement` (page 179) and has a default value of `kATSUseLineHeight`. See "Line Height and Font Tracking Selectors" (page 223) for information on the values that can be associated with this tag.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineDescentTag`
: Specifies line descent. The associated value is of type `ATSUTextMeasurement` (page 179) and has a default value of `kATSUseLineHeight`. See "Line Height and Font Tracking Selectors" (page 223) for information on the values that can be associated with this tag.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineLangRegionTag`
: Specifies line language region. The associated value is a region code (see the Script Manager reference for a list of region codes) and has a default value of `kTextRegionDontCare`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineTextLocatorTag`
: Specifies line text location. The associated value is of type `TextBreakLocatorRef` and has a default value of `NULL`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineTruncationTag`
: Specifies where in a line truncation should occur. The associated value is of type `ATSULineTruncation` and has a default value of `kATSUTruncateNone`. See "Line Truncation Selectors" (page 220) for the values that can be associated with the line truncation tag.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineFontFallbacksTag`
: Specifies line font fallbacks. The associated value is of type `ATSUFontFallbacks` (page 173). See "Font Fallback Methods" (page 214) for information on the values that can be associated with this tag.

  Available in Mac OS X v10.1 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineDecimalTabCharacterTag`
: Specifies the current setting for the decimal separator, and affects the behavior of decimal tabs for a text layout (not an individual line). The associated value is of type `CFStringRef`. The CFString object (`CFStringRef`) is retained by the style object in which it is set. The default value is the user setting in System Preferences.

  Declared in `ATSUnicodeTypes.h`.

  Available in Mac OS X version 10.3 and later.

`kATSULineHighlightCGColorTag`
: Specifies the current setting of the highlight color and opacity. The associated value is of type `CGColorRef`. This can be set as a line or layout control. The CGColor object (CGColorRef) is retained by the text layout object in which it is set.

  Available in Mac OS X version 10.3 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULayoutOperationOverrideTag`
: Specifies to override a layout operation. The associated value is of type `ATSULayoutOperationOverrideSpecifier` and has a default value of `NULL`.

  Available starting with Mac OS X version 10.2.

  Declared in `ATSUnicodeTypes.h`.

`kATSUMaxLineTag`
: A convenience tag that specifies the upper limit of the text layout attribute tags.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULineLanguageTag`
: Not recommended. Instead use `kATSULineLangRegionTag`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUCGContextTag`
: Specifies to use a Quartz context. When you use this tag to set up a Quartz context, ATSUI uses an 8-bit, sub-pixel rendering. This method of rendering positions glyph origins on fractional points, which results in superior rendering compared to ATSUI's default 4-bit pixel-aligned rendering. The attribute has a default value of `NULL`; you must provide a pointer to a `CGContext`. The `CGContext` is not retained by the text layout object; if the context is destroyed, the text layout contains an invalid `CGContext`. Available only in Mac OS X.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUQDBoldfaceTag`
: Specifies a boldface text style. Text style attribute tags are included for compatibility with the `Style` type used by the QuickDraw function `TextFace`. If a font variant for this text style exists, ATSUI uses that variant. Otherwise, the variant is generated algorithmically. The associated value is of type `Boolean` and has a default value of `false`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUQDItalicTag`
: Specifies an italic text style. Text style attribute tags are included for compatibility with the `Style` type used by the QuickDraw function `TextFace`. If a font variant for this text style exists, ATSUI uses that variant. Otherwise, the variant is generated. The associated value is of type `Boolean` and has a default value of `false`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUQDUnderlineTag`
: Specifies an underline text style. Text style attribute tags are included for compatibility with the `Style` type used by the QuickDraw function `TextFace`. If a font variant for this text style exists, ATSUI uses that variant. Otherwise, the variant is generated. The associated value is of type `Boolean` and has a default value of `false`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUQDCondensedTag`
: Specifies a condensed text style. Text style attribute tags are included for compatibility with the `Style` type used by the QuickDraw function `TextFace`. If a font variant for this text style exists, ATSUI uses that variant. Otherwise, the variant is generated. The associated value is of type `Boolean` and has a default value of `false`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUQDExtendedTag`
: Specifies an extended text style. Text style attribute tags are included for compatibility with the `Style` type used by the QuickDraw function `TextFace`. If a font variant for this text style exists, ATSUI uses that variant. Otherwise, the variant is generated The associated value is of type `Boolean` and has a default value of `false`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUFontTag`
: Specifies a unique value that identifies a font to the font management system. The associated value is of type `ATSUFontID` (page 173) and has a default value of `GetScriptVariable (smSystemScript, smScriptAppFond)`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUSizeTag`

Specifies the font size of the text in the style run. The associated value, in typographic points (72 per inch), is of type `Fixed` and has a default value of `GetScriptVariable (smSystemScript, smScriptAppFondSize)`.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUColorTag`

Specifies the color of the glyphs in a style run. The associated value is of type `RGBColor` and has a default value of `(0,0,0)`.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSULangRegionTag`

Specifies a language region. The associated value is a region code (see the Script Manager reference for a list of region codes) and has a default value of `GetScriptManagerVariable (smRegionCode)`.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUVerticalCharacterTag`

Specifies which direction (vertical or horizontal) glyphs should be drawn. The associated value is of type `ATSUVerticalCharacterType` and has a default value of `kATSUStronglyHorizontal`. See "Vertical Character Types" (page 234) for more information on the values that can be associated with this tag.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUImposeWidthTag`

Specifies an imposed width. The associated value is of type `ATSUTextMeasurement` (page 179) and has a default value of `0`; all glyphs use their own font defined advance widths.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUBeforeWithStreamShiftTag`

Specifies a uniform shift parallel to the baseline of the positions of individual pairs or sets of glyphs in the style run that's applied before (to the left) the glyphs of the style run. The associated value is of type `Fixed` and has a default value of `0`. Starting with Mac OS version 10.3, glyphs cannot be negatively shifted such that later glyphs appear before earlier glyphs. In other words, ATSUI limits the shift to a value that is, at most, the advance of the previous glyph.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUAfterWithStreamShiftTag`

Specifies a uniform shift parallel to the baseline of the positions of individual pairs or sets of glyphs in the style run that's applied after (to the right) the glyphs of the style run. The associated value is of type `Fixed` and has a default value of `0`. Starting with Mac OS version 10.3, glyphs cannot be negatively shifted such that later glyphs appear before earlier glyphs. In other words, ATSUI limits the shift to a value that is, at most, the advance of the current glyph.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUCrossStreamShiftTag`
: Specifies the distance to raise or lower glyphs in the style run perpendicular to the text stream. This shift is vertical for horizontal text and horizontal for vertical text. The associated value (in points, 72 per inch) is of type `Fixed` and has a default value of `0`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUTrackingTag`
: Specifies the relative proportion of font-defined adjustments to apply to interglyph positions. The associated value is of type `Fixed` and has a default value of `kATSNoTracking`. See "Line Height and Font Tracking Selectors" (page 223) for information on the values that can be associated with this tag.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUHangingInhibitFactorTag`
: Specifies to what degree punctuation glyphs can hang beyond the end of a line for justification purposes. The associated value is a `Fract` value between `0` and `1` and has a default value of `0`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUKerningInhibitFactorTag`
: Specifies how much to inhibit kerning; that is, the increase or decrease the space between glyphs. The associated value is a `Fract` value between `0` and `1` and has a default value of `0`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUDecompositionFactorTag`
: Specifies the fractional adjustment to the font-specified threshold at which ligature decomposition occurs during justification. The associated value is a `Fract` value between `-1.0` and `1.0` and has a default value of `0` (no adjustment to the font-specified threshold).

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUBaselineClassTag`
: Specifies the preferred baseline (such as Roman, hanging, or ideographic centered) to use for text of a given font in a style run. The associated value is of type `BslnBaselineClass` (see SFNTLayoutTypes.h) and has a default value of `kBSLNRomanBaseline`. You can set the value to `kBSLNNoBaselineOverride` to use intrinsic baselines.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUPriorityJustOverrideTag`
: Specifies the degree to which ATSUI should override justification behavior for glyphs in the style run. The associated value is of type `ATSJustWidthDeltaEntryOverride` (page 176). The default values in this structure are all `0`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUNoLigatureSplitTag`
: Specifies whether or not ligatures and compound characters in a style have divisible components. The associated value is a `Boolean` and has a default value of `false`; ligatures and compound characters have divisible components.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUNoCaretAngleTag`
: Specifies whether the text caret or edges of a highlighted area are always parallel to the slant of the style run's text or always perpendicular to the baseline. The associated value is a `Boolean` and has a default value of `false`; use the character's angularity to determine its boundaries.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUSuppressCrossKerningTag`
: Specifies whether or not to suppress cross kerning. The associated value is a `Boolean` and has a default value of `false`; do not suppress automatic cross kerning (defined by font).

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUNoOpticalAlignmentTag`
: Specifies the amount to which ATSUI should adjust glyph positions at the ends of lines to give a more even visual appearance to margins. The associated value is a `Boolean` and has a default value of `false`; do not suppress character's automatic optical positional alignment

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUForceHangingTag`
: Specifies to treat glyphs in a style run as hanging punctuation, whether or not the font designer intended them to be. The associated value is a `Boolean` and has a default value of `false`; do not force the character's to hang beyond the line boundaries

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUNoSpecialJustificationTag`
: Specifies whether processes (such as glyph stretching and ligature decomposition) that occur at the end of the justification process should be applied. The associated value is a `Boolean` and has a default value of `false`; perform post-compensation justification if needed

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUStyleTextLocatorTag`
: Specifies style text locator. The associated value is of type `TextBreakLocatorRef` and has a default value of `NULL`—region derived locator or the default Text Utilities locator.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUStyleRenderingOptionsTag`
: Specifies style rendering options. The associated value is of type `ATSUStyleRenderingOptions` and has a default value of `kATSStyleApplyHints`—ATS glyph rendering uses hinting. See "Style Rendering Options" (page 231) for more information on the values that can be associated with this tag.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUAscentTag`
: Specifies the ascent value of a style's font. The associated value is of type `ATSUTextMeasurement` (page 179) and has a default value of the ascent value of the style object's font with the current point size.

  Available starting with Mac OS X version 10.2.

  Declared in `ATSUnicodeTypes.h`.

`kATSUDescentTag`
: Specifies the descent value of a style's font. The associated value is of type `ATSUTextMeasurement` (page 179) and has a default value of the descent value of the style object's font with the current point size. The leading value is not included as par of the descent.

  Declared in `ATSUnicodeTypes.h`.

  Available starting with Mac OS X version 10.2.

`kATSULeadingTag`
: Specifies the leading value of a style's font. The associated value is of type `ATSUTextMeasurement` (page 179) and has a default value of the leading value of the style object's font with the current point size.

  Available starting with Mac OS X version 10.2.

  Declared in `ATSUnicodeTypes.h`.

`kATSUGlyphSelectorTag`
: Specifies a glyph collection. The associated value is an address to an `ATSUGlyphSelector` (page 175) data structure. Using this tag allows you access to characters in the fonts that otherwise would not be accessible. You can choose the variant glyph by providing a font-specific glyph ID or a CID. For more information on CID conventions, see http://www.adobe.com. You can get the variant glyph information from an input method through the Text Services Manager using the Carbon event key, `kEventParamTextInputGlyphInfoArray`.

  Declared in `ATSUnicodeTypes.h`.

  Available starting with Mac OS X version 10.2.

`kATSURGBAlphaColorTag`
: Specifies RGB color with an alpha channel. The associated value is of type `ATSURGBAlphaColor` and has a default value of (0,0,0,1).

  Available starting with Mac OS X version 10.2.

  Declared in `ATSUnicodeTypes.h`.

`kATSUFontMatrixTag`

Specifies a font transformation matrix. The associated value is of type `CGAffineTransform`. (See the Quartz 2D reference documentation for more information on this data type.) You can use a font matrix to achieve effects through ATSUI at a style-run level that were previously available only by changing settings directly in a `CGContext`. When you use the tag `kATSUFontMatrixTag`, you associate a font transformation matrix with an `ATSUStyle` object. You can set the values in the font transformation matrix to achieve such effects as reversing glyphs across the X-axes and rotating glyphs Note that ATSUI's layout uses the transformed metrics so layout will be effected and in some cases the effects might be unexpected. For example, for a transformation that mirrors the glyph across the Y-axes the metrics are in reverse and glyphs are rendered on top of each other.

Declared in `ATSUnicodeTypes.h`.

Available starting with Mac OS X version 10.2.

`kATSUStyleUnderlineCountOptionTag`

Specifies the number of strokes to be drawn for an underline. The associated value is of type `ATSUStyleLineCountType`. The default value is `kATSUStyleSingleLineCount`. May be set as a style attribute.

Available in Mac OS X version 10.3 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUStyleUnderlineColorOptionTag`

Specifies the color of the strokes to draw for an underlined run of text. The associated value is of type `CGColorRef`. The default value is `NULL`. If `NULL`, the text color is used. The CGColor object (`CGColorRef`) is retained by the style object in which it is set. May be set as a style attribute.

Declared in `ATSUnicodeTypes.h`.

Available in Mac OS X version 10.3 and later.

`kATSUStyleStrikeThroughTag`

Specifies strikethrough style. The associated value is of type `Boolean`. The default value is `false`. May be set as a style attribute.

Available in Mac OS X version 10.3 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUStyleStrikeThroughCountOptionTag`

Specifies the number of strokes to be drawn for a strikethrough. The associated value is of type `ATSUStyleLineCountType`. The default value is `kATSUStyleSingleLineCount`. May be set as a style attribute.

Available in Mac OS X version 10.3 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUStyleStrikeThroughColorOptionTag`

Specifies the color of the strokes to draw for a strikethrough style. The associated value is of type `CGColorRef`. The CGColor object (`CGColorRef`) is retained by the style object in which it is set. The default value is `NULL`. If `NULL`, the text color is used. May be set as a style attribute.

Declared in `ATSUnicodeTypes.h`.

Available in Mac OS X version 10.3 and later.

`kATSUStyleDropShadowTag`
: Specifies the text should be drawn with a drop shadow. The associated value is of type `Boolean`. The default value is `false`. Only takes effect if a CGContext is used for drawing. If you set this style attribute, you also need to set the drop shadow color using the tag `kATSUStyleDropShadowColorOptionTag`.

  Declared in `ATSUnicodeTypes.h`.

  Available in Mac OS X version 10.3 and later.

`kATSUStyleDropShadowBlurOptionTag`
: Specifies the amount of blur for a drop shadow. The associated value is of type `float`. The default value is `0.0`. May be set as a style attribute.

  Available in Mac OS X version 10.3 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUStyleDropShadowColorOptionTag`
: Specifies the color and opacity of a drop shadow. The associated value is of type `CGColorRef`. The default value is `NULL`. You need to set the `CGColorRef` to a value other than `NULL` if you want to see the drop shadow. May be set as a style attribute.

  Available in Mac OS X version 10.3 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUStyleDropShadowOffsetOptionTag`
: Specifies the amount of offset from the text to be used when drawing a drop shadow. The associated value is of type `CGSize`. The default value is (`3.0`, `-3.0`). May be set as a style attribute.

  Available in Mac OS X version 10.3 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUMaxStyleTag`
: A convenience tag that specifies the upper limit of style attribute tags.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULanguageTag`
: This tag is obsolete. Instead use `kATSULangRegionTag`.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUMaxATSUITagValue`
: Specifies this maximum Apple ATSUI reserved tag value. If you define a tag, it must have a value larger than the value of this tag.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

Discussion

An attribute tag cannot be used in versions of the Mac OS that are earlier than the version in which the tag was introduced. For example, a tag available in Mac OS version 10.2 cannot be used in Mac OS version 10.1 or earlier. You can call the function `Gestalt` to check version information for ATSUI.

Attribute tags indicates the particular type of attribute under consideration: font, size, color, and so on. Each style run may have at most one attribute with a given attribute tag (that is, a style run can't have more than one font or size) but may have none.

Some of the constants specify attributes that are applied to a style run, while other attributes are applied to an entire text layout object or to just a line in a text layout object. The constant descriptions assume horizontal text. If you set or get the an attribute that has been set for vertical text, you should interpret the constant descriptions accordingly.

Most of the constants in this section are described in further detail in *Inside Mac OS X: Rendering Unicode Text With ATSUI*. Where appropriate, that document provides illustrations that show the effect of applying an attribute. It also describes how to write code that sets style, line, and layout attributes.

A style run may have at most one style attribute with a given attribute tag. That is, a style run can't have more than one font or size attribute set but the style run does not need to have any attribute set explicitly.

When you set an attribute value for a line, the value overrides the attribute value set for the text layout object that contains the line. This is true even if you set line attributes before you set attributes for the entire text layout object that contains the line.

You can create your own attribute tag as long as your tag is outside those values reserved by Apple— 0 to 65,535 (0 to 0x0000FFFF). See *Rendering Unicode Text With ATSUI* for information on creating and registering your own attribute tags.

Background Data Types

Specify the data type of the background—a color or a callback.

```
typedef UInt32 ATSUBackgroundDataType;
enum {
    kATSUBackgroundColor = 0,
    kATSUBackgroundCallback = 1
};
```

Constants

`kATSUBackgroundColor`

Specifies the data type of the text background is a color.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUBackgroundCallback`

Specifies the data type of the text background is a callback.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Caret Movement Types

Specify the unit distance by which the caret moves.

```
typedef UInt16 ATSUCursorMovementType;
enum {
    kATSUByCharacter = 0,
    kATSUByTypographicCluster = 1,
    kATSUByWord = 2,
    kATSUByCharacterCluster = 3,
    kATSUByCluster = 1
};
```

Constants

`kATSUByCharacter`

Specifies to move the caret by a units based on single characters.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUByTypographicCluster`

Specifies to move the caret by units of clusters based on characters or ligatures.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUByWord`

Specifies to move the caret by units based on words.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUByCharacterCluster`

Specifies to move the caret by units based only on clusters of characters.

Available only in Mac OS X and in CarbonLib versions 1.3 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUByCluster`

An obsolete name for the constant `kATSUByTypographicCluster`.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion

A caret movement type is used to indicate the unit (character, word, and so on) by which to move the caret. You use these constants when you call the ATSUI caret movement functions. Functions that use caret movement types use this information to calculate the edge offset in memory that corresponds to the resulting cursor position.

Convenience Constants

Specify whether to clear values or whether drawing, measuring, or hit-testing should be done relative to the current pen location in the current graphics port.

```
enum {
    kATSUUseGrafPortPenLoc = (unsigned long)0xFFFFFFFF,
    kATSUClearAll = (unsigned long)0xFFFFFFFF
};
```

Constants

`kATSUUseGrafPortPenLoc`

Indicates that drawing, measuring, or hit-testing should be done relative to the current pen location in the current graphics port.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUClearAll`

Removes all previously set values from a style object, a single line, or a text layout object.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion

You can pass the `kATSUUseGrafPortPenLoc` constant to functions that operate on text layout objects to indicate that drawing, measuring, or hit-testing should be done relative to the current pen location in the current graphics port.

You can pass the `kATSUClearAll` constant to the following functions to remove previously set values from a style object: to `ATSUClearAttributes` (page 24) to remove style run attributes, to `ATSUClearFontFeatures` (page 25) to remove font features, and to `ATSUClearFontVariations` (page 26) to remove font variations.

You can also use the `kATSUClearAll` constant to remove previously set text layout attributes: to `ATSUClearLineControls` (page 29), to remove text layout attributes from a single line of a text layout object, and to `ATSUClearLayoutControls` (page 28) to remove text layout attributes from every line in a text layout object.

Direct Data Selectors

Specify the layout data to obtain when calling the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout`.

```
typedef UInt32 ATSUDirectDataSelector;
enum {
    kATSUDirectDataAdvanceDeltaFixedArray = 0L,
    kATSUDirectDataBaselineDeltaFixedArray = 1L,
    kATSUDirectDataDeviceDeltaSInt16Array = 2L,
    kATSUDirectDataStyleIndexUInt16Array = 3L,
    kATSUDirectDataStyleSettingATSUStyleSettingRefArray = 4L,
    kATSUDirectDataLayoutRecordATSLayoutRecordVersion1 = 100L,
    kATSUDirectDataLayoutRecordATSLayoutRecordCurrent =
            kATSUDirectDataLayoutRecordATSLayoutRecordVersion1
};
```

Constants

`kATSUDirectDataAdvanceDeltaFixedArray`

Specifies the parallel advance delta (delta X) array, which is an array of `Fixed` values. This array is created only on demand. If you plan to modify the data in this array, you should set the `iCreate` parameter to `true` when you call the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) to obtain this array.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeDirectAccess.h`.

`kATSUDirectDataBaselineDeltaFixedArray`

Specifies the parallel baseline delta (delta Y) array, which is an array of `Fixed` values. This array is created only on demand. If you plan to modify the data in this array, you should set the `iCreate` parameter to `true` when you call the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) to obtain this array.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeDirectAccess.h`.

`kATSUDirectDataDeviceDeltaSInt16Array`

Specifies the parallel device delta array, which is an array of `SInt16` values used to adjust truncated fractional values for devices that do not accept fractional positioning. The array specified by this selector is also used to provide precise positioning for connected scripts. This array is created only on demand. If you plan to modify the data in this array, you should set the `iCreate` parameter to `true` when you call the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) to obtain this array.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeDirectAccess.h`.

`kATSUDirectDataStyleIndexUInt16Array`

Specifies the parallel style index array, which is an array of (`UInt16`) values. The values in this array are indexes into the style setting reference (`ATSUStyleSettingRef`) array. This array is created only on demand. If you plan to modify the data in this array, you should set the `iCreate` parameter to `true` when you call the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) to obtain this array.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeDirectAccess.h`.

`kATSUDirectDataStyleSettingATSUStyleSettingRefArray`
: Specifies the style setting reference (`ATSUStyleSettingRef`) array. This array is always available if the text layout object has any text associated with it. Setting the `iCreate` parameter when you call the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) to obtain this array has no effect.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeDirectAccess.h`.

`kATSUDirectDataLayoutRecordATSLayoutRecordVersion1`
: Specifies the `ATSLayoutRecord` array, with the version 1 of the `ATSLayoutRecord` data structure. You should not use this selector. Instead use the selector `kATSUDirectDataLayoutRecordATSLayoutRecordCurrent` to ensure that your code uses the most current version of the `ATSLayoutRecord` data structure. ATSUI performs the most efficient processing only for the latest version of `ATSLayoutRecord` data structure. This array is always available if the text layout object has any text associated with it. Setting the `iCreate` parameter when you call the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) to obtain this array has no effect.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeDirectAccess.h`.

`kATSUDirectDataLayoutRecordATSLayoutRecordCurrent`
: Specifies the `ATSLayoutRecord` array, with the current version of the `ATSLayoutRecord` data structure. Always use this selector to get the array of `ATSLayoutRecord` data structures. This array is always available if the text layout object has any text associated with it. Setting the `iCreate` parameter when you call the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46) to obtain this array has no effect.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeDirectAccess.h`.

Discussion

You can provide direct data selectors to the functions `ATSUDirectGetLayoutDataArrayPtrFromLineRef` (page 45) or `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` (page 46).

Flattened Data Font Type Selectors

Specifies the data type for flattened font name data.

```
typedef UInt32 ATSFlatDataFontSpeciferType;
enum {
    kATSFlattenedFontSpecifierRawNameData = 'namd'
};
```

Constants

`kATSFlattenedFontSpecifierRawNameData`
: Specifies to use the font name as the flattened font name.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeFlattening.h`.

Flattened Data Format Selectors

Specify the format to use when flattening or unflattening data.

```
typedef UInt32 ATSUFlattenedDataStreamFormat;
enum {
    kATSUDataStreamUnicodeStyledText = 'ustl'
};
```

Constants

`kATSUDataStreamUnicodeStyledText`

Specifies to use the `'ustl'` data specification when flattening or unflattening data.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeFlattening.h`.

Flattened Style Run Data Options

Specify options to use when flattening ATSUI style run data.

```
typedef UInt32 ATSUFlattenStyleRunOptions;
enum {
        kATSUFlattenOptionNoOptionsMask = 0x00000000
};
```

Constants

`kATSUFlattenOptionNoOptionsMask`

Specifies that no options are to be used.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeFlattening.h`.

Discussion

Additional options may be added in the future.

Flattened Data Version Numbers

Specify versions of the `'ustl'` specification.

```
enum {
    kATSFlatDataUstlVersion0      = 0,
    kATSFlatDataUstlVersion1      = 1,
    kATSFlatDataUstlVersion2      = 2,
    kATSFlatDataUstlCurrentVersion = kATSFlatDataUstlVersion2};
```

Constants

`kATSFlatDataUstlVersion0`

Specifies version 0. This version is obsolete.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeFlattening.h`.

`kATSFlatDataUstlVersion1`

  Specifies version 1. This version is obsolete.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeFlattening.h`.

`kATSFlatDataUstlVersion2`

  Specifies version 2.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeFlattening.h`.

`kATSFlatDataUstlCurrentVersion`

  Specifies the current version.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeFlattening.h`.

Discussion

The ATSUI functions `ATSUFlattenStyleRunsToStream` and `ATSUUnflattenStyleRunsFromStream` operate on data that conform to version 2 of the `'ustl'` specification.

Font Fallback Methods

Specify the method by which ATSUI tries to find an appropriate font for a character if the assigned font does not contain the needed glyphs.

```
typedef UInt16 ATSUFontFallbackMethod;
enum {
    kATSUDefaultFontFallbacks = 0,
    kATSULastResortOnlyFallback = 1,
    kATSUSequentialFallbacksPreferred = 2,
    kATSUSequentialFallbacksExclusive = 3
};
```

Constants

`kATSUDefaultFontFallbacks`

  Specifies to use ATSUI's default font search method. ATSUI searches through all available fonts on the system for one that matches any text that cannot be drawn with the font specified in the current ATSU style object (`ATSUStyle`). ATSUI first searches in the standard application fonts for various languages. If that fails, it searches through the remaining fonts on the system in whatever order the Font Manager returns them. After ATSUI has searched all the fonts in the system, any unmatched text is drawn with the last-resort font.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSULastResortOnlyFallback`

  Specifies that ATSUI should use the last resort font if the assigned font does not contain the needed glyphs.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUSequentialFallbacksPreferred`
Specifies that ATSUI should first search sequentially through the list of supplied fonts before it searching through all available fonts on the system.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUSequentialFallbacksExclusive`
Specifies that ATSUI should search exclusively through the list of supplied fonts. ATSUI use the last-resort font if it does not find a match in the list of supplied fonts.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Glyph Origin Selectors

Specify which glyph origin to use to determine the width of the typographic glyph bounds.

```
enum {
    kATSUseCaretOrigins           = 0,
    kATSUseDeviceOrigins          = 1,
    kATSUseFractionalOrigins      = 2,
    kATSUseOriginFlags            = 3
};
```

Constants

`kATSUseCaretOrigins`
Specifies to use the caret origin to determine the width of the typographic glyph bounds. The caret origin is halfway between two characters.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSUseDeviceOrigins`
Specifies to use the glyph origin in device space to determine the width of the typographic glyph bounds. This is useful if you need to adjust text on the screen.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSUseFractionalOrigins`
Specifies to use the glyph origin in fractional absolute positions (which are uncorrected for display device) to determine the width of the typographic glyph bounds. This provides the ideal position of laid-out text and is useful if you need to scale text on the screen. The glyph origin is also used to obtain the width of the typographic bounding rectangle when you call the function `ATSUMeasureText`.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSUseOriginFlags`
The number of glyph origin selectors.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

Discussion

You can pass a glyph bounds selector in the `iTypeOfBounds` parameter of the function `ATSUGetGlyphBounds` (page 75) to indicate whether the width of the resulting typographic glyph bounds is determined using the caret origin, glyph origin in device space, or glyph origin in fractional absolute positions.

Glyph Collection Types

Specify a character set.

```
typedef UInt16 GlyphCollection;
enum {
    kGlyphCollectionGID           = 0,
    kGlyphCollectionAdobeCNS1     = 1,
    kGlyphCollectionAdobeGB1      = 2,
    kGlyphCollectionAdobeJapan1   = 3,
    kGlyphCollectionAdobeJapan2   = 4,
    kGlyphCollectionAdobeKorea1   = 5,
    kGlyphCollectionUnspecified   = 0xFF
};
```

Constants

`kGlyphCollectionGID`

Indicates that the glyph value represents the actual glyph ID of a specific font.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeTypes.h`.

`kGlyphCollectionAdobeCNS1`

Specifies Adobe CNS1 CID-keyed fonts.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeTypes.h`.

`kGlyphCollectionAdobeGB1`

Specifies Adobe GB1 CID-keyed fonts.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeTypes.h`.

`kGlyphCollectionAdobeJapan1`

Specifies Adobe Japan1 CID-keyed fonts.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeTypes.h`.

`kGlyphCollectionAdobeJapan2`

Specifies Adobe Japan2 CID-keyed fonts.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeTypes.h`.

`kGlyphCollectionAdobeKorea1`

Specifies Adobe Korea1 CID-keyed fonts.

Available in Mac OS X v10.2 and later.

Declared in `ATSUnicodeTypes.h`.

`kGlyphCollectionUnspecified`
: Indicates that the glyph collection is not specified.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSUnicodeTypes.h`.

Discussion
A CID-keyed font is a PostScript font that uses a font file format developed by Adobe for fonts that have large character sets, such as Chinese, Japanese, and Korean fonts. For more information on CID-keyed fonts, see the Adobe website:

http://partners.adobe.com/

Glyph Direction Selectors

Specify a glyph direction.

```
enum {
    kATSULeftToRightBaseDirection = 0,
    kATSURightToLeftBaseDirection = 1
};
```

Constants

`kATSULeftToRightBaseDirection`
: Imposes left-to-right direction on glyphs in a line of horizontal text; for vertical text, imposes top-to-bottom direction.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSURightToLeftBaseDirection`
: Imposes right-to-left direction on glyphs in a line of horizontal text; for vertical text, imposes bottom-to-top direction.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

Discussion
These constants specify values for the `kATSULineDirectionTag` attribute tag. You can use one of these constants to set or obtain glyph direction in a line of text or an entire text layout object, regardless of their font-specified direction; see the functions `ATSUSetLayoutControls` (page 122), `ATSUSetLineControls` (page 124), `ATSUGetLayoutControl` (page 82), and `ATSUGetLineControl` (page 83).

Glyph Property Flags

Specify properties for a glyph.

```
typedef UInt32 ATSGlyphInfoFlags;
enum {
    kATSGlyphInfoAppleReserved    = 0x1FFBFFE8,
    kATSGlyphInfoIsAttachment     = (unsigned long)0x80000000,
    kATSGlyphInfoIsLTHanger       = 0x40000000,
    kATSGlyphInfoIsRBHanger       = 0x20000000,
    kATSGlyphInfoTerminatorGlyph  = 0x00080000,
    kATSGlyphInfoIsWhiteSpace     = 0x00040000,
    kATSGlyphInfoHasImposedWidth  = 0x00000010,
    kATSGlyphInfoByteSizeMask     = 0x00000007
};
```

Constants

`kATSGlyphInfoAppleReserved`

This flag is reserved by Apple. If you try to use it you may get an invalid value error.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSGlyphInfoIsAttachment`

Specifies that the glyph attaches to another glyph.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSGlyphInfoIsLTHanger`

Specifies that the glyph can hang off the left or top edge of a line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSGlyphInfoIsRBHanger`

Specifies that the glyph can hang off the right or bottom edge of a line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSGlyphInfoTerminatorGlyph`

Specifies that the glyph is not truly a glyph, but an end-marker to allow the calculation of the previous glyph's advance.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSGlyphInfoIsWhiteSpace`

Specifies that the glyph is a whitespace glyph.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSGlyphInfoHasImposedWidth`

Specifies that the glyph has an imposed width (that is, an advance width) specified by the style.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSGlyphInfoByteSizeMask`

Specifies the size of the character that spawned the glyph. This is a three-bit mask that you can use to obtain the size of the original character that spawned a glyph. If you perform a logical `and` operation between this mask and an `ATSGlyphInfoFlags` flag, you obtain the size in bytes of the original character (0 - 7 bytes).

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

Discussion

Glyph information flags are set in the individual `ATSLayoutRecord` structure and apply only to the `ATSGlyphRef` reference in that structure. The flags are used by the ATSUI to tag a glyph with one or more specific properties.

Highlight Methods

Specify a text highlighting method.

```
typedef UInt32 ATSUHighlightMethod;
enum {
    kInvertHighlighting = 0,
    kRedrawHighlighting = 1
};
```

Constants

`kInvertHighlighting`

Specifies to use inversion for highlighting. You can use this when the background is a single color.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kRedrawHighlighting`

Specifies to use your callback for highlighting. You should use this when the background is complex (containing, for example, multiple colors, patterns, or pictures).

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion

You set the highlighting method by calling the function `ATSUSetHighlightingMethod` (page 121).

Invalid Font ID Constant

Specifies a Font ID is not valid.

```
enum {
    kATSUInvalidFontID = 0
};
```

Constants

`kATSUInvalidFontID`

Indicates that the font ID is invalid.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion
The functions `ATSUFONDtoFontID` (page 56), `ATSUFindFontFromName` (page 52), and `ATSUMatchFontsToText` (page 106) pass back this constant to indicate an invalid font ID. This constant is available with ATSUI 1.0.

Line Truncation Selectors

Specify where in a line truncation should occur.

```
typedef UInt32 ATSULineTruncation;
enum {
    kATSUTruncateNone = 0,
    kATSUTruncateStart = 1,
    kATSUTruncateEnd = 2,
    kATSUTruncateMiddle = 3,
    kATSUTruncateSpecificationMask = 7,
    kATSUTruncFeatNoSquishing = 8
};
```

Constants

`kATSUTruncateNone`
Specifies not to truncate the line.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUTruncateStart`
Specifies to truncate the line at the beginning.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUTruncateEnd`
Specifies to truncate the line at the end.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUTruncateMiddle`
Specifies to truncate the line in the middle

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUTruncateSpecificationMask`
Reserved for the truncation specification (0 - 7).

Available in Mac OS X v10.1 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUTruncFeatNoSquishing`
Specifies not to perform any negative justification in lieu of truncation.

Available in Mac OS X v10.1 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion
Line truncation options specify values for the `kATSULineTruncation` attribute tag. You can add any line truncation option to the option `kATSUTruncateSpecificationMask`. For example, adding `kATSUTruncateEnd` and `kATSUTruncFeatNoSquishing` to the mask `kATSUTruncateSpecificationMask` results in the value `0x0000000A`.

Layout Callback Status Values

Specify the status of a layout operation override callback.

```
typedef UInt32 ATSULayoutOperationCallbackStatus;
enum {
    kATSULayoutOperationCallbackStatusHandled = 0x00000000,
    kATSULayoutOperationCallbackStatusContinue = 0x00000001
};
```

Constants

`kATSULayoutOperationCallbackStatusHandled`
Specifies that your callback function has handled the operation which triggered the callback. This indicates to ATSUI that it does not need to perform any further processing for the layout operation.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationCallbackStatusContinue`
Specifies that your callback function has not handled the operation which triggered the callback. This indicates to ATSUI that needs to perform its own processing for the layout operation.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

Discussion
You must return one of these status values from your `ATSUDirectLayoutOperationOverrideProcPtr` (page 164) callback function to indicate to ATSUI whether or not your callback handled the layout operation.

Layout Operation Selectors

Specify a layout operation.

```
typedef UInt32 ATSULayoutOperationSelector;
enum {
    kATSULayoutOperationNone      = 0x00000000,
    kATSULayoutOperationJustification = 0x00000001,
    kATSULayoutOperationMorph     = 0x00000002,
    kATSULayoutOperationKerningAdjustment = 0x00000004,
    kATSULayoutOperationBaselineAdjustment = 0x00000008,
    kATSULayoutOperationTrackingAdjustment = 0x00000010,
    kATSULayoutOperationPostLayoutAdjustment = 0x00000020,
    kATSULayoutOperationAppleReserved = (unsigned long)0xFFFFFFC0
};
```

Constants

`kATSULayoutOperationNone`

Specifies that no layout operation is currently selected.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationJustification`

Specifies the justification operation.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationMorph`

Specifies the character-morphing operation.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationKerningAdjustment`

Specifies the kerning-adjustment operation.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationBaselineAdjustment`

Specifies the baseline-adjustment operation.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationTrackingAdjustment`

Specifies the tracking-adjustment operation.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationPostLayoutAdjustment`

Specifies the period of time after ATSUI has completed its layout operations.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSULayoutOperationAppleReserved`

This selector is reserved for future use.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

Discussion

You can use layout operation selectors to specify to ATSUI which operations to override. These selectors can also be passed from ATSUI to your application to indicate which operation is currently in progress.

Line Alignment Selectors

Specify the alignment of text relative to the margins in a line of text or in an entire text layout object.

```
#define kATSUStartAlignment ((Fract) 0x00000000L)
#define kATSUEndAlignment ((Fract) 0x40000000L)
#define kATSUCenterAlignment ((Fract) 0x20000000L)
```

Constants

`kATSUStartAlignment`

Specifies that horizontal text should be drawn to the right of the left margin (that is, its left edge coincides with the text layout object's position plus text width). Vertical text should be drawn below the top margin.

`kATSUEndAlignment`

Specifies that horizontal text should be drawn to the left of the right margin. Vertical text should be drawn above the bottom margin.

`kATSUCenterAlignment`

Specifies that horizontal text should be drawn between the left and right margins with an equal amount of space on either side. Vertical text should be drawn between the top and bottom margins with an equal amount of space on either side.

Discussion

You can use one of these constants to set or obtain the alignment of text relative to the margins in a line of text or in an entire text layout object; see the functions `ATSUSetLayoutControls` (page 122), `ATSUSetLineControls` (page 124), `ATSUGetLayoutControl` (page 82), and `ATSUGetLineControl` (page 83), respectively.

Line Height and Font Tracking Selectors

Specify how to determine line height and whether to turn off font tracking.

```
enum {
    kATSUseGlyphAdvance = 0x7FFFFFFF,
    kATSUseLineHeight = 0x7FFFFFFF,
    kATSNoTracking = (long)0x80000000
};
```

Constants

`kATSUseGlyphAdvance`

Specifies that ATSUI use the natural glyph advance value in a line or entire text layout object.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSUseLineHeight`

Specifies that ATSUI use the natural line ascent and descent values dictated by the font and pixel size to determine line ascent and descent in a line or entire text layout object.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSNoTracking`
: A value of type `negativeInfinity` that indicates that font tracking should be off.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSLayoutTypes.h`.

Discussion

You use line height selectors to set line ascent and descent text layout attributes. You can set the line ascent text layout attribute for a line or an entire text layout object by passing the `kATSULineAscentTag` tag to the functions `ATSUSetLineControls` (page 124) and `ATSUSetLayoutControls` (page 122), respectively. You can set the line descent text layout attribute for a line or an entire text layout object by passing the `kATSULineDescentTag` tag to the functions `ATSUSetLineControls` (page 124) and `ATSUSetLayoutControls` (page 122), respectively.

Line Justification Selectors

Specify the degree of line justification for a single line or an entire text layout object.

```
#define kATSUNoJustification        ((Fract) 0x00000000L)
#define kATSUFullJustification      ((Fract) 0x40000000L)
```

Constants

`kATSUNoJustification`
: Indicates no justification.

`kATSUFullJustification`
: Full justification between the text margins. White space is “stretched” to make the line extend to both text margins.

Discussion

You can set the line justification text layout attribute for a line or an entire text layout object by passing the `kATSULineJustificationFactorTag` tag the functions `ATSUSetLineControls` (page 124) and `ATSUSetLayoutControls` (page 122), respectively.

Line Layout Attribute Tags

Specify line layout attributes to be applied at the line level.

```
typedef UInt32 ATSLineLayoutOptions;
enum {
    kATSLineNoLayoutOptions       = 0x00000000,
    kATSLineIsDisplayOnly         = 0x00000001,
    kATSLineHasNoHangers          = 0x00000002,
    kATSLineHasNoOpticalAlignment = 0x00000004,
    kATSLineKeepSpacesOutOfMargin = 0x00000008,
    kATSLineNoSpecialJustification = 0x00000010,
    kATSLineLastNoJustification   = 0x00000020,
    kATSLineFractDisable          = 0x00000040,
    kATSLineImposeNoAngleForEnds  = 0x00000080,
    kATSLineFillOutToWidth        = 0x00000100,
    kATSLineTabAdjustEnabled      = 0x00000200,
    kATSLineIgnoreFontLeading     = 0x00000400,
    kATSLineApplyAntiAliasing     = 0x00000800,
    kATSLineNoAntiAliasing        = 0x00001000,
    kATSLineDisableNegativeJustification = 0x00002000,
    kATSLineDisableAutoAdjustDisplayPos = 0x00004000,
    kATSLineUseQDRendering        = 0x00008000,
    kATSLineDisableAllJustification = 0x00010000,
    kATSLineDisableAllGlyphMorphing = 0x00020000,
    kATSLineDisableAllKerningAdjustments = 0x00040000,
    kATSLineDisableAllBaselineAdjustments = 0x00080000,
    kATSLineDisableAllTrackingAdjustments = 0x00100000,
    kATSLineDisableAllLayoutOperations = kATSLineDisableAllJustification
|
        kATSLineDisableAllGlyphMorphing |
        kATSLineDisableAllKerningAdjustments |
        kATSLineDisableAllBaselineAdjustments |
        kATSLineDisableAllTrackingAdjustments,
    kATSLineUseDeviceMetrics      = 0x01000000,
    kATSLineBreakToNearestCharacter = 0x02000000,
    kATSLineAppleReserved         = (unsigned long)0xFCE00000};
```

Constants

`kATSLineNoLayoutOptions`

Specifies not to apply any options.

Available i n ATSUI 1.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineIsDisplayOnly`

This line option is no longer used. Instead use `kATSLineUseDeviceMetrics`.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineHasNoHangers`

Specifies not to form hanging punctuation on the line. If the bit specified by this mask is set, the automatic hanging punctuation in the text layout object is overridden. The value in this bit overrides any adjustment to hanging punctuation set for a style run inside the text layout object using the style run attribute tags `kATSUForceHangingTag` or `kATSUHangingInhibitFactorTag`.

Declared in `ATSLayoutTypes.h`.

Available in ATSUI 1.0 and later.

`kATSLineHasNoOpticalAlignment`
: Specifies not to perform optical alignment on the line. Optical alignment adjusts characters at the text margin so that they appear to be properly aligned; strict alignment can often cause the illusion of a ragged edge. The value in this bit overrides any adjustment to optical alignment set for a style run inside the text layout object using the style run attribute tag `kATSUNoOpticalAlignmentTag`.

  Declared in `ATSLayoutTypes.h`.

  Available in ATSUI 1.0 and later.

`kATSLineKeepSpacesOutOfMargin`
: Specifies that the trailing white spaces at the end of a line of justified text should be placed outside the margin.

  Available in ATSUI 1.0 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineNoSpecialJustification`
: Specifies not to perform post-compensation justification on the line, even if such processing is necessary. This flag cannot be set for a single line of a text layout object. The value in this bit overrides any adjustment to the postcompensation actions set for a style run using the style run attribute tag `kATSUNoSpecialJustificationTag`.

  Declared in `ATSLayoutTypes.h`.

  Available in ATSUI 1.0 and later.

`kATSLineLastNoJustification`
: Specifies not to justify a line if it is the last line of a justified text layout object. This flag is meaningless when setting a line's text layout attributes.

  Available in ATSUI 1.0 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineFractDisable`
: Specifies to position of the text in the line or text layout object relative to fractional absolute positions, which are uncorrected for device display. This provides the ideal position of laid-out text and is useful for scaling text onscreen. This origin is also used to get the width of the typographic bounding rectangle when you call the function `ATSUGetUnjustifiedBounds` (page 93).

  Declared in `ATSLayoutTypes.h`.

  Available in ATSUI 1.1 and later.

`kATSLineImposeNoAngleForEnds`
: Specifies to draw the carets on the far right and left sides of an unrotated line as vertical, no matter what the angle of text.

  Available in ATSUI 1.1 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineFillOutToWidth`
: Specifies to extend highlighting to both ends of a line, regardless of caret locations. This option does not effect the caret locations. This is provided for your convenience to extend your highlighting to the full width of the line.

  Available in ATSUI 1.1 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineTabAdjustEnabled`
: Specifies to automatically adjust the tab character width so that it fits the specified line width. If you are using ATSUI's tab functions—`ATSUSetTabArray` (page 129) and `ATSUGetTabArray` (page 89) to define a tab rule you do not need to use this selector. The selector is useful if you are handling your own tabs and only applies if the tab is at the end of a line (backing store). You must set this bit to ensure that highlighting is done correctly across tab stops. To ensure this, you should also set the bit specified by the `kATSLineImposeNoAngleForEnds` mask constant.

  Declared in `ATSLayoutTypes.h`.

  Available in ATSUI 1.2 and later.

`kATSLineIgnoreFontLeading`
: Specifies to ignore any leading value specified by a font.

  Available in ATSUI 2.3 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineApplyAntiAliasing`
: Specifies that Apple Type Services should produce antialiased glyph images even if system preferences or Quartz settings indicate otherwise.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineNoAntiAliasing`
: Specifies that Apple Type Services should turn-off antialiasing glyph imaging even if system preferences or Quartz settings indicate otherwise. This option negates the `kATSLineApplyAntiAliasing` bit if it is set.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineDisableNegativeJustification`
: Specifies to allow glyph positions to extend beyond the line's assigned width if the line width is not sufficient to hold all its glyphs. This ensures that negative justification is not used.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineDisableAutoAdjustDisplayPos`
: Specifies not to automatically adjust individual character positions when rendering lines that have any integer glyph positioning, whether the integer glyph positioning is due to non-antialiased characters or though the use of the selector `kATSLineFractDisable`.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineUseQDRendering`
: Specifies to use QuickDraw to render a line of text instead of the default ATSUI rendering. With Mac OS X version 10.2, ATSUI renders text through Quartz, even if you do not attach a `CGContext` to a text layout object. In the default case, ATSUI retrieves the internal canonical `CGContext` of the current port, and renders to that port using Quartz at an antialiasing setting that simulates QuickDraw rendering. That is, a 4-bit pixel-aligned antialiasing. Because the default setting gives you simulated QuickDraw rendering, you should use the tag `kATSLineUseQDRendering` only if you must have backward compatibility. With Mac OS X version 10.3, this option no longer does anything different from not declaring a `CGContext`.

  Available in Mac OS X v10.2 and later.

  Declared in `ATSLayoutTypes.h`.

`kATSLineDisableAllJustification`
Specifies not to perform any justification operations on the line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineDisableAllGlyphMorphing`
Specifies not to perform any glyph-morphing operations on the line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineDisableAllKerningAdjustments`
Specifies not to perform any kerning-adjustment operations on the line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineDisableAllBaselineAdjustments`
Specifies not to perform any baseline-adjustment operations on the line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineDisableAllTrackingAdjustments`
Specifies not to perform any tracking-adjustment operations on the line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineDisableAllLayoutOperations`
Specifies to turn off all layout adjustments for this line.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineUseDeviceMetrics`
Specifies to used rounded device metrics instead of fractional path metrics. This optimizes display of text and should be used only in cases in which the text is displayed onscreen as opposed to printed or output to PDF. If you use this option to display text onscreen as well as to print or create a PDF, you will get different results between the two types of output. This attribute is not recommended for Quartz antialiased text.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineBreakToNearestCharacter`
Specifies that line breaking should occur at the nearest character, not word. This could cause a word to be split over multiple lines.

Available in Mac OS X version 10.3 and later.

Declared in `ATSLayoutTypes.h`.

`kATSLineAppleReserved`
This selector is reserved by Apple. If you try to use it, ATSUI returns the `kATSUInvalidAttributeValueEr` result code.

Available in ATSUI 1.1 and later.

Declared in `ATSLayoutTypes.h`.

Discussion
You can use a constant of type `ATSLineLayoutOptions` to set or obtain the line layout options in a line of text or for an entire text layout object; see the functions `ATSUSetLineControls` (page 124) and `ATSUSetLayoutControls` (page 122), respectively.

Line Layout Width Selector

Specifies a line width.

```
enum {
    kATSUUseLineControlWidth = 0X7FFFFFFF
};
```

Constants
`kATSUUseLineControlWidth`
Indicates that the functions `ATSUBreakLine` or `ATSUBatchBreakLines` should use the previously set line width attribute for the current line to determine how many characters can fit on the line. If no line width has been set for the line, these functions use the line width set for the text layout object; if not set, these functions use the default line width value.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion
You can pass this constant to the functions `ATSUBreakLine` (page 22) or `ATSUBatchBreakLines` (page 20) to indicate that the function should use the line width previously set for that line to calculate the soft line break. If no line width has been set for the line, these functions use the line width set for the text layout object.

No Selectors Option

Specifies no selectors are chosen.

```
enum {
    kATSUNoSelector = 0x0000FFFF
};
```

Constants
`kATSUNoSelector`
Specifies no selectors are chosen.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Style Comparison Options

Specify how two style objects compare to each other.

```
typedef UInt16 ATSUStyleComparison;
enum {
    kATSUStyleUnequal = 0,
    kATSUStyleContains = 1,
    kATSUStyleEquals = 2,
    kATSUStyleContainedBy = 3
};
```

Constants

`kATSUStyleUnequal`

Specifies that styles are unequal.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUStyleContains`

Specifies that style 1 contains style 2 as a proper subset.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUStyleEquals`

Specifies that style 1 equals style 2.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUStyleContainedBy`

Specifies that style 1 is contained by style 2.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion

The function `ATSUCompareStyles` (page 31) returns a constant of type `ATSUStyleComparison` to indicate whether two style objects are the same, different, or a subset of one another.

Style Line Count Types

Specifies how many lines to draw for a given style type.

```
typedef UInt16 ATSUStyleLineCountType;
enum {
    kATSUStyleSingleLineCount     = 1,
    kATSUStyleDoubleLineCount     = 2
};
```

Constants

`kATSUStyleSingleLineCount`

Specifies to use a single line.

Available in Mac OS X v10.3 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUStyleDoubleLineCount`

Specifies to use a double line.

Available in Mac OS X v10.3 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion
These constants are available in Mac OS X version 10.3 and later. Currently only the underline and strike through styles support this type.

Style Rendering Options

Specify rendering options for a style object.

```
typedef UInt32 ATSStyleRenderingOptions;
enum {
    kATSStyleNoOptions            = 0x00000000,
    kATSStyleNoHinting            = 0x00000001,
    kATSStyleApplyAntiAliasing    = 0x00000002,
    kATSStyleNoAntiAliasing       = 0x00000004,
    kATSStyleAppleReserved        = (unsigned long)0xFFFFFFF8,
    kATSStyleApplyHints           = kATSStyleNoOptions
};
```

Constants

`kATSStyleNoOptions`

Specifies no options are set.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSStyleNoHinting`

Specifies that Apple Type Services (ATS) should produce unhinted glyph outlines. The default behavior is for ATS to produce is hinted glyph outlines.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSStyleApplyAntiAliasing`

Specifies that Apple Type Services should produce antialiased glyph images even if system preferences or Quartz 2D settings indicate otherwise.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSStyleNoAntiAliasing`

Specifies that Apple Type Services should turn-off antialiasing glyph imaging even if system preferences or Quartz 2D settings indicate otherwise. This selector negates the `kATSStyleApplyAntiAliasing` selector if is set.

Available in Mac OS X v10.2 and later.

Declared in `ATSLayoutTypes.h`.

`kATSStyleAppleReserved`

This selector is reserved by Apple. If you try to use it, you will get an invalid value error.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

`kATSStyleApplyHints`

Specifies that Apple Type Services should produce hinted glyph outlines. This selector is obsolete; do not use it. It is listed here only for backwards compatibility.

Available in Mac OS X v10.0 and later.

Declared in `ATSLayoutTypes.h`.

Discussion
You can set style rendering attributes for a style object (`ATSUStyle`) by using the `kATSUStyleRenderingOptionsTag` attribute tag and calling the function `ATSUSetAttributes` (page 119). Style rendering options provide fine control over how a style is rendered.

Tab Positioning Options

Specify text positioning for ATSUI tab stops.

```
typedef UInt16 ATSUTabType;
enum {
    kATSULeftTab                  = 0,
    kATSUCenterTab                = 1,
    kATSURightTab                 = 2,
    kATSUDecimalTab               = 3,
    kATSUNumberTabTypes           = 4
};
```

Constants

`kATSULeftTab`
    Specifies that the left side of the tabbed text should be flush against the tab stop.

    Available in Mac OS X v10.2 and later.

    Declared in `ATSUnicodeTypes.h`.

`kATSUCenterTab`
    Specifies that the tabbed text should be centered on the tab stop.

    Available in Mac OS X v10.2 and later.

    Declared in `ATSUnicodeTypes.h`.

`kATSURightTab`
    Specifies that the right side of the tabbed text should be flush against the tab stop.

    Available in Mac OS X v10.2 and later.

    Declared in `ATSUnicodeTypes.h`.

`kATSUDecimalTab`
    Specifies that the decimal point of a value should be centered on the tab stop. To set a decimal tab, use the tag `kATSULineDecimalTabCharacterTag`. This tag specifies the current setting for the decimal separator, and affects the behavior of decimal tabs for a text layout (not an individual line). The default character that is used as a separator is set by the user in System Preferences.

    Declared in `ATSUnicodeTypes.h`.

    Available in Mac OS X version 10.3 and later.

`kATSUNumberTabTypes`
    Specifies the number of valid tab types.

    Available in Mac OS X v10.2 and later.

    Declared in `ATSUnicodeTypes.h`.

Discussion
You can use tab type constants to set a tab ruler. The default value is the user setting in System Preferences.

Text Buffer Convenience Constants

Refer to the beginning or end of a text buffer.

```
enum {
    kATSUFromTextBeginning = (unsigned long)0xFFFFFFFF,
    kATSUToTextEnd = (unsigned long)0xFFFFFFFF,
    kATSUFromPreviousLayout = (unsigned long)0xFFFFFFFE,
    kATSUFromFollowingLayout      = (unsigned long)0xFFFFFFFD
};
```

Constants

`kATSUFromTextBeginning`

Indicates that the range of text to be operated on should start at the beginning of the text layout object's text buffer.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUToTextEnd`

Indicates that the range of text to be operated on should span to the end of the text layout object's text buffer.

Available in Mac OS X v10.0 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUFromPreviousLayout`

Used for bidirectional cursor movement between paragraphs in the functions `ATSURightwardCursorPosition` and `ATSULeftwardCursorPosition`.

Available in Mac OS X v10.3 and later.

Declared in `ATSUnicodeTypes.h`.

`kATSUFromFollowingLayout`

Used for bidirectional cursor movement between paragraphs in the functions `ATSURightwardCursorPostion` and `ATSULeftwardCursorPosition`.

Available in Mac OS X v10.3 and later.

Declared in `ATSUnicodeTypes.h`.

Discussion

Do not pass these constants to functions which do not explicitly state they will accept them.

ATSUI functions that draw, highlight, measure, or otherwise operate on text do so to a range of text, not the entire text buffer (unless you specify the entire buffer). You specify the beginning of this range with an edge offset of type `UniCharArrayOffset`, and demarcate the end of the range by indicating a length of type `UniCharCount`.

If you want the range to start at the beginning of the text buffer, you should pass the constant `kATSUFromTextBeginning`. If you want the range to span the end of the text buffer, you should pass the constant `kATSUToTextEnd`. If you want the range to span the entire text buffer, pass `kATSUFromTextBeginning` in conjunction with the constant `kATSUToTextEnd`. For bidirectional text, you can specify the previous layout by passing the constant `kATSUFromPreviousLayout` and the following layout by passing the constant `kATSUFromFollowingLayout`.

Unflattened Style Run Data Options

Specify options to use when unflattening ATSUI style run data.

```
typedef UInt32 ATSUUnflattenStyleRunOptions;
enum {
    kATSUUnflattenOptionNoOptionsMask = 0x00000000
};
```

Constants

`kATSUUnflattenOptionNoOptionsMask`
  Specifies that no options are to be used.

Discussion

Additional options may be added in the future.

Vertical Character Types

Specify the glyph orientation of font tracking settings or a style run.

```
typedef UInt16 ATSUVerticalCharacterType;
enum {
    kATSUStronglyHorizontal = 0,
    kATSUStronglyVertical = 1
};
```

Constants

`kATSUStronglyHorizontal`
  Specifies a horizontal orientation.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

`kATSUStronglyVertical`
  Specifies a vertical orientation.

  Available in Mac OS X v10.0 and later.

  Declared in `ATSUnicodeTypes.h`.

Discussion

You can pass a constant of type `ATSUVerticalCharacterType` to the functions `ATSUCountFontTracking` (page 37) and `ATSUGetIndFontTracking` (page 80) to specify the glyph orientation of font tracking settings, since font tracking settings differ depending upon glyph orientations.

You can also use one of these constants to set or obtain the glyph orientation of a style run; see the functions `ATSUSetAttributes` (page 119) and `ATSUGetAttribute` (page 65), respectively.

Result Codes

The most common result codes returned by Apple Type Services for Unicode Imaging are listed below.

Result Code	Value	Description
`kATSUInvalidTextLayoutErr`	-8790	The ATSUI text layout object is not initialized or is in an otherwise invalid state. Available beginning with ATSUI 1.0.

Result Code	Value	Description
`kATSUInvalidStyleErr`	-8791	The ATSUI style object is not initialized or in an otherwise invalid state. Available beginning with ATSUI 1.0.
`kATSUInvalidTextRangeErr`	-8792	The text range extends beyond the limits of the text layout object's text range. Available beginning with ATSUI 1.0.
`kATSUFontsMatched`	-8793	One or more characters cannot be rendered with the assigned font, but can be rendered with a substitute font from among those currently active. Available beginning with ATSUI 1.0.
`kATSUFontsNotMatched`	-8794	One or more characters can neither be rendered with the assigned font nor with any other currently active font. Available beginning with ATSUI 1.0.
`kATSUNoCorrespondingFontErr`	-8795	The font ID corresponds to an existing font that isn't available to ATSUI. Available beginning with ATSUI 1.0.
`kATSUInvalidFontErr`	-8796	The font ID does not correspond to any installed font or ATSUI is unable to obtain the font data (as in the case of a protected font). Available beginning with ATSUI 1.0.
`kATSUInvalidAttributeValueErr`	-8797	The attribute value is invalid or undefined. Available beginning with ATSIU 1.0.
`kATSUInvalidAttributeSizeErr`	-8798	The allocated attribute value size is less than required. Available beginning with ATSUI 1.0.
`kATSUInvalidAttributeTagErr`	-8799	The tag is an ATSUI-reserved value or the wrong type of attribute tag (that is, a style run attribute tag instead of text layout attribute tag and vice versa). Available beginning with ATSUI 1.0.
`kATSUInvalidCacheErr`	-8800	Indicates an attempt to read style data from an invalid cache (that is, the format of the cached data does not match that used by ATSUI or the cached data is corrupt). Available beginning with ATSUI 1.0.

Result Code	Value	Description
`kATSUNotSetErr`	-8801	The ATSUI style object's attribute, font feature, font variation is not set; the ATSUI text layout object or single line's attribute is not set; or a font name is not set. Available beginning with ATSUI 1.0.
`kATSUNoStyleRunsAssignedErr`	-8802	No style runs are assigned to the ATSUI text layout object. Available beginning with ATSUI 1.1.
`kATSUQuickDrawTextErr`	-8803	The QuickDraw function `DrawText` encountered an error rendering or measuring a line of text. Available beginning with ATSUI 1.1.
`kATSULowLevelErr`	-8804	Apple Type Services (ATS) encountered an error while performing an operation requested by ATSUI. Available beginning with ATSUI 1.1.
`kATSUNoFontCmapAvailableErr`	-8805	The `'CMAP'` table cannot be accessed or synthesized for a font. Available beginning with ATSUI 1.1.
`kATSUNoFontScalerAvailableErr`	-8806	There is no font scaler available for a font. Available beginning with ATSUI 1.1.
`kATSUCoordinateOverflowErr`	-8807	The coordinate values passed to the function caused a coordinate overflow (greater than 32 KB). Available beginning with ATSUI 1.1.
`kATSULineBreakInWord`	-8808	The function `ATSUBreakLine` performed a line break within a word. Available beginning with ATSUI 1.2.
`kATSUBusyObjectErr`	-8809	An ATSUI object is being used by another thread. Available in Mac OS X v10.1 and later.
`kATSUInvalidFontFallbacksErr`	-8900	The `ATSUFontFallback` object is not initialized or is otherwise in an in valid state. Available in Mac OS X v10.1 and later.
`kATSUUnsupportedStreamFormatErr`	-8901	The data-flattening format is invalid or is not supported by this version of ATSUI. Available in Mac OS X v10.2 and later.
`kATSUBadStreamErr`	-8902	The data is not formatted as specified by the data-flattening format constant, or the data is corrupt. Available in Mac OS X v10.2 and later.

Result Code	Value	Description
`kATSUOutputBufferTooSmallErr`	-8903	The output buffer is too small to contain the data output by the function. Available in Mac OS X v10.2 and later.
`kATSUInvalidCallInsideCallbackErr`	-8904	Your callback is making a call that could cause an infinite recursion. Available in Mac OS X v10.2 and later.
`kATSULastErr`	-8959	No ATSUI-related result codes may exceed this value. Result code values between `kATSUInvalidTextLayoutErr` and `kATSULastErr` are reserved. Available beginning with ATSUI 1.0.

Gestalt Constants

You can check for version and feature availability information by using the ATSUI attribute and version selectors defined in the Gestalt Manager. For more information see Gestalt Manager Reference.

Deprecated ATSUI Functions

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

Deprecated in Mac OS X v10.0

ATSUCreateTextLayoutWithTextHandle

Creates an opaque text layout object containing default text layout attributes as well as associated text and text styles. (Deprecated in Mac OS X v10.0. Use `ATSUCreateTextLayoutWithTextPtr` (page 42) instead. See the Discussion for more details.)

Not recommended.

```
OSStatus ATSUCreateTextLayoutWithTextHandle (
   UniCharArrayHandle iText,
   UniCharArrayOffset iTextOffset,
   UniCharCount iTextLength,
   UniCharCount iTextTotalLength,
   ItemCount iNumberOfRuns,
   const UniCharCount iRunLengths[],
   ATSUStyle iStyles[],
   ATSUTextLayout *oTextLayout
);
```

Parameters

iText

A handle of type `UniCharArrayHandle` referring to a text buffer containing UTF-16–encoded text. ATSUI associates this buffer with the new text layout object and analyzes the entire text of the buffer when obtaining the layout context for the current text range. Thus, for paragraph-format text, if you specify a buffer containing less than a complete paragraph, some of ATSUI's layout results are not guaranteed to be accurate. For example, with a buffer of less than a full paragraph, ATSUI can neither reliably obtain the context for bidirectional processing nor reliably generate accent attachments and ligature formations for Roman text.

iTextOffset

A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range to include in the layout. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iTextLength` parameter.

iTextLength

A `UniCharCount` value specifying the length of the text range. Note that `iTextOffset + iTextLength` must be less than or equal to the value of the *iTextTotalLength* parameter. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

iTextTotalLength
A `UniCharCount` value specifying the length of the entire text buffer. This value should be greater than or equal to the range of text defined by the `iTextLength` parameter.

iNumberOfRuns
An `ItemCount` value specifying the number of text style runs you want to define within the text range. The number of style objects and style run lengths passed in the `iStyles` and `iRunLengths` parameters, respectively, should each be equal to the number of runs specified here.

iRunLengths
A pointer to a `UniCharCount` array specifying the lengths of each style run in the text layout object. You can pass `kATSUToTextEnd` for the last style run length if you want the style run to extend to the end of the text range. If the sum of the style run lengths is less than the total length of the text range, the remaining characters are assigned to the last style run.

iStyles
A pointer to the first element in an `ATSUStyle` array. Each element in the array must contain a valid style object that corresponds to a style run defined by the `iRunLengths` array.

oTextLayout
A valid pointer to an `ATSUTextLayout` value. On return, the value refers to the newly created text layout object.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should use the function `ATSUCreateTextLayoutWithTextPtr` (page 42) instead of using the function `ATSUCreateTextLayoutWithTextHandle`.

The `ATSUCreateTextLayoutWithTextHandle` function creates a text layout object associated with style objects and text and containing the default text layout attributes described in "Attribute Tags" (page 196). To provide nondefault line or layout attributes for a text layout object, you can call the functions `ATSUSetLineControls` (page 124) or `ATSUSetLayoutControls` (page 122). After setting text attributes, call `ATSUDrawText` (page 50) to draw the text.

Because the only way that ATSUI interacts with text is via the memory references you associate with a text layout object, you are responsible for keeping these references updated, as in the following cases:

1. When the user deletes or inserts a subrange within a text buffer (but the buffer itself is not relocated), you should call the functions `ATSUTextDeleted` (page 135) and `ATSUTextInserted` (page 136), respectively.

2. When you relocate the entire text buffer (but no other changes have occurred that would affect the buffer's current subrange), you should call the function `ATSUTextMoved` (page 137).

3. When both the buffer itself is relocated and a subrange of the buffer's text is deleted or inserted (that is, a combination of cases 1 and 2, above), you must use either the function `ATSUSetTextHandleLocation` (page 241) or the function `ATSUSetTextPointerLocation` (page 131) to inform ATSUI.

4. When you are associating an entirely different buffer with a text layout object, you must call either the function `ATSUSetTextHandleLocation` (page 241) or the function `ATSUSetTextPointerLocation` (page 131).

Note that, because ATSUI objects retain state information, doing superfluous calling can degrade performance. For example, you could call `ATSUSetTextHandleLocation` rather than `ATSUTextInserted` when the user inserts text, but there would be a performance penalty, as all the layout caches are flushed when you call `ATSUSetTextHandleLocation`, rather than just the affected ones.

Text layout objects are readily reusable and should themselves be cached for later use, if possible.

The `ATSUCreateTextLayoutWithTextHandle` function associates text with a text layout object via a handle, but ATSUI functions that need to access the text return the handle to its original state upon completion.

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

Deprecated in Mac OS X v10.0.

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUIdle

Performs background processing. (Deprecated in Mac OS X v10.0. There is no replacement because this function does nothing in Mac OS X.)

Not recommended.

```
OSStatus ATSUIdle (
   ATSUTextLayout iTextLayout
);
```

Parameters

`iTextLayout`
A reference to the text layout object in which you want ATSUI to perform background processing.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The function `ATSUIdle` is not recommended. Current versions of ATSUI do not implement background processing for text layout objects. In Mac OS X, the function `ATSUIdle` does nothing.

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

Deprecated in Mac OS X v10.0.

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUSetTextHandleLocation

Associates text with a text layout object. (Deprecated in Mac OS X v10.0. Use `ATSUSetTextPointerLocation` (page 131) instead. See the Discussion for more details.)

Not recommended.

```
OSStatus ATSUSetTextHandleLocation (
   ATSUTextLayout iTextLayout,
   UniCharArrayHandle iText,
   UniCharArrayOffset iTextOffset,
   UniCharCount iTextLength,
   UniCharCount iTextTotalLength
);
```

Parameters

iTextLayout
An `ATSUTextLayout` value specifying the text layout object with which to associate text.

iText
A handle of type `UniCharArrayHandle`, referring to a text buffer containing UTF-16–encoded text. ATSUI associates this buffer with the text layout object and analyzes the complete text of the buffer when obtaining the layout context for the current text range. Thus, for paragraph-format text, if you specify a buffer containing less than a complete paragraph, some of ATSUI's layout results are not guaranteed to be accurate. For example, with a buffer of less than a full paragraph, ATSUI can neither reliably obtain the context for bidirectional processing nor reliably generate accent attachments and ligature formations for Roman text.

iTextOffset
A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the range to include in the layout. To indicate that the specified text range starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`,. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iTextLength` parameter.

iTextLength
A `UniCharCount` value specifying the length of the text range. Note that `iTextOffset + iTextLength` must be less than or equal to the value of the *iTextTotalLength* parameter. If you want the range of text to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

iTextTotalLength
A `UniCharCount` value specifying the length of the entire text buffer. This value should be greater than or equal to the range of text defined by the `iTextLength` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should use the function `ATSUSetTextPointerLocation` (page 131) instead of the function `ATSUSetTextHandleLocation`.

For ATSUI to render your text, you must associate the text with both a text layout object and style information. Some functions, such as `ATSUCreateTextLayoutWithTextPtr` (page 42), create a text layout object and associate text with it concurrently. However, if you use the function `ATSUCreateTextLayout` (page 41) to create a layout object, you must assign text to the text layout object prior to attempting most ATSUI operations.

You can use either of the functions `ATSUSetTextHandleLocation` or `ATSUSetTextPointerLocation` (page 131) to associate text with a text layout object. When you call these functions, you are both assigning a text buffer to a text layout object and specifying the current text subrange within the buffer to include in the layout.

If there is already text associated with a text layout object, calling `ATSUSetTextHandleLocation` or `ATSUSetTextPointerLocation` overrides the previously associated text, as well as clearing the object's layout caches. You would typically only call these functions for a text layout object with existing associated text if either (a) both the buffer itself is relocated and a subrange of the buffer's text is deleted or inserted or (b) when associating an entirely different buffer with a text layout object.

Note that, because ATSUI objects retain state, doing superfluous calling can degrade performance. For example, you could call `ATSUSetTextHandleLocation` rather than `ATSUTextInserted` (page 136) when the user simply inserts a subrange of text within a text buffer, but there would be a performance penalty, as all the layout caches are flushed by `ATSUSetTextHandleLocation`, rather than just the affected ones.

Similarly, you should not call `ATSUSetTextHandleLocation`, when an entire text buffer associated with a text layout object is relocated, but no other changes have occurred that would affect the buffer's current subrange. Instead, you should call `ATSUTextMoved` (page 137), which is a more focused function and therefore more efficient.

After associating text with the text layout object, use `ATSUSetRunStyle` (page 127) to associate style information with the text. You can then call the function `ATSUDrawText` (page 50) to display the text.

Note that while `ATSUSetTextHandleLocation` associates text with a text layout object via a handle, ATSUI functions that need to access the text return the handle to its original state upon function completion.

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

Deprecated in Mac OS X v10.0.

Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

Deprecated in Mac OS X v10.1

ATSUCopyToHandle

Copies an ATSUI style to a handle. (Deprecated in Mac OS X v10.1. Use `ATSUFlattenStyleRunsToStream` (page 55) instead.)

Not Recommended

```
OSStatus ATSUCopyToHandle (
   ATSUStyle iStyle,
   Handle oStyleHandle
);
```

Parameters

`iStyle`
An `ATSUStyle` value.

`oStyleHandle`
A valid handle.

Return Value
A result code.

Discussion

The `ATSUCopyToHandle` function is not recommended for use, as this function does not produce the correct data format for the display of ATSUI style data. You should instead use the function `ATSUFlattenStyleRunsToStream` to flatten style data and the function `ATSUUnflattenStyleRunsFromStream` to unflatten style data. These functions read and write data using the `ustl` data specification. You can use a data block format of this type to copy and paste Unicode-encoded styled text between applications or within your application. The `ustl` data structure contains flattened text layout data, flattened style run data, and flattened style list data. For more information on the `ustl` data structure see *Inside Mac OS X: ATSUI Reference*.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.1.

Not available to 64-bit applications.

Declared In

`ATSUnicodeFlattening.h`

Deprecated in Mac OS X v10.3

ATSUDrawGlyphInfo

Draws glyphs at the specified location, based on style and layout information specified for each glyph. (Deprecated in Mac OS X v10.3. Use functions from "Accessing Glyph Data" (page 18) instead.)

Not recommended.

```
OSStatus ATSUDrawGlyphInfo (
   ATSUGlyphInfoArray *iGlyphInfoArray,
   Float32Point iLocation
);
```

Parameters

`iGlyphInfoArray`

A pointer to an `ATSUGlyphInfoArray` structure containing the glyph information to draw. You can obtain an `ATSUGlyphInfoArray` structure from the function `ATSUGetGlyphInfo` (page 246).

`iLocation`

A `Float32Point` data structure that contains the x and y coordinates at which to draw the glyph(s). Each coordinate in the `Float32Point` data structure is a `Float32` value.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

You must use `ATSUDrawGlyphInfo` to draw glyphs if you have previously called the function `ATSUGetGlyphInfo` (page 246), and you have modified the glyph information. However, if you want to modify the glyph information you should use the functions `ATSUGlyphGetQuadraticPaths` (page 98) or `ATSUGlyphGetCubicPaths` (page 95) instead of calling the function `ATSUGetGlyphInfo`.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.3.

Not available to 64-bit applications.

Declared In
`ATSUnicodeGlyphs.h`

ATSUGetFontFallbacks

Obtains the global font list and search order that ATSUI uses when a font does not have the glyph needed to image a character. (Deprecated in Mac OS X v10.3. Use font fallback objects instead.)

Not recommended.

```
OSStatus ATSUGetFontFallbacks (
   ItemCount iMaxFontFallbacksCount,
   ATSUFontID oFontIDs[],
   ATSUFontFallbackMethod *oFontFallbackMethod,
   ItemCount *oActualFallbacksCount
);
```

Parameters

iMaxFontFallbacksCount

An `ItemCount` value specifying the maximum number of fonts that you want to obtain. Typically, this is equivalent to the size of the array allocated in the `oFontIDs` parameter. To determine this value, see the Discussion.

oFontIDs

A pointer to memory you have allocated for an array of `ATSUFontID` values. If you are uncertain of how much memory to allocate, see the Discussion. On return, the array contains font IDs identifying the fonts ATSUI searches when seeking a substitute font.

oFontFallbackMethod

A pointer to an `ATSUFontFallbackMethod` value. On return, the value identifies the order in which ATSUI searches fonts. See "Font Fallback Methods" (page 214) for a description of possible values.

oActualFallbacksCount

A pointer to an `ItemCount` value. On return, the value specifies the actual number of fonts that ATSUI searches. This value may be greater than that passed in the `iMaxFontFallbacksCount` parameter.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
You should not use this function because it operates on a global scope and may not be available in future versions of ATSUI. You should instead use the function `ATSUGetObjFontFallbacks` (page 85) with a font fallback object that has been associated with a text layout object. See *Inside Mac OS X: Rendering Unicode Text With ATSUI* for step-by-step instructions on creating a font fallback object and associating it with a text layout object.

Special Considerations

Global font fallback settings can be changed by any ATSUI client, so they can be changed unexpectedly. The only way to ensure that ATSUI uses your preferred font fallback settings for your text is to create a font fallback object and associated it with a text layout object. See the Discussion for more details.

Version Notes
Available beginning with ATSUI 1.1.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.3.
Not available to 64-bit applications.

Declared In
`ATSUnicodeObjects.h`

ATSUGetGlyphInfo

Obtains a copy of the style and layout information for each glyph in a line. (Deprecated in Mac OS X v10.3. Use functions from "Accessing Glyph Data" (page 18) instead.)

Not recommended.

```
OSStatus ATSUGetGlyphInfo (
   ATSUTextLayout iTextLayout,
   UniCharArrayOffset iLineStart,
   UniCharCount iLineLength,
   ByteCount *ioBufferSize,
   ATSUGlyphInfoArray *oGlyphInfoPtr
);
```

Parameters

`iTextLayout`
An `ATSUTextLayout` value specifying the text layout object to examine.

`iLineStart`
A `UniCharArrayOffset` value specifying the offset from the beginning of the text buffer to the first character of the line to examine. To indicate that the line starts at the beginning of the text buffer, you can pass the constant `kATSUFromTextBeginning`. To specify the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iLineLength` parameter.

`iLineLength`
A `UniCharCount` value specifying the length of the line. If you want the line to extend to the end of the text buffer, you can pass the constant `kATSUToTextEnd`.

`ioBufferSize`
A pointer to a `ByteCount` value specifying the size of the buffer you have allocated for the `ATSUGlyphInfoArray` structure produced in the *`oGlyphInfoPtr`* parameter. On return, the value specifies the actual size of the `ATSUGlyphInfoArray` structure.

`oGlyphInfoPtr`
A pointer to an `ATSUGlyphInfoArray` structure. On return, the structure contains values identifying the text layout object, the number of glyphs in the specified line, and an array of `ATSUGlyphInfo` structures for each of the glyphs. Each `ATSUGlyphInfo` structure contains information identifying the glyph, the style object with which it is associated, and other related layout values.

Return Value
A result code. See "ATSUI Result Codes" (page 234).

Discussion
The `ATSUGetGlyphInfo` function obtains a copy of the style and layout information for each glyph in a line of text. Copying can be slow, so it's best to use this function only if you do not plan to modify the glyph information. If you do modify the glyph information, you can only draw the modified glyphs by calling the

function ATSUDrawGlyphInfo (page 244). Because you are working with a copy of the glyph data and not the actual data that ATSUI has, if you try to draw text by calling the ATSUDrawText (page 50) function, none of the changes you make to the glyph information will be reflected in the drawn text.

Note that is you obtain glyph information with the function ATSUGetGlyphInfo and then draw glyphs using ATSUDrawGlyphInfo, ATSUI does not take synthetic styles into account when it draw. This means that font substitution will not work.

If you want to modify glyph information you should instead use the ATSUI direct-access functions ATSUGlyphGetQuadraticPaths (page 98) or ATSUGlyphGetCubicPaths (page 95). You use each of these functions along with callback functions you supply for drawing the glyphs. When you modify and draw glyphs using ATSUI's direct-access functions, you obtain access to the same information as that supplied by the function ATSUGetGlyphInfo, but in a way that allows font substitution to work. For more information on retrieving and drawing glyph outlines, see *Inside Mac OS X: Rendering Unicode Text With ATSUI*.

The Unicode characters in the text layout object (ATSUTextLayout) and the glyphs returned by the function ATSUGetGlyphInfo do not necessarily have a one-to-one correspondence. For example, the accented Latin character é can be represented by an e with a combining ´ accent. In this case, two characters map to one glyph.

Common ligatures such as ﬁ also form automatically for some fonts, causing two characters to map to one glyph. Right-to-left scripts such as Arabic, and complex scripts such as Devanagari or Thai have even more complicated mappings from characters to glyphs.

For this reason it's best to use the high level ATSUI functions whenever possible, and to associate a paragraph of text with a text layout object. Your application is then completely insulated from such issues.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.3.
Not available to 64-bit applications.

Declared In
ATSUnicodeGlyphs.h

ATSUMeasureText

(Deprecated in Mac OS X v10.3. Use ATSUGetUnjustifiedBounds (page 93) instead.)

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.3.
Not available to 64-bit applications.

Declared In
ATSUnicodeDrawing.h

ATSUSetFontFallbacks

Sets, on a global scope, the font list and search order for ATSUI to use when a font does not have the glyph needed to image a character. (Deprecated in Mac OS X v10.3. Use font fallback objects instead.)

Not recommended.

```
OSStatus ATSUSetFontFallbacks (
   ItemCount iFontFallbacksCount,
   const ATSUFontID iFontIDs[],
   ATSUFontFallbackMethod iFontFallbackMethod
);
```

Parameters

`iFontFallbacksCount`

An `ItemCount` value specifying the number of fonts to be searched. This value should be equivalent to the number of elements in the `iFontIDs` array.

`iFontIDs`

A pointer to the first `ATSUFontID` value in the array of fonts to be searched.

`iFontFallbackMethod`

An `ATSUFontFallbackMethod` value specifying the order in which ATSUI is to search the fonts. See "Font Fallback Methods" (page 214) for a description of possible search orders.

Return Value

A result code. See "ATSUI Result Codes" (page 234).

Discussion

When you call `ATSUSetFontFallbacks`, any settings you apply are global to the process and are used by all ATSUI clients in the process. Therefore, any ATSUI clients in the process can change these global font fallback settings unexpectedly. Other application threads can modify the font fallbacks settings, as well. The only way to ensure that ATSUI uses your preferred font fallback settings for your text is to create a font fallback object and associated it with a text layout object.

You create a font fallback object by calling the function `ATSUCreateFontFallbacks` (page 39). You define settings for the object by calling the function `ATSUSetObjFontFallbacks` (page 126). To associate the font fallback object with a text layout object, call either of the functions `ATSUSetLayoutControls` (page 122) or `ATSUSetLineControls` (page 124). See *Inside Mac OS X: Rendering Unicode Text With ATSUI* for step-by-step instructions on creating a font fallback object and associating it with a text layout object.

Special Considerations

You should not use this function because it operates on a global scope and may not be available in future versions of ATSUI. Instead, use font fallback objects as described in the Discussion.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.3.

Not available to 64-bit applications.

Declared In

`ATSUnicodeObjects.h`

Document Revision History

This table describes the changes to *ATSUI Reference*.

Date	Notes
2007-06-28	Corrected several technical and typographical errors.
	Removed information that incorrectly stated that you can set custom layout and line attributes. You can custom attributes only for styles.
	Added clarification to the use of `CGContextRef` drawing destination in the function `ATSUHighLightText`.
2006-07-31	Added deprecation infomation.
2005-11-09	Minor techincal fix.
2005-08-11	Added information to the function ATSUFindFontFromName and made a few minor technical corrections.
2005-07-07	Fixed typographical errors.
2003-10-15	Added framework and header file information to the introduction. The ATSUI API is now defined in multiple header files.
	Updated header file information for each function.
	Added two constants for decimal tab support—`kATSUDecimalTab` and `kATSULineDecimalTabCharacterTag`.
	Changed the name of the "Text Offset and Length Constants" group to "Text Buffer Convenience Constants" (page 233) to reflect the addition of two constants used for bidirectional support.
	Added a constant to support highlighting using a `CGColor` type—`kATSULineHighlightCGColorTag`.
	Added several new style attributes. See "Attribute Tags" (page 196).
	Added a new enumeration—"Style Line Count Types" (page 230)—to specify line counts for underline and strike through styles.
	Added information to the functions `ATSURightwardCursorPosition` (page 118) and `ATSULeftwardCursorPosition` (page 103) concerning new constants for bidirectional support. See "Text Buffer Convenience Constants" (page 233) for a description of the new constants.

Date	Notes
	Added a new line layout option for line breaking—`kATSLineBreakToNearestCharacter`.
	Improved wording for the function `ATSUFindFontFromName` (page 52).
	Fixed typographical errors.
2003-06-12	Updated availability information for the function `ATSUBatchBreakLines`. Improved wording in the discussion of the `ATSUGetSoftLineBreaks` function.
2003-04-07	Removed the ATSUI gestalt constants. All gestalt constants are now documented in *Inside Mac OS X: Gestalt Manager Reference*.
	Corrected information about the result code returned from the function `ATSUGlyphGetCurvePaths`.
2002-10-28	Added header information for each function.
	Added additional information to the functions `ATSUFlattenStyleRunsFromStream` and `ATSUUnflattenStyleRunsFromStream`.
	Moved list of deprecated functions to Appendix A.
	Fixed typographical errors.
2002-09-10	Updated for ATSUI version 2.4. This document includes a complete revision of the existing documentation.
	Added documentation for flattening and unflattening functions and data types. These allow you to save and retrieve ATSUI style data in a flattened format. The format for flattening (`'ustl'`) is also updated.
	Added documentation for direct-access functions and data types. These allow you to access glyph data directly.
	Added documentation for functions and data types added for Mac OS X version 10.2, including new line-layout options, style rendering options, glyph information flags, a layout operation callback, support for tabs, and batch line breaking. You'll find new attribute tags for text measurement (ascent, descent, leading, color (includes alpha channel), glyph selection, and font transformation.
	Added documentation for font fallback objects.
	Added documentation for new gestalt constants.
	Revised existing ATSUI result codes and added new ones.
	Updated Carbon support status of functions. Moved deprecated functions and data types to *Legacy ATSUI Reference*.
	Replaced the function `ATSUMeasureText` with `ATSUGetUnjustifiedBounds`.

Document Revision History

Date	Notes
2001-07-17	Updated for ATSUI version 1.2.

Index