INSIDE MACINTOSH

Apple Type Services for Unicode Imaging Reference

For ATSUI 1.1

May 7, 1999
Technical Publications

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

Simultaneously published in the United States and Canada.

Contents

Introduction

Apple Type Services for Unicode Imaging (ATSUI) provides you with the ability to render Unicode-encoded text using many of the advanced typography features provided with QuickDraw GX. It automatically handles many of the complexities inherent in text layout, including how to correctly render text in bidirectional and vertical script systems.

This preliminary document describes the ATSUI programming interface through ATSUI 1.1. New features in ATSUI 1.1 include the ability to track fonts, control memory allocation, specify fallback fonts, obtain typographic glyph bounds, clear the layout cache of a line or entire text layout object, and manipulate line attributes.

This document should be useful to anyone who wants to write a new text editor or word processing application that renders Unicode-encoded text. You can also use it to modify existing applications to support the rendering of Unicode-encoded text.

If you want more information on how ATSUI interacts with fonts, you can see the Apple font web site:

<http://fonts.apple.com/>

This document describes the ATSUI API in the following chapters:

- "ATSUI Reference" (page 11) describes the complete API through ATSUI 1.1, including functions, data types, constants, and result codes.
- "Document Revision History" (page 261) provides a history of changes to this document.
- "Functions New to ATSUI 1.1 and Changed From ATSUI 1.0" (page 263) alphabetically lists all ATSUI 1.1 functions and any ATSUI 1.0 functions whose implementation has changed with ATSUI 1.1.
- "Glossary" (page 297) provides an alphabetical listing of typographic and ATSUI-specific terms.

ATSUI Reference

This chapter describes the ATSUI programming interface through ATSUI 1.1. It is broken down into the following sections:

- "Gestalt Selectors" (page 12) describes the selectors you can use to determine ATSUI's availability.
- "Functions for Manipulating Style Objects" (page 14) describes the functions you can use to manipulate a style object and its contents.
- "Functions for Obtaining Font Data" (page 47) describes the functions you can use to obtain font data.
- "Functions for Manipulating Text Layout Objects" (page 78) describes the functions you can use to manipulate a text layout object and its contents.
- "Functions for Responding to User Actions" (page 126) describes the functions you can use to respond to actions like text insertion and deletion, cursor movement, line breaking, and highlighting.
- "Functions for Manipulating Memory Settings" (page 174) describes the functions you can use to manipulate memory settings in ATSUI.
- "Application-Defined Functions for Controlling Memory Allocation" (page 178) describes the functions you can provide if you wish to exercise complete control over memory allocation operations in ATSUI.
- "Data Types" (page 182) describes the data types and structures you use with ATSUI functions.
- "Resource" (page 199) describes the `'ustl'` clipboard data block format that you can use to describe styled text in the clipboard.
- "Constants" (page 203) describes the constants defined by ATSUI for your application's use.
- "Result Codes" (page 256) describes the result codes returned by ATSUI functions.

Gestalt Selectors

Before calling any functions dependent upon ATSUI, your application should pass the `gestaltATSUVersion` selector to the `Gestalt` function to determine the version of ATSUI installed on the user's system. You can also determine version information by testing for the feature bits described below.

```
enum {
    gestaltATSUVersion          = 'uisv',
};
```

Constant descriptions

`gestaltATSUVersion` The `Gestalt` selector you pass to determine the version of ATSUI installed on the user's system. For a description of the currently-defined bits, see below.

If you pass the `gestaltATSUVersion` selector, on return the `Gestalt` function passes back a `Fixed` value that represents the version of ATSUI installed on the user's system.

```
enum {
    gestaltOriginalATSUVersion  = 1.0,
    gestaltATSUUpdate1          = 2.0
};
```

Constant descriptions

`gestaltOriginalATSUVersion`
A `Fixed` value that indicates that version 1.0 of ATSUI is installed on the user's system.

`gestaltATSUUpdate1`
A `Fixed` value that indicates that version 1.1 of ATSUI is installed on the user's system.

You pass the following `Gestalt` selector to determine which features of ATSUI are available.

```
enum {
     gestaltATSUFeatures        = 'uisf',
};
```

Constant descriptions

`gestaltATSUFeatures`
: A `Gestalt` selector. For a description of the currently-defined bits, see the discussion below.

On return, the `Gestalt` function passes back a 32-bit value which you can use to test to determine the available features.

If ATSUI 1.1 is installed on the user's system, the bits specified by the constants `gestaltATSUTrackingFeature`, `gestaltATSUMemoryFeature`, `gestaltATSUFallbacksFeature`, `gestaltATSUGlyphBoundsFeature`, `gestaltATSULineControlFeature`, `gestaltATSULayoutCacheClearFeature`, and `gestaltATSULayoutCreateAndCopyFeature` will be set.

```
enum {
    gestaltATSUTrackingFeature              = 0x00000001,
    gestaltATSUMemoryFeature                = 0x00000001,
    gestaltATSUFallbacksFeature             = 0x00000001,
    gestaltATSUGlyphBoundsFeature           = 0x00000001,
    gestaltATSULineControlFeature           = 0x00000001,
    gestaltATSULayoutCacheClearFeature      = 0x00000001,
    gestaltATSULayoutCreateAndCopyFeature   = 0x00000001
};
```

Constant descriptions

`gestaltATSUTrackingFeature`
: If this bit is set, you can obtain a font tracking setting and name code using the functions `ATSUCountFontTracking` (page 61) and `ATSUGetIndFontTracking` (page 62).

`gestaltATSUMemoryFeature`
: If this bit is set, you can control ATSUI's memory allocation using the functions `ATSUCreateMemorySetting` (page 174), `ATSUSetCurrentMemorySetting` (page 176), `ATSUGetCurrentMemorySetting` (page 176), and `ATSUDisposeMemorySetting` (page 177).

`gestaltATSUFallbacksFeature`
: If this bit is set, you can set and obtain fallback font search methods using the functions `ATSUSetFontFallbacks` (page 119) and `ATSUGetFontFallbacks` (page 120).

`gestaltATSUGlyphBoundsFeature`
: If this bit is set, you can obtain typographic glyph bounds using the function `ATSUGetGlyphBounds` (page 145).

`gestaltATSULineControlFeature`
: If this bit is set, you can manipulate line attributes using the functions `ATSUCopyLineControls` (page 98), `ATSUSetLineControls` (page 100), `ATSUGetLineControl` (page 102), `ATSUGetAllLineControls` (page 104), and `ATSUClearLineControls` (page 105).

`gestaltATSULayoutCacheClearFeature`
: If this bit is set, you can flush the layout cache of a line or entire text layout object using the function `ATSUClearLayoutCache` (page 88).

`gestaltATSULayoutCreateAndCopyFeature`
: If this bit is set, you can create a copy of a text layout object using the function `ATSUCreateAndCopyTextLayout` (page 85).

VERSION NOTES

The selector constants `gestaltATSUUpdate1`, `gestaltATSUTrackingFeature`, `gestaltATSUMemoryFeature`, `gestaltATSUFallbacksFeature`, `gestaltATSUGlyphBoundsFeature`, `gestaltATSULineControlFeature`, `gestaltATSULayoutCacheClearFeature`, and `gestaltATSULayoutCreateAndCopy` are available with ATSUI 1.1. All other selector constants are available with ATSUI 1.0.

Functions for Manipulating Style Objects

This section describes the functions you can use to to manipulate a style object and its contents:

- "Creating, Clearing, and Disposing of Style Objects" (page 15)
- "Copying Style Contents" (page 22)
- "Flattening and Unflattening Style Objects" (page 26)
- "Manipulating Style Run Attributes" (page 29)
- "Manipulating Font Features" (page 36)

- "Manipulating Font Variations in a Style Object" (page 42)

Creating, Clearing, and Disposing of Style Objects

ATSUI provides the following functions for creating, clearing, and disposing of style objects:

- `ATSUCreateStyle` (page 15) creates a style object.
- `ATSUCreateAndCopyStyle` (page 16) creates a copy of a style object.
- `ATSUCompareStyles` (page 18) compares the contents of two style objects.
- `ATSUSetStyleRefCon` (page 19) sets application-specific style data.
- `ATSUGetStyleRefCon` (page 19) obtains application-specific style data.
- `ATSUStyleIsEmpty` (page 20) indicates whether a style object contains any previously set style run attribute, font feature, or font variation values.
- `ATSUClearStyle` (page 21) removes all previously set style run attribute, font feature, and font variation values from a style object.
- `ATSUDisposeStyle` (page 22) disposes of the memory associated with a style object.

ATSUCreateStyle

Creates a style object.

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

`oStyle` A pointer to a reference of type `ATSUStyle` (page 195). On return, `oStyle` refers to the newly-created style object. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The newly-created style object contains the default font feature and variation values defined by the font and the default style attribute values listed in Table 2-1 (page 237).

You can set style run attribute, font feature, and font variations, respectively, in this style object by calling the functions `ATSUSetAttributes` (page 29), `ATSUSetFontFeatures` (page 37), and `ATSUSetVariations` (page 42).

Once you have set these values, you can can create a copy of this style object (essentially, create a clone) by calling the function `ATSUCreateAndCopyStyle` (page 16). You can also copy some portion of the contents of this style object into another style object by calling the functions `ATSUCopyAttributes` (page 23), `ATSUOverwriteAttributes` (page 24), and `ATSUUnderwriteAttributes` (page 25).

SPECIAL CONSIDERATIONS

`ATSUCreateStyle` allocates memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCreateAndCopyStyle` (page 16)

ATSUCreateAndCopyStyle

Creates a copy of a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose contents you want to copy. You cannot pass `NULL` for this parameter.

`oStyle`	A pointer to a reference of type `ATSUStyle` (page 195). On return, the reference points to a style object containing the same style run attribute, font feature, and font variation values as the style object you passed in the `iStyle` parameter. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCreateAndCopyStyle` function creates a clone of a style object containing the same style run attribute, font feature, and font variation values. `ATSUCreateAndCopyStyle` does not copy reference constants.

If you wish to copy the entire contents of a style object into an existing style object, you should call the function `ATSUCopyAttributes` (page 23). To copy the previously set values from a style object into an existing style object, you should call the function `ATSUOverwriteAttributes` (page 24). If you instead wish to copy values previously set in the source and unset in the destination style object into the destination style object, you should call the function `ATSUUnderwriteAttributes` (page 25).

SPECIAL CONSIDERATIONS

`ATSUCreateAndCopyStyle` allocates memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCreateStyle` (page 15)

ATSUCompareStyles

Compares the contents of two style objects.

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

`iFirstStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose contents you want to compare. You cannot pass `NULL` for this parameter.

`iSecondStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose contents you want to compare. You cannot pass `NULL` for this parameter.

`oComparison` A pointer to a value of type `ATSUStyleComparison`. See "Style Comparison Constants" (page 236) for a description of possible values. On return, the value indicates whether the contents of the two style objects are the same, different, or a subset of eachother. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCompareStyles` function determines whether the style run attribute, font feature, and font variation values set in two style objects are the same, different, or a subset of eachother. `ATSUCompareStyles` does not consider reference constants or application-defined style run attributes in the comparison.

You should call `ATSUCompareStyles` to implement style sheets and tables of style runs.

VERSION NOTES

Available with ATSUI 1.0.

ATSUSetStyleRefCon

Sets application-specific style data.

```
OSStatus ATSUSetStyleRefCon (
                    ATSUStyle iStyle,
                    UInt32 iRefCon);
```

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose application-specific data you want to set. You cannot pass `NULL` for this parameter.

`iRefCon` A 32-bit value, pointer, or handle to application-specific style data.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Note that when you copy or clear a style object that contains a reference constant, the reference constant will not be copied or removed. When you dispose of a style object that contains a reference constant, you are resposible for freeing any memory allocated for the reference constant. Calling `ATSUDisposeStyle` (page 22) will not do so.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetStyleRefCon` (page 19)

ATSUGetStyleRefCon

Obtains application-specific style data.

```
OSStatus ATSUGetStyleRefCon (
                    ATSUStyle iStyle,
                    UInt32 *oRefCon);
```

`iStyle`	A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose application-specific data you want to obtain. You cannot pass `NULL` for this parameter.
`oRefCon`	A pointer to a 32-bit value, pointer, or handle to application-specific style data.You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetStyleRefCon` (page 19)

ATSUStyleIsEmpty

Indicates whether a style object contains any previously set style run attribute, font feature, or font variation values.

```
OSStatus ATSUStyleIsEmpty (
                    ATSUStyle iStyle,
                    Boolean *oIsClear);
```

`iStyle`	A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose settings status you want to determine. You cannot pass `NULL` for this parameter.
`oIsClear`	A pointer to a value of type `Boolean`. On return, the value indicates whether the style contains any previously set style run attribute, font feature, or font variation values. If `true`, the style object contains only default values. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUStyleIsEmpty` function does not consider reference constants when evaluting the contents of a style object.

VERSION NOTES

Available with ATSUI 1.0.

ATSUClearStyle

Removes all previously set style run attribute, font feature, and font variation values from a style object.

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

`iStyle`	A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose contents you want to clear. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUClearStyle` removes all previously set style run attribute, font feature, and font variation values from a style object, including application-defined attribute values. It does not remove reference constants.

If you want to remove selected style run attribute, font feature, or font variation values from a style object, call the functions `ATSUClearAttributes` (page 34), `ATSUClearFontFeatures` (page 41), and `ATSUClearFontVariations` (page 46), respectively.

VERSION NOTES

Available with ATSUI 1.0.

ATSUDisposeStyle

Disposes of the memory associated with a style object.

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

`iStyle`	A reference of type `ATSUStyle` (page 195). Pass a reference to the style object whose memory you want to dispose. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUDisposeStyle` function to dispose of the memory associated with a style object, including style run attribute settings. `ATSUDisposeStyle` only frees memory associated with the style object and its internal structures. It does not dispose of the memory pointed to by custom style run attributes and reference constants. You are ultimately responsible for doing so.

VERSION NOTES

Available with ATSUI 1.0.

Copying Style Contents

ATSUI provides the following functions for copying style contents:

- `ATSUCopyAttributes` (page 23) copies both set and unset values from the source into the destination style object.
- `ATSUOverwriteAttributes` (page 24) copies previously set values from the source into the destination style object.
- `ATSUUnderwriteAttributes` (page 25) copies values previously set in the source and unset in the destination style object into the destination style object.

ATSUCopyAttributes

Copies both set and unset values from the source into the destination style object.

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

`iSourceStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose set and unset style run attribute, font feature, and font variation values you want to copy. You cannot pass `NULL` for this parameter.

`iDestinationStyle`
A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose style run attribute, font feature, and font variation values you want to replace. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCopyAttributes` function differs from the `ATSUOverwriteAttributes` function in that it copies both previously set and unset style run attribute, font feature, and font variation values from the source into the destination style object.

`ATSUCopyAttributes` will not copy the contents of memory referenced by pointers or handles within custom style run attributes or within reference constants. It is your responsibility to ensure that this memory remains valid until the source style object is disposed of.

SPECIAL CONSIDERATIONS

`ATSUCopyAttributes` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

ATSUOverwriteAttributes (page 24)

ATSUUnderwriteAttributes (page 25)

ATSUOverwriteAttributes

Copies previously set values from the source into the destination style object.

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

`iSourceStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose previously set style run attribute, font feature, and font variation values you want to copy. You cannot pass `NULL` for this parameter.

`iDestinationStyle`
A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose corresponding values you want to replace. Only the previously set style run attribute, font feature, and font variation values in the source style object will be copied into this style object. All other quantities in the destination style object are left unchanged. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUOverwriteAttributes` function differs from the `ATSUCopyAttributes` function in that it only copies previuosly set style run attribute, font feature, and font variation values in a style object. Unlike `ATSUCopyAttributes`, it does not copy unset (or default) values.

`ATSUOverwriteAttributes` will not copy the contents of memory referenced by pointers or handles within custom style run attributes or within reference constants. It is your responsibility to ensure that this memory remains valid until the source style object is disposed of.

SPECIAL CONSIDERATIONS

`ATSUOverwriteAttributes` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCopyAttributes` (page 23)

`ATSUUnderwriteAttributes` (page 25)

ATSUUnderwriteAttributes

Copies values previously set in the source and unset in the destination style object into the destination style object.

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

`iSourceStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose previously set values that are not set in the destination style object you want to copy. You cannot pass `NULL` for this parameter.

`iDestinationStyle`
A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose corresponding unset values you want to replace. Only those style run attribute, font feature, and font variation values that were previously set in the source style object and unset in this style object will be copied into this style object. All other quantities in the destination style object are left unchanged. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUUnderwriteAttributes` function differs from the `ATSUOverwriteAttributes` function in that it only copies those style run attribute, font feature, and font variation values that were previously set in the source style object but unset in the destination style object. If a style run attribute, font feature, or font variation value is set in both the source and destination style object, its value in the destination style object will remain unchanged.

`ATSUUnderwriteAttributes` will not copy the contents of memory referenced by pointers or handles within custom style run attributes or within reference constants. It is your responsibility to ensure that this memory remains valid until the source style object is disposed of.

SPECIAL CONSIDERATIONS

`ATSUUnderwriteAttributes` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCopyAttributes` (page 23)

`ATSUOverwriteAttributes` (page 24)

Flattening and Unflattening Style Objects

NOT RECOMMENDED

ATSUI provides the following functions for flattening and unflattening style objects:

- `ATSUCopyToHandle` (page 27) copies styled Unicode text data from a style object to a handle.

- `ATSUPasteFromHandle` (page 28) pastes styled Unicode text data from a handle to a style object.

ATSUCopyToHandle

NOT RECOMMENDED

Copies styled Unicode text data from a style object to a handle.

```
OSStatus ATSUCopyToHandle (
                    ATSUStyle iStyle,
                    Handle oStyleHandle);
```

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose style information you want to copy. If the style object contains pointers or handles within custom style run attributes or reference constants, the contents of this memory will not be copied into the handle. You cannot pass `NULL` for this parameter.

`oStyleHandle` A value of type `Handle`. On return, the handle contains the address of the flattened style information. It does not include the contents of memory pointed to by pointers or handles within custom style run attributes or reference constants. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCopyToHandle` function does not produce the correct data format for displaying ATSUI style data. You should instead use the clipboard data block format described in `ustl` (page 199) when you want to provide clipboard support or copy and paste styled text between applications or within an application.

SPECIAL CONSIDERATIONS

`ATSUCopyToHandle` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ustl` (page 199)

ATSUPasteFromHandle

NOT RECOMMENDED

Pastes styled Unicode text data from a handle into a style object.

```
OSStatus ATSUPasteFromHandle (
                    ATSUStyle iStyle,
                    Handle iStyleHandle);
```

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose style information is to be copied.

`iStyleHandle` A handle of type `Handle`. Pass a handle containing the address of style information that was produced by calling `ATSUCopyToHandle` (page 27).

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUPasteFromHandle` function does not produce the correct data format for displaying ATSUI style data. You should instead use the clipboard data block format described in `ustl` (page 199) when you want to provide clipboard support or copy and paste styled text between applications or within an application.

SPECIAL CONSIDERATIONS

`ATSUPasteFromHandle` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ustl` (page 199)

Manipulating Style Run Attributes

ATSUI provides the following functions for manipulating style run attributes:

- `ATSUSetAttributes` (page 29) sets style run attribute values for a style object.
- `ATSUGetAttribute` (page 31) obtains a style run attribute value from a style object.
- `ATSUGetAllAttributes` (page 33) obtains all the style run attribute values from a style object.
- `ATSUClearAttributes` (page 34) removes previously set style run attribute values from a style object.
- `ATSUCalculateBaselineDeltas` (page 35) calculates the distances from the baseline with a y-delta of 0 to each of the other baseline types in a style object.

ATSUSetAttributes

Sets style run attribute values for a style object.

```
OSStatus ATSUSetAttributes (
                    ATSUStyle iStyle,
                    ItemCount iAttributeCount,
```

```
                    ATSUAttributeTag iTag[],
                    ByteCount iValueSize[],
                    ATSUAttributeValuePtr iValue[]);
```

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose style run attribute values you want to set. You cannot pass `NULL` for this parameter.

`iAttributeCount`
A value of type `ItemCount` that represents the number of style run attributes you want to set.

`iTag` An array of values of type `ATSUAttributeTag`. See "Style Run Attribute Tags" (page 237) for a description of Apple-defined tag values. Each element in the array must contain a valid tag that corresponds to a style run attribute. Note that if you pass a text layout attribute or an ATSUI-reserved tag value in this parameter, `ATSUSetAttributes` returns the result code `kATSUInvalidAttributeTagErr`. You cannot pass `NULL` for this parameter.

`iValueSize` An array of values of type `ByteCount`. This array contains the sizes (in bytes) of the style run attribute values being set. Note that if you pass an attribute value size that is less than required, `ATSUSetAttributes` returns the result code `kATSUInvalidAttributeSizeErr`. You cannot pass `NULL` for this parameter.

`iValue` An array of pointers of type `ATSUAttributeValuePtr` (page 187). Each pointer in the array must reference a style run attribute value that corresponds to a tag in the `iTag` array, and the value referenced by the pointer must be legal for that tag. Note that if you pass an invalid or undefined value, `ATSUSetAttributes` returns the result code `kATSUInvalidAttributeValueErr`. You cannot pass `NULL` for this parameter.

function result A result code. If there is a function error, `ATSUSetAttributes` will not set any style run attributes. The result code `kATSUNoFontCmapAvailableErr` indicates that no `'CMAP'` table can be accessed or synthesized for the font. The result code `kATSUNoFontScalerAvailableErr` indicates that there is no font scaler available for the font.For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUSetAttributes` function sets multiple style run attribute values simultaneously. Unset attributes retain their default values listed in Table 2-1 (page 237).

SPECIAL CONSIDERATIONS

`ATSUSetAttributes` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetAttribute` (page 31)

`ATSUGetAllAttributes` (page 33)

ATSUGetAttribute

Obtains a style run attribute value from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose style run attribute value you want to obtain. You cannot pass `NULL` for this parameter.

`iTag` A style run attribute tag of type `ATSUAttributeTag`. Pass a valid tag that corresponds to the style run attribute whose value you want to determine. See "Style Run Attribute Tags" (page 237) for a description of Apple-defined tag values. Note that if you pass

a text layout attribute or an ATSUI-reserved tag value in this parameter, `ATSUGetAttribute` returns the result code `kATSUInvalidAttributeTagErr`.

`iMaximumValueSize`
A value of type `ByteCount` that represents the amount of memory allocated for the style run attribute value. You can predetermine this value by first calling `ATSUGetAttribute (iStyle, 0, 0, NULL, &oTagValuePairCount)`. Note that if you pass an attribute value size that is less than required, `ATSUGetAttribute` returns the result code `kATSUInvalidAttributeSizeErr`.

`oValue` A pointer of type `ATSUAttributeValuePtr` (page 187). Before calling `ATSUGetAttribute`, pass a pointer to memory you have allocated for the attribute value or `NULL` if you don't know how big the attribute value will be (as in the case of custom style run attributes). If you pass `NULL`, on return, `ATSUGetAttribute` passes back the attribute value size in the `oActualVariationCount` parameter. If you instead pass a pointer to memory you have allocated for the value, on return, `oValue` points to the style run attribute value. Note that if you did not previously set the attribute value, `ATSUGetAttribute` passes back its default value in this parameter and returns the result code `kATSUNotSetErr`.

`oActualValueSize`
A pointer to a value of type `ByteCount`. On return, `oActualValueSize` points to 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 you wish to obtain, as in the case of custom style run attributes.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetAttributes` (page 29)

`ATSUGetAllAttributes` (page 33)

ATSUGetAllAttributes

Obtains all the style run attribute values from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose style run attribute tags and value sizes you want to obtain. You cannot pass `NULL` for this parameter.

`oAttributeInfoArray`
An array of structures of type `ATSUAttributeInfo` (page 186). On return, each structure in the array contains a tag/value-size pair that corresponds to a particular style run attribute value in the style object. You can predetermine how much memory to allocate for this array by first calling `ATSUGetAllAttributes (iStyle, NULL, 0, &oTagValuePairCount)`.

`iTagValuePairArraySize`
A value of type `ItemCount`. This value represents the maximum number of `ATSUAttributeInfo` (page 186) structures that you want passed back in the `oAttributeInfoArray` array. You can predetermine this value by first calling `ATSUGetAllAttributes (iStyle, NULL, 0, &oTagValuePairCount)`.

`oTagValuePairCount`
A pointer to a value of type `ItemCount`. On return, the value represents the actual number of `ATSUAttributeInfo` structures set in the style object. This may be greater than the value passed in the `iTagValuePairArraySize` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

You can pass the tag and value size data passed back in the `oAttributeInfoArray` parameter to the `ATSUGetAttribute` (page 31) function to obtain the value of a particular style run attribute.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetAttribute` (page 31)

ATSUClearAttributes

Removes previously set style run attribute values from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose previously set style run attributes you want to remove. You cannot pass `NULL` for this parameter.

`iTagCount` A value of type `ItemCount` that represents the number of previously set style run attribute values you want to remove. To remove all previously set style run attribute values, pass the constant `kATSUClearAll`.

`iTag` An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that identifies a style run attribute you want to remove. See "Style Run Attribute Tags" (page 237) for a description of Apple-defined tag values. Note that if you pass a text layout attribute or an ATSUI-reserved tag value in this parameter, `ATSUClearAttributes` returns the result code `kATSUInvalidAttributeTagErr`. If you pass the `kATSUClearAll` constant in the `iTagCount` parameter, the value in this parameter will be ignored.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUClearLineControls` function removes those previously set text layout attribute values that are identified by the tags in the `iTag` array. It sets these values to the default values listed in Table 2-2 (page 251).

To remove all the previously set style run attribute values from a style object, pass the constant `kATSUClearAll` in the `iTagCount` parameter. You can remove unset attribute values from a style object without a function error.

If you want to remove all previously set style run attribute, font feature, and font variation values from a style object, call the `ATSUClearStyle` (page 21) function.

VERSION NOTES

Available with ATSUI 1.0.

ATSUCalculateBaselineDeltas

Calculates the default baseline deltas in a particular style run.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose baseline positions you want to control placement of all the glyphs in a single line or in each line of a text layout object. You cannot pass `NULL` for this parameter.

`iBaselineClass`
A value of type `BslnBaselineClass`. Pass the primary baseline from which to calculate the distance to each of the other baseline types. See "Baseline Type Constants" (page 204) for a description of possible values. Pass the constant `kBSLNNoBaselineOverride` if you want to use the standard baseline value from the current font.

`oBaselineDeltas`
: An array of type `BslnBaselineRecord` (page 196). On return, the array contains the baseline deltas (that is, the distance from a specified baseline to each of the other baseline types in the style object). You cannot pass `NULL` for this parameter.

function result
: A result code. See "Result Codes" (page 256).

DISCUSSION

The function `ATSUCalculateBaselineDeltas` (page 35) determines the default baseline deltas (that is, the distances from a specified baseline to each of other baselines) for a specific style object. These deltas are in turn applied to the cross-stream shifting of glyphs in the style run. The deltas are determined by various style run attributes, including font and text size.

You can use these positions to set the default baseline positions for all glyphs in a text layout object or a single line. In order to determine the deltas that you want to use to control placement of all the glyphs on the line, pass in the style object of the dominant style run in the line.

To set these values in a line or text layout object, pass these deltas and the `kATSULineBaselineValuesTag` tag constant to the functions `ATSUSetLineControls` (page 100) and `ATSUSetLayoutControls` (page 92).

VERSION NOTES

Available with ATSUI 1.0.

Manipulating Font Features

ATSUI provides the following functions for manipulating font features:

- `ATSUSetFontFeatures` (page 37) sets font feature values in a style object.
- `ATSUGetFontFeature` (page 38) obtains a font feature value from a style object.
- `ATSUGetAllFontFeatures` (page 39) obtains all the font feature values from a style object.
- `ATSUClearFontFeatures` (page 41) removes previously set font feature values from a style object.

ATSUSetFontFeatures

Sets font feature values in a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose font feature type and selector values you want to set. You cannot pass `NULL` for this parameter.

`iFeatureCount` A value of type `ItemCount` that represents the number of font feature type and selector values you want to set.

`iType` An array of values of type `ATSUFontFeatureType` (page 191) or one of the feature selectors described in "Font Feature Types and Selectors" (page 271). Each element in the array must contain a valid feature type that corresponds to a feature selector in the `iSelector` parameter. You cannot pass `NULL` for this parameter.

`iSelector` An array of values of type `ATSUFontFeatureSelector` (page 191) or one of the feature selectors described in "Font Feature Types and Selectors" (page 271). Each element in the array must contain a feature selector that corresponds to a feature type in the `iType` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUSetFontFeatures` function sets multiple font feature type and selector values simultaneously. Unset features retain their font-defined default values.

To set a font feature in a style object, you must specify both the feature type (that is, the type of font feature to employ) and the feature selector (that is, the level or style of employment).

If you set contradictory font features, ATSUI will not remove a feature when a contradictory feature is set. You are responsible for maintaining the list of font feature settings and removing contradictory settings when they occur.

The order that `ATSUSetFontFeatures` actually sets font features depends on the font-defined order, not the chronological order in which you set them calling `ATSUSetFontFeatures`.

SPECIAL CONSIDERATIONS

`ATSUSetFontFeatures` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetFontFeature` (page 38)

ATSUGetFontFeature

Obtains a font feature value from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose font feature type and selector you want to obtain. You cannot pass `NULL` for this parameter.

`iFeatureIndex` A value of type `ItemCount` that represents the 0-based index of the font feature type and selector whose value you want to obtain. To predetermine the maximum valid value for this index, subtract one from the value passed back from `ATSUGetAllFontFeatures (iStyle, 0, NULL, NULL, &oActualFeatureCount)`.

`oFeatureType` A pointer to a value of type `ATSUFontFeatureType` (page 191) or to one of the feature types described in "Font Feature Types and Selectors" (page 271). On return, the value represents the indexed feature type value. Note that if the feature type value has not been set, `ATSUGetFontFeature` passes back the font-specified default value and returns the result code `kATSUNotSetErr`.

`oFeatureSelector`
A pointer to a value of type `ATSUFontFeatureSelector` (page 191) or to one of the feature selectors described in "Font Feature Types and Selectors" (page 271). On return, the value represents the indexed feature selector value. Note that if the feature selector value has not been set, `ATSUGetFontFeature` passes back the font-specified default value and returns the result code `kATSUNotSetErr`.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetFontFeatures` (page 37)

ATSUGetAllFontFeatures

Obtains all the font feature values from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose font features you want to obtain. You cannot pass `NULL` for this parameter.

`iMaximumFeatureCount`
A value of type `ItemCount`. This value represents the number of font feature types and selectors you want passed back in the `oFeatureType` and `oFeatureSelector` arrays, respectively. You can predetermine this value by first calling `ATSUGetAllFontFeatures(iStyle, 0, NULL, NULL, &oActualFeatureCount)`.

`oFeatureType` An array of values of type `ATSUFontFeatureType` (page 191) or one of the feature type constants described in "Font Feature Types and Selectors" (page 271). On return, the array contains the font feature types in the style object. You can predetermine how much memory to allocate for this array by first calling `ATSUGetAllFontFeatures(iStyle, 0, NULL, NULL, &oActualFeatureCount)`.

`oFeatureSelector`
An array of values of type `ATSUFontFeatureSelector` (page 191) or one of the feature selector constants described in "Font Feature Types and Selectors" (page 271). On return, the array contains the font feature selectors in the style object. You can predetermine how much memory to allocate for this array by first calling `ATSUGetAllFontFeatures(iStyle, 0, NULL, NULL, &oActualFeatureCount)`.

`oActualFeatureCount`
A pointer to a value of type `ItemCount`. On return, the value represents the actual number of font feature types and selectors set in the style object. This may be greater than the value passed in the `iMaximumFeatureCount` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

ATSUClearFontFeatures

Removes previously set font feature values from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose previously set font features you want to remove. You cannot pass `NULL` for this parameter.

`iFeatureCount` A value of type `ItemCount` that represents the number of previously set font feature values you want to remove. To remove all the previously set font feature values from a style object, pass the constant `kATSUClearAll`.

`iType` An array of values of type `ATSUFontFeatureType` (page 191) or one of the feature type constants described in "Font Feature Types and Selectors" (page 271). Each element in the array must contain a valid feature selector that identifies a font feature attribute you want to remove. If you pass the `kATSUClearAll` constant in the `iFeatureCount` parameter, the value in this parameter will be ignored.

`iSelector` An array of values of type `ATSUFontFeatureSelector` (page 191) or one of the feature selector constants described in "Font Feature Types and Selectors" (page 271). Each element in the array must contain a valid feature selector that identifies a font feature attribute you want to remove. If you pass the `kATSUClearAll` constant in the `iFeatureCount` parameter, the value in this parameter will be ignored.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUClearFontFeatures` function removes those previously set font feature values that are identified by the feature types and selectors in the `iType` and `iSelector` arrays. It sets these values to their font-defined default values.

To remove all the previously set font feature values from a style object, pass the constant `kATSUClearAll` in the `iFeatureCount` parameter. You can remove unset font feature values from a style object without a function error.

If you want to remove all previously set style run attribute, font feature, and font variation values from a style object, call the `ATSUClearStyle` (page 21) function.

VERSION NOTES

Available with ATSUI 1.0.

Manipulating Font Variations in a Style Object

ATSUI provides the following functions for manipulating font variations in a style object:

- `ATSUSetVariations` (page 42) sets font variation values for a style object.
- `ATSUGetFontVariationValue` (page 44) obtains a font variation value from a style object.
- `ATSUGetAllFontVariations` (page 45) obtains all the font variation values from a style object.
- `ATSUClearFontVariations` (page 46) removes previously set font variation values from a style object.

ATSUSetVariations

Sets font variation values for a style object.

```
OSStatus ATSUSetVariations (
                    ATSUStyle iStyle,
                    ItemCount iVariationCount,
                    ATSUFontVariationAxis iAxes[],
                    ATSUFontVariationValue iValue[]);
```

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose font variations you want to set. You cannot pass `NULL` for this parameter.

`iVariationCount`
: A value of type `ItemCount` that represents the number of font variations being set. This should correspond to the number of elements in the `iAxes` and `iValue` arrays.

`iAxes`
: An array of values of type `ATSUFontVariationAxis` (page 192). Each element in the array must contain a valid variation axis that corresponds to a variation value in the `iValue` parameter. You cannot pass `NULL` for this parameter.

`iValue`
: An array of values of type `ATSUFontVariationValue` (page 193). Each element in the array must contain a valid variation value that corresponds to a variation axis in the `iAxes` parameter. You cannot pass `NULL` for this parameter.

function result
: A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUSetVariations` function sets multiple font variation axes and values simultaneously. Unset font variations retain their default values. If the font does not support the passed in variation axes, the variations will have no visual effect.

SPECIAL CONSIDERATIONS

`ATSUSetVariations` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetFontVariationValue` (page 44)

ATSUGetFontVariationValue

Obtains a font variation value from a style object.

```
OSStatus ATSUGetFontVariationValue (
                    ATSUStyle iStyle,
                    ATSUFontVariationAxis iATSUFontVariationAxis,
                    ATSUFontVariationValue *oATSUFontVariationValue);
```

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose font variation axis value you want to obtain. You cannot pass `NULL` for this parameter.

`iATSUFontVariationAxis`
A value of type `ATSUFontVariationAxis` (page 192). This value represents the font variation axis whose value you want to obtain.

`oATSUFontVariationValue`
A pointer to a value of type `ATSUFontVariationValue` (page 193). On return, the value represents the value of the specified font variation axis. Note that if the feature selector value has not been set, `ATSUGetFontFeature` passes back the font-specified default value and returns the result code `kATSUNotSetErr`. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetVariations` (page 42)

`ATSUGetAllFontVariations` (page 45)

ATSUGetAllFontVariations

Obtains all the font variation values from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose font variations you want to obtain. You cannot pass `NULL` for this parameter.

`iVariationCount`
A value of type `ItemCount`. This value represents the number of font variation axes and values that you want passed back in the `oVariationAxes` and `oATSUFontVariationValues` arrays. You can predetermine this value by first calling `ATSUGetAllFontVariations (iStyle, 0, NULL, NULL, &oActualVariationCount)`.

`oVariationAxes`
An array of values of type `ATSUFontVariationAxis` (page 192). On return, the array contains the font variation axes for the specified font instance. You can predetermine how much memory to allocate for this array by first calling `ATSUGetAllFontVariations (iStyle, 0, NULL, NULL, &oActualVariationCount)`.

`oATSUFontVariationValues`
An array of values of type `ATSUFontVariationValue` (page 193). On return, the array contains the values of the font variation axes passed back in the `oVariationAxes` array. You can predetermine how much memory to allocate for this array by first calling `ATSUGetAllFontVariations (iStyle, 0, NULL, NULL, &oActualVariationCount)`.

`oActualVariationCount`
A pointer to a value of type `ItemCount`. On return, the value represents the actual number of font variations set in the style

object. This may be greater than the value passed in the `iVariationCount` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

ATSUClearFontVariations

Removes previously set font variation from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to a valid style object whose previously set font variations you want to remove. You cannot pass `NULL` for this parameter.

`iAxisCount` A value of type `ItemCount` that represents the number of previously set font variation values you want to remove. To remove all the previously set font variation values from a style object, pass the constant `kATSUClearAll`.

`iAxes` An array of values of type `ATSUFontVariationAxis` (page 192). Each element in the array must contain a valid font variation axis that corresponds to a font variation attribute value you want to remove. If you pass the `kATSUClearAll` constant in the `iAxisCount` parameter, the value in this parameter will be ignored.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUClearFontVariations` function removes those previously set font variation values that are identified by the font variation axes in the `iAxes` array. It sets these values to their font-defined default values.

To remove all previously set font variation values from a style object, pass the constant `kATSUClearAll` in the `iAxisCount` parameter. You can remove unset font variation values from a style object without a function error.

If you want to remove all previously set style run attribute, font feature, and font variation values from a style object, call the `ATSUClearStyle` (page 21) function.

VERSION NOTES

Available with ATSUI 1.0.

Functions for Obtaining Font Data

This section describes the ATSUI functions you can use to obtain data from a font.

- "Identifying and Finding ATSUI-Compatible Fonts" (page 47)
- "Finding Font Names" (page 52)
- "Converting Between Font IDs and Family Numbers" (page 59)
- "Obtaining Font Tracking Data" (page 61)
- "Obtaining Font Feature Data" (page 64)
- "Otaining Font Variation Data" (page 70)
- "Obtaining Font Instance Data" (page 74)

Identifying and Finding ATSUI-Compatible Fonts

ATSUI provides the following functions for identifying and finding the installed fonts that are ATSUI-compatible:

- `ATSUFontCount` (page 48) counts the number of installed fonts that are ATSUI-compatible.
- `ATSUGetFontIDs` (page 49) obtains the font IDs of all the installed fonts that are ATSUI-compatible.
- `ATSUFindFontFromName` (page 50) finds the first installed ATSUI-compatible font with a specified font name, name code, language, platform, and script.

ATSUFontCount

Counts the number of installed fonts that are ATSUI-compatible.

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

`oFontCount` A pointer to a value of type `ItemCount`. On return, the value represents the number of ATSUI-compatible fonts that are installed on the user's system. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUFontCount` function counts only those installed fonts that are compatible with ATSUI. Fonts that are incompatible with ATSUI include fonts that cannot be used to represent Unicode, the last resort font, and fonts whose names begin with a period or a percent sign. After calling `ATSUFontCount`, pass the number of fonts into the `iArraySize` parameter of the `ATSUGetFontIDs` (page 49) function to get the font IDs of all installed, ATSUI-compatible fonts.

Note that the number of available fonts may change while your application is running. Althought fonts cannot be removed from the Fonts folder while an application other than the Finder is running, they can be removed from other locations, resulting in a decrease in the font number. It is also possible for one to be added and another removed between two successive calls of `ATSUFontCount`, leaving this number unchanged. However, `ATSUGetFontIDs` (page 49) would return different results.

You should call `ATSUFontCount` and `ATSUGetFontIDs` whenever your application is brought to the foreground to rebuild your font menu, if necessary.

SEE ALSO

`ATSUGetFontIDs` (page 49)

ATSUGetFontIDs

Obtains the font IDs of all the installed fonts that are ATSUI-compatible.

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

`oFontIDs` An array of values of type `ATSUFontID` (page 192). On return, the array contains the IDs of all ATSUI-compatible installed fonts. You can predetermine how much memory to allocate for this array by first calling `ATSUGetFontIDs (&oFontIDs, 0, &oFontCount)`. You cannot pass `NULL` for this parameter.

`iArraySize` A value of type `ItemCount`. This value represents the number of IDs you want passed back in the `oFontIDs` parameter. You can predetermine this value by first calling `ATSUGetFontIDs (&oFontIDs, 0, &oFontCount)`.

`oFontCount` A pointer to a value of type `ItemCount`. On return, the value represents the actual number of installed fonts on the system. This may be greater than the `iArraySize` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUGetFontIDs` function obtains the font IDs of those installed fonts that are compatible with ATSUI. Fonts that are incompatible with ATSUI include fonts that cannot be used to represent Unicode, the last resort font, and fonts whose names begin with a period or a percent sign.

You should call `ATSUFontCount` before calling `ATSUGetFontIDs` to obtain the number of ATSUI-compatible fonts installed on the user's system. You can pass this value in the `iArraySize` parameter of `ATSUGetFontIDs` to get the font IDs of all installed, ATSUI-compatible fonts.

Note that the number of available fonts may change while your application is running. Althought fonts cannot be removed from the Fonts folder while an application other than the Finder is running, they can be removed from other locations, resulting in a decrease in the font number. It is also possible for one to be added and another removed between two successive calls of `ATSUFontCount`, leaving this number unchanged. However, `ATSUGetFontIDs` would return different results.

You should call `ATSUFontCount` and `ATSUGetFontIDs` whenever your application is brought to the foreground to rebuild your font menu, if necessary.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUFontCount` (page 48)

ATSUFindFontFromName

Finds the first installed font with the indicated name given the name's code, platform, script, and language.

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

`iName` A pointer to the buffer that contains the name of the font whose ID you want to obtain.

`iNameLength` A value of type `ByteCount` that represents the length (in bytes) of the font name.

`iFontNamePlatform`
: A value of type `FontPlatformCode`. The value identifies the type of platform that constitutes a match during a search for a font using other specific criteria. See "Font Name Platform Code Constants" (page 216) for a description of possible values. If any type of platform will constitute a match, pass the constant `kFontNoPlatform`, described in "No Font Name Platform, Language, or Script Constants" (page 235).

`iFontNameScript`
: A value of type `FontScriptCode`. The value identifies the type of language that constitutes a match during a search for a font using other specific criteria. See "Font Name Script Code Constants" (page 217) for a description of possible values. If any type of script will constitute a match, pass the constant `kFontNoScript`, described in "No Font Name Platform, Language, or Script Constants" (page 235).

`iFontNameLanguage`
: A value of type `FontLanguageCode`. The value identifies the type of language that constitutes a match during a search for a font using other specific criteria. See "Font Name Language Code Constants" (page 209) for a description of possible values. If any type of language will constitute a match, pass the constant `kFontNoLanguage`, described in "No Font Name Platform, Language, or Script Constants" (page 235).

`oFontID`
: A pointer to a value of type `ATSUFontID` (page 192). On return, the value represents the ID of the first installed font with the specified name code, platform, script, and language. Note that if no installed font corresponds to these parameters, `ATSUFindFontFromName` passes back the constant `kATSUInvalidFontID` in this parameter and returns the result code `kATSUInvalidFontErr`.

function result
: A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUFindFontFromName` function finds the first font name in the name table with the specified name code, platform, language, and script. Because ATSUI cannot guarantee the uniqueness of names among installed fonts,

`ATSUFindFontFromName` finds the first (but not necessarily the only) font with the specified parameters.

`ATSUFindFontFromName` is provided for convenience only. You may wish to replicate its functionality if you wish create a more sophisticated name-matching algorithm or better guarantee the uniqueness of names among installed fonts.

SPECIAL CONSIDERATIONS

`ATSUFindFontFromName` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetIndFontName` (page 54)

`ATSUFindFontName` (page 56)

Finding Font Names

ATSUI provides the following functions for finding font names:

- `ATSUCountFontNames` (page 53) counts the number of font name strings defined in a font name table.
- `ATSUGetIndFontName` (page 54) obtains the indicated name from a font's name table.
- `ATSUFindFontName` (page 56) finds the first font name in a font name table with a specified name, platform, script, and language code.

ATSUCountFontNames

Counts the number of font name strings defined in a font name table.

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

`iFont` A value of type `ATSUFontID` (page 192) that identifies the font whose font names you want to count.

`oFontNameCount` A pointer to a value of type `ItemCount`. On return, `oFontNameCount` points to the total number of entries in the font name table. This includes the names of font features, variations, tracking settings,and instances, as well as the types of font names identified by the pre-defined name constants described in "Font Name Code Constants" (page 213). You cannot pass a `NULL` pointer for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to an installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUCountFontNames` function passes back the total number of entries in a specified font's name table. This includes repetitions of the same name in different platforms, languages, and scripts, as well as other strings such as the names of font features, variations, tracking settings,and instances. You can use this count with the function `ATSUGetIndFontName` (page 54) to iterate through the entries of a font name table.

VERSION NOTES

Available with ATSUI 1.0.

ATSUGetIndFontName

Obtains the name, platform, script, and language codes and the font name string corresponding to an indexed font name.

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

`iFontID` A value of type `ATSUFontID` (page 192) that represents the font whose font name you want to obtain.

`iFontNameIndex` A value of type `ByteCount`. Pass the 0-based index of the font name you wish to find. The index should not exceed the value passed back by the function `ATSUCountFontNames` (page 53).

`iMaximumNameLength`
The number of bytes you have allocated to contain the font name in the buffer pointed to by `oName`. If you do not allocate enough space, `ATSUGetIndFontName` will pass back a partial string in the `oName` parameter. You can predetermine this value by first calling `ATSUGetIndFontName (iFontID, iFontNameIndex, 0, NULL, NULL, &oActualNameLength, NULL, NULL, NULL)`.

`oName` A pointer to the buffer that contains the font name string. On return, the buffer contains the name of the specified font. If the buffer you allocated is not large enough, `ATSUGetIndFontName` passes back a partial string. You can predetermine how much memory to allocate for this array by first calling `ATSUGetIndFontName (iFontID, iFontNameIndex, 0, NULL, NULL, &oActualNameLength, NULL, NULL, NULL)`. You cannot pass `NULL` for this parameter.

`oActualNameLength`
A pointer to a value of type `ByteCount`. On return, the value represents the actual length of the font name string. This may be larger than the value passed in the `iMaximumNameLength` parameter. You cannot pass `NULL` for this parameter.

`oFontNameCode` A value of type `FontNameCode`. On return, the value indicates the type of font name. See "Font Name Code Constants" (page 213) for a description of possible values.

`oFontNamePlatform`
A pointer to a value of type `FontPlatformCode`. On return, the value represents the type of platform. See "Font Name Platform Code Constants" (page 216) for a description of possible values.

`oFontNameScript`
A pointer to a value of type `FontScriptCode`. On return, the value represents the type of script. See "Font Name Script Code Constants" (page 217) for a description of possible values.

`oFontNameLanguage`
A pointer to a value of type `FontLanguageCode`. On return, the value represents the type of language. See "Font Name Language Code Constants" (page 209) for a description of possible values.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUGetIndFontName` function passes back the font name and information about the name (that is, its type, platform, script, and language) given an index in the list of font names for a particular font.

You can pass the count obtained by the function `ATSUCountFontNames` (page 53) into the `iFontNameIndex` parameter to iterate through the entries of a font name table.

SPECIAL CONSIDERATIONS

`ATSUGetIndFontName` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUFindFontFromName` (page 50)

`ATSUFindFontName` (page 56)

ATSUFindFontName

Finds the first font name in a font name table with a specified name, platform, script, and language code.

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

`iFontID` A value of type `ATSUFontID` (page 192) that identifies the font whose font name table you want to search.

`iFontNameCode` A value of type `FontNameCode`. Pass the type of font name you want to find. You must past a valid value in this parameter. See "Font Name Code Constants" (page 213) for a description of possible values.

iFontNamePlatform

A value of type FontPlatformCode. Pass the encoding of the font name you want to find. See "Font Name Platform Code Constants" (page 216) for a description of possible values. You can pass the constant kFontNoPlatform, described in "No Font Name Platform, Language, or Script Constants" (page 235), to indicate that you don't care about the encoding of the font name string. See the discussion for a description of this case.

iFontNameScript

A value of type FontScriptCode. Pass the platform-specific ID of the font name you want to find. See "Font Name Script Code Constants" (page 217) for a description of possible values. You can pass the constant kFontNoScript, described in "No Font Name Platform, Language, or Script Constants" (page 235), to indicate that you don't care about the platform-specific ID of the font name string. See the discussion for a description of this case.

iFontNameLanguage

A value of type FontLanguageCode. Pass the language of the font name you want to find. See "Font Name Language Code Constants" (page 209) for a description of possible values. You can pass the constant kFontNoLanguage, described in "No Font Name Platform, Language, or Script Constants" (page 235), to indicate that you don't care about the language of the font name string. See the discussion for a description of this case.

iMaximumNameLength

A value of type ByteCount. Pass the number of bytes that you expect for the font name string or 0 if you don't know the length. For more information, see the discussion. If the value is less than the actual string length, ATSUFindFontName will pass back a truncated string in the oName parameter and the actual length in the oActualNameLength parameter. In this case, you should call ATSUFindFontName again with the actual length passed back from the previous call in this parameter.

oName

A pointer to a buffer containing the font name string. Before calling ATSUFindFontName, pass a pointer to memory you have allocated for the buffer or NULL if you do not know the size of the font name string. If you have allocated enough memory for the buffer, on return, oName points to the font name string. If the

buffer is not large enough, `ATSUFindFontName` passes back a partial string. If you passed `NULL`, on return, `ATSUFindFontName` passes back the actual string length in the `oActualNameLength` parameter.

`oActualNameLength`
A pointer to a value of type `ByteCount`. On return, the value represents the actual length of the font name string. You should check this value to make sure that you allocated enough memory for the buffer. For more information, see the discussion.

`oFontNameIndex`
A pointer to a value of type `ItemCount`. On return, the value represents the index of the font name in the font's list of font names.

function result A result code. The result code `kATSUNotSetErr` indicates that the font has no name in its name table matching the given parameters. The result code `kATSUInvalidFontErr` indicates that the specified font does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUFindFontName` function searches the specified font's name table for a font name with the specified name, platform, script, and language codes. If it finds a font name, `ATSUFindFontName` passes back font name in the `oName` parameter and its index in the font's list of font names in the `oFontNameIndex` parameter.

You can predetermine this value by first calling `ATSUFindFontName (iFontID, iFontNameCode, iFontNamePlatform, iFontNameScript, iFontNameLanguage, 0, NULL, &oActualNameLength, NULL)`.

If it is larger than the value you passed in the `iMaximumNameLength` parameter, you should call `ATSUFindFontName` again and use this value in `iMaximumNameLength` and to allocate memory for the buffer in `oName`.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUFindFontFromName` (page 50)

`ATSUGetIndFontName` (page 54)

Converting Between Font IDs and Family Numbers

ATSUI provides the following functions for converting between font IDs and family numbers:

- `ATSUFONDtoFontID` (page 59) obtains the ID of a font with a specified font family number (and style, if it exists).
- `ATSUFontIDtoFOND` (page 60) obtains the font family number (and style, if it exists) of a font with a specified ID.

ATSUFONDtoFontID

Obtains the ID of a font with a specified font family number (and style, if it exists).

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

`iFONDNumber` A value of type `short`. The value represents the font family number of the font whose ID you want to obtain.

`iFONDStyle` A value of type `Style`. The unsigned `char` represents the font family style, if it exists, of the font whose ID you want to obtain. Font family styles exist in fonts that split a font family into several font family numbers.

`oFontID` A pointer to a value of type `ATSUFontID` (page 192). On return, the value represents the ID of the font with the specified font family number and style. Note that if there are no installed fonts with the specified parameters, `ATSUFONDtoFontID` passes back the constant `kATSUInvalidFontID` and returns the result code `kATSUInvalidFontErr`. If a font exists with the specified font family number and style, but that font isn't compatible with ATSUI, `ATSUFONDtoFontID` passes back the constant

kATSUInvalidFontID and returns the result code kATSUNoCorrespondingFontErr. You cannot pass NULL for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The ATSUFONDtoFontID function obtains the ID of a font with a specified font family number if it is compatible with ATSUI. Fonts that are incompatible with ATSUI do not have font IDs. Incompatible fonts include fonts that cannot be used to represent Unicode, the last resort font, and fonts whose names begin with a period or a percent sign.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

ATSUFontIDtoFOND (page 60)

ATSUFontIDtoFOND

Obtains the font family number (and style, if it exists) of a font with a specified ID.

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

iFontID A value of type ATSUFontID (page 192). The value represents the ID of the font whose font family number you want to obtain.

oFONDNumber A pointer to a value that represents the font family number of the specified font. Note that if there are no installed fonts with the specified ID, ATSUFontIDtoFOND passes back the constant kATSUInvalidFontID and returns the result code kATSUInvalidFontErr. If the specified ID correspond to an

existing font, but that font isn't compatible with ATSUI, `ATSUFONDtoFontID` passes back the constant `kATSUInvalidFontID` and returns the result code `kATSUNoCorrespondingFontErr`.You cannot pass `NULL` for this parameter.

`oFONDStyle` A pointer to a value of type `Style`. On return, the unsigned `char` represents the font family style, if it exists, of the specified font. Font family styles exist in fonts that split a font family into several font family numbers. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUFONDtoFontID` (page 59)

Obtaining Font Tracking Data

ATSUI provides the following functions for obtaining font tracking data:

- `ATSUCountFontTracking` (page 61) counts the number of font tracking settings in a font.
- `ATSUGetIndFontTracking` (page 62) obtains an indexed font tracking value and its name code.

ATSUCountFontTracking

Counts the number of font tracking settings in a font.

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

`iFontID` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose tracking settings you want to count.

`iCharacterOrientation`
A value of type `ATSUVerticalCharacterType`. This value represents the glyph orientation for the font tracking values that you want to count. See "Glyph Orientation Constants" (page 223) for a description of possible values. It is necessary to specify this value because font tracking settings differ depending upon glyph orientation.

`oTrackingCount` A pointer to a value of type `ItemCount`. On return, the value represents the number of font tracking settings in the font. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUGetIndFontTracking` (page 62)

ATSUGetIndFontTracking

Obtains an indexed font tracking value and its name code.

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

`iFontID` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose font tracking value and name code you want to obtain.

`iCharacterOrientation`
A value of type `ATSUVerticalCharacterType`. This value represents the glyph orientation for the font tracking setting and name code you want to obtain. See "Glyph Orientation Constants" (page 223) for a description of possible values. It is necessary to specify this value because there are different font tracking values for different glyph orientations.

`iTrackIndex` The index of the font tracking value whose name code you want to obtain. To predetermine the maximum valid index, call `ATSUCountFontTracking (iFontID, iCharacterOrientation, &oTrackingCount)` and subtract one from the value passed back in the `oTrackingCount` parameter.

`oFontTrackingValue`
A pointer to a `Fixed` value. On return, `oFontTrackingValue` points to the font tracking value corresponding to the specified index and character orientation.

`oNameCode` A pointer to a value of type `FontNameCode`. On return, `oNameCode` points to the name code of the font tracking value. See "Font Name Code Constants" (page 213) for a description of possible values. You can pass this value to the `ATSUFindFontName` (page 56) function to find the font tracking name identified by this name code. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUGetIndFontTracking` function passes back the name code and value of the indexed font tracking setting passed in the `iTrackIndex` parameter, given the specified character orientation. You can pass the name code value into the `iFontNameCode` parameter of the `ATSUFindFontName` (page 56) function to find the font tracking name identified by this name code.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUCountFontTracking` (page 61)

Obtaining Font Feature Data

ATSUI provides the following functions for obtaining font feature data:

- `ATSUCountFontFeatureTypes` (page 64) counts the number of available font feature in a font.
- `ATSUGetFontFeatureTypes` (page 65) obtains all the feature types from a font feature.
- `ATSUCountFontFeatureSelectors` (page 66) counts the number of feature selectors in a specified font feature and feature type.
- `ATSUGetFontFeatureSelectors` (page 67) obtains all the font feature selectors from a font feature.
- `ATSUGetFontFeatureNameCode` (page 69) obtains a feature selector or feature type name code.

ATSUCountFontFeatureTypes

Counts the number of available font feature in a font.

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

`iFontID` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose font features you want to count.

`oTypeCount` A pointer to a value of type `ItemCount`. On return, the value represents the number of font features available in the font. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetFontFeatureTypes` (page 65)

ATSUGetFontFeatureTypes

Obtains all the feature types from a font feature.

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

`iFont` A value of type `ATSUFontID` (page 192). This value represents the ID of the font whose feature types you want to obtain.

`iMaximumTypes` A value of type `ItemCount`. This value represents the number of feature types you want passed back in the `oTypes` array. You can predetermine this value by first calling `ATSUGetFontFeatureTypes (iFont, 0, NULL, &oActualTypeCount)`.

`oTypes` An array of values of type `ATSUFontFeatureType` (page 191). On return, the array contains the font feature types that are in the font. You can predetermine how much memory to allocate for this array by first calling `ATSUGetFontFeatureTypes (iFont, 0, NULL, &oActualTypeCount)`. You cannot pass `NULL` for this parameter.

`oActualTypeCount`
A pointer to a value of type `ItemCount`. On return, the value represents the actual number of font feature types set in the font. This may be greater than the value passed in the `iMaximumTypes` parameter. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCountFontFeatureTypes` (page 64)

ATSUCountFontFeatureSelectors

Counts the number of feature selectors in a specified font feature and feature type.

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

`iFont` A value of type `ATSUFontID` (page 192). This value represents the ID of the font whose defined feature selectors you want to count.

`iType` A value of type `ATSUFontFeatureType` (page 191). This value represents a valid feature type whose feature selectors you want to count.

`oSelectorCount` A pointer to a value of type `ItemCount`. On return, the value represents the number of feature selectors defined in the specified feature type. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetFontFeatureSelectors` (page 67)

ATSUGetFontFeatureSelectors

Obtains all the font feature selectors from a font feature.

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

`iFont` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose feature type you want to determine the defined feature selectors for.

`iType` A value of type `ATSUFontFeatureType` (page 191) that represents a valid font feature type whose defined font selectors you want to obtain.

`iMaximumSelectors`
A value of type `ItemCount`. This value represents the number of font feature selectors you want passed back in the `oSelectors` array. You can predetermine this value by first calling `ATSUGetFontFeatureSelectors (iFont, iType, 0, NULL, NULL, &oActualSelectorCount)`.

`oSelectors` An array of values of type `ATSUFontFeatureSelector` (page 191). On return, the array contains the font feature selectors defined for the specified font feature type. You can predetermine how much memory to allocate for this array by first calling `ATSUGetFontFeatureSelectors (iFont, iType, 0, NULL, NULL, &oActualSelectorCount)`. You cannot pass `NULL` for this parameter.

`oSelectorIsOnByDefault`
An array of values of type `Boolean`. On return, each element in the array contains a value that indicates whether the corresponding font feature selector is on. If `true`, the font feature selector is on by default. You can predetermine how much memory to allocate for this array by first calling `ATSUGetFontFeatureSelectors (iFont, iType, 0, NULL, NULL, &oActualSelectorCount)`. You cannot pass `NULL` for this parameter.

`oActualSelectorCount`
A pointer to a value of type `ItemCount`. On return, the value represents the actual number of font feature selectors defined for the font. This may be greater than the value passed in the `iMaximumSelectors` parameter. You cannot pass `NULL` for this parameter.

`oIsMutuallyExclusive`
A pointer to a value of type `Boolean`. On return, the value indicates whether the font feature selectors can be on simultaneously in the feature type. If `true`, only one selector can be used at a time. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCountFontFeatureSelectors` (page 66)

ATSUGetFontFeatureNameCode

Obtains a feature selector or feature type name code.

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

`iFont` A value of type `ATSUFontID` (page 192). This value represents the ID of the font whose feature type or selector name code you want to obtain.

`iType` A value of type `ATSUFontFeatureType` (page 191). This value represents the feature type whose name code you want to obtain.

`iSelector` A value of type `ATSUFontFeatureSelector` (page 191). Pass the feature selector whose name code you want to obtain.

`oNameCode` A pointer to a value of type `FontNameCode`. On return, the value represents the name code of the feature type or selector. See "Font Name Code Constants" (page 213) for a description of possible values. If you pass the constant `kATSUNoSelector` in the `iSelector` parameter, the value passed back represents the feature type name code. You can pass this value to the `ATSUFindFontName` (page 56) function to find the font feature type or selector name identified by this name code. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. The result code `kATSUNotSetErr` indicates that the font has no name in its name table for the indicated font feature. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUGetFontFeatureNameCode` function passes back the name code of the feature selector unless you pass the `kATSUNoSelector` constant in the `iSelector` parameter. In this case, it passes back the name code of the feature type.

You can pass this value in the `iFontNameCode` parameter of the `ATSUFindFontName` (page 56) function to find the font feature type or selector name identified by this name code.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetFontFeatureSelectors` (page 67)

Otaining Font Variation Data

ATSUI provides the following functions for obtaining font variation data:

- `ATSUCountFontVariations` (page 70) counts the number of font variations for a specified font.
- `ATSUGetIndFontVariation` (page 71) obtains the axis and values from a font variation.
- `ATSUGetFontVariationNameCode` (page 73) obtains a font variation name code.

ATSUCountFontVariations

Counts the number of font variations defined for a specified font.

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

`iFont` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose font variations you want to count.

`oVariationCount`
A pointer to a value of type `ItemCount`. On return, the value represents the number of font variations defined for the specified font. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetIndFontVariation` (page 71)

ATSUGetIndFontVariation

Obtains the axis and values from a font variation.

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

`iFont` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose specific font variation values you want to obtain.

`iVariationIndex`
A value of type `ItemCount`. This value represents the index of the font variation whose axis and values you want to obtain. To predetermine the maximum valid value for this index, call `ATSUCountFontVariations (iFont, &oVariationCount)` and subtract one from the value passed back in the `oVariationCount` parameter.

`oATSUFontVariationAxis`
A pointer to a value of type `ATSUFontVariationAxis` (page 192). On return, the value represents the variation axis for the specified font variation. You cannot pass `NULL` for this parameter.

`oMinimumValue`
A pointer to a value of type `ATSUFontVariationValue` (page 193). On return, the value represents the minimum value for the specified font variation. You cannot pass `NULL` for this parameter.

`oMaximumValue`
A pointer to a value of type `ATSUFontVariationValue` (page 193). On return, the value represents the maximum value for the specified font variation. You cannot pass `NULL` for this parameter.

`oDefaultValue` A pointer to a value of type `ATSUFontVariationValue` (page 193). On return, the value represents the default value for the specified font variation. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUGetIndFontVariation` function to determine the font variation axis and the minimum, maximum, and default values for a specified font variation.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCountFontVariations` (page 70)

ATSUGetFontVariationNameCode

Obtains a font variation name code.

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

`iFont`	A value of type `ATSUFontID` (page 192) that represents the ID of the font whose variation name code you want to obtain.
`iAxis`	A value of type `ATSUFontVariationAxis` (page 192). This value represents the font variation whose name code you want to obtain.
`oNameCode`	A pointer to a value of type `FontNameCode`. On return, the value represents the name code of the font variation. See "Font Name Code Constants" (page 213) for a description of possible values. You can pass this value to the `ATSUFindFontName` (page 56) function to find the font variation name identified by this name code. You cannot pass `NULL` for this parameter.
function result	A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. The result code `kATSUNotSetErr` indicates that the font has no name in its name table for the indicated font variation. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUGetIndFontVariation` function passes back the name code of the specified font variation. You can pass this value in the `iFontNameCode` parameter of the `ATSUFindFontName` (page 56) function to find the font variation name identified by this name code.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetIndFontVariation` (page 71)

Obtaining Font Instance Data

ATSUI provides the following functions for obtaining font instance data:

- `ATSUCountFontInstances` (page 74) counts the number of font instances available in a font.
- `ATSUGetFontInstance` (page 75) obtains the indicated name from a font's name table.
- `ATSUGetFontInstanceNameCode` (page 77) obtains a font instance name code.

ATSUCountFontInstances

Counts the number of font instances available in a font.

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

`iFont` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose font instances you want to count.

`oInstances` A pointer to a value of type `ItemCount`. You cannot pass a `NULL` pointer for this parameter. On return, the value represents the number of font instances in the font.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetFontInstance` (page 75)

ATSUGetFontInstance

Obtains the font name for a font instance you specify by index.

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

`iFont` A value of type `ATSUFontID` (page 192) that identifies the font whose instance name you want to obtain.

`iFontInstanceIndex`
A value of type `ItemCount`. Pass the 0-based index of the font instance whose name you want to find. The number of font instances is returned by the `ATSUCountFontInstances` (page 74) function.

`iMaximumVariations`
A value of type `ItemCount`. Pass the number of axes and values you want passed back in the `oAxes` and `oValues` arrays. To predetermine this value, see the discussion below.

`oAxes` An array of values of type `ATSUFontVariationAxis` (page 192). If you pass `NULL`, on return, `ATSUGetFontInstance` passes back the size of this array in the `oActualVariationCount` parameter. If you instead pass a pointer to memory you have allocated for the array, on return, this array contains the font variation axes of the indexed font instance. You cannot pass `NULL` for this parameter.

`oValues` An array of values of type `ATSUFontVariationValue` (page 193). If you pass `NULL`, on return, `ATSUGetFontInstance` passes back the size of this array in the `oActualVariationCount` parameter. If you instead pass a pointer to memory you have allocated for the array, on return, this array contains the values of the axes in the `oAxes` parameter. You cannot pass `NULL` for this parameter.

`oActualVariationCount`
A pointer to a value of type `ItemCount`. On return, this value represents the actual number of font variations defined for the

font. This may be greater than the value passed in the `iMaximumVariations` parameter. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUGetFontInstance` function passes back the font name for the font instance specified by the value of the `iFontInstanceIndex` parameter. Then `ATSUGetFontInstance` copies the font variation settings for that instance into the `ATSUFontVariationAxis` and `ATSUFontVariationValue` parameters, if they are not set to `NULL`.

Each instance is always identified by a complete set of font variations. A font instance contains a value for each font variation available, even if that value is only the default font variation setting.

Your application must allocate enough memory in the `ATSUFontVariationAxis` and `ATSUFontVariationValue` parameters to store as many font variations as are available. You can predetermine this number by calling the `ATSUCountFontInstances` (page 74) function.

The best way to use `ATSUGetFontInstance` is to call it twice:

1. Pass the font ID and index in the `iFont` and `iFontInstanceIndex` parameters and `NULL` for the `iMaximumVariations`, `ATSUFontVariationAxis`, and `ATSUFontVariationValue` parameters. `ATSUGetFontInstance` returns the actual number of font variations in the `oActualVariationCount` parameter.
2. Allocate enough space for arrays of the returned size, then call the function again, passing a pointer in the `ATSUFontVariationAxis` and `ATSUFontVariationValue` parameters. On return, the arrays contain the font variations axes and values.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCountFontInstances` (page 74)

ATSUGetFontInstanceNameCode

Obtains a font instance name code.

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

`iFont` A value of type `ATSUFontID` (page 192) that represents the ID of the font whose instance name code you wish to obtain.

`iInstanceIndex`
The index of the font instance value whose name code you want to obtain. To predetermine the maximum valid value for this index, call `ATSUCountFontInstance (iFont, &oInstances)` and subtract one from the value passed back in the `oInstances` parameter.

`oNameCode` A pointer to a value of type `FontNameCode`. On return, `oNameCode` points to the name code of the font instance setting. See "Font Name Code Constants" (page 213) for a description of possible values. You can pass this value to the `ATSUFindFontName` (page 56) function to find the font instance setting name identified by this name code. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the ID does not correspond to any installed font. The result code `kATSUNotSetErr` indicates that the font has no name in its name table for the indicated font variation. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUGetFontInstanceNameCode` function passes back the name code of the indexed font instance setting passed in the `iInstanceIndex` parameter. You can pass the name code value into the `iFontNameCode` parameter of the

ATSUFindFontName (page 56) function to find the font variation name identified by this name code.

VERSION NOTES

Available with ATSUI 1.0.

Functions for Manipulating Text Layout Objects

This section describes the you can use to manipulate a text layout object and its contents:

- "Creating and Disposing of Text Layout Objects" (page 78)
- "Manipulating Text Layout Attributes" (page 90)
- "Manipulating Line Attributes" (page 98)
- "Determining and Updating Text Memory Location" (page 106)
- "Updating and Determining Style Runs" (page 114)
- "Providing Font Substitutions" (page 118)

Creating and Disposing of Text Layout Objects

ATSUI provides the following functions for creating and disposing of text layout objects:

- ATSUCreateTextLayout (page 79) creates a text layout object.
- ATSUCreateTextLayoutWithTextPtr (page 80) creates a text layout object containing a pointer to a Unicode text buffer.
- ATSUCreateTextLayoutWithTextHandle (page 83) creates a text layout object containing a handle to a Unicode text buffer.
- ATSUCreateAndCopyTextLayout (page 85) creates a copy of a text layout object.
- ATSUSetTextLayoutRefCon (page 87) sets application-specific text layout data.
- ATSUGetTextLayoutRefCon (page 88) obtains application-specific text layout data.

- `ATSUClearLayoutCache` (page 88) clears the entire layout cache, or if specified, a single line's layout cache.
- `ATSUDisposeTextLayout` (page 90) disposes of the memory associated with a text layout object.

ATSUCreateTextLayout

Creates a text layout object.

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

`oTextLayout` A pointer to a reference of type `ATSUTextLayout` (page 195). On return, the reference points to the newly created text layout object.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCreateTextLayout` function creates a text layout object contains the default text layout attribute values listed in Table 2-2 (page 251). To set the text layout attributes to non-default values, call the function `ATSUSetLayoutControls` (page 92).

The newly-created text layout object is uninitialized (that is, it does not contain a pointer or handle to a text buffer or style runs). Many ATSUI functions require that you initialize a text layout object before passing it to these functions. To determine whether or not a particular function requires an initialized text layout object, you should check the function's `iTextLayout` parameter description.

To initialize a text layout object, you should call the following functions:

- `ATSUSetRunStyle` (page 114) to assign style runs
- `ATSUSetTextPointerLocation` (page 107) or `ATSUSetTextHandleLocation` (page 109) to set a pointer or handle to a Unicode text buffer
- `ATSUSetSoftLineBreak` (page 159) or `ATSUBreakLine` (page 156) to set soft line breaks

You can also create an initialized text layout object by instead calling the function `ATSUCreateTextLayoutWithTextHandle` (page 83) or `ATSUCreateTextLayoutWithTextPtr` (page 80).

SPECIAL CONSIDERATIONS

`ATSUCreateTextLayout` allocates memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

ATSUCreateTextLayoutWithTextPtr

Creates a text layout object containing a pointer to a Unicode text buffer.

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

`iText` A pointer of type `ConstUniCharArrayPtr` (page 197). Pass a pointer to the Unicode text buffer that you want to assign to the text layout object. You must supply this buffer with a block of Unicode text. You are responsible for making sure there is always text in this buffer as long as the text layout object exists. You are also responsible for allocating the memory associated with this pointer.

`iTextOffset` A value of type `UniCharArrayOffset` (page 198). Pass the edge offset of the beginning of the range of text that you want to perform layout operations on. You can pass the constant

kATSUFromTextBeginning, described in "Text Offset Constant" (page 256), if you want the range of text to start at the beginning of the text buffer.

iTextLength — A value of type UniCharCount (page 198). Pass the length of the range of text you want to perform layout operations on. You can pass the constant kATSUtexttoEnd, described in "Text Length Constant" (page 255), if you want the range of text to extend to the end of the text buffer. Note that if the range of text is outside the text buffer, ATSUCreateTextLayoutWithTextPtr returns the result code kATSUInvalidTextRangeErr.

iTextTotalLength — A value of type UniCharCount (page 198). Pass the the length of the text buffer. This value should be greater than the range of text you passed in the iTextOffset and iTextLength parameters, unless you want to perform layout operations on the entire text buffer.

iNumberOfRuns — A value of type ItemCount. Pass the number of style runs you want to assign to the text layout object.

iRunLengths — An array of values of type UniCharCount (page 198). Pass an array of style run lengths. Each element in the array must correspond to a style object in the iStyles array. You can pass the constant kATSUToTextEnd, described in "Text Length Constant" (page 255), for the last style run length if you wish it to extend to end of the text buffer. If the sum of the style run lengths is less than the value you passed in the iTextLength parameter, the remaining characters in the range of text are automatically assigned to the last style run.

iStyles — An array of references of type ATSUStyle (page 195). Pass an array of style objects. Each element in the array must reference a valid style object and correspond to a style run length in the iRunLengths array.

oTextLayout — A pointer to a reference of type ATSUTextLayout (page 195). On return, oTextLayout points to the newly-created text layout object. You cannot pass NULL for this parameter.

function result — A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCreateTextLayoutWithTextPtr` function creates a text layout object containing a pointer to a Unicode text buffer and the default text layout attribute values listed in Table 2-2 (page 251). To set the text layout attributes to non-default values, call the function `ATSUSetLayoutControls` (page 92). To set the attributes of a single line in the text layout object, call the function `ATSUSetLineControls` (page 100).

The newly-created text layout object is initialized (that is, it contains a pointer to a Unicode text buffer and at least one style run). You can also create an initialized text layout object by calling the function `ATSUCreateTextLayoutWithTextHandle` (page 83).

You must specify the range of text you want to perform layout operations on in the `iTextOffset` and `iTextLength` parameters. To include the entire text buffer, pass the constants `kATSUFromTextBeginning` and `kATSUTexttoEnd`, respectively.

If you pass in a range of text that is a subset of the text buffer, the text layout object will scan the remaining text to get the full context for such things as bidirectional processing.

You must supply the text buffer with a block of Unicode text. You are also responsible for updating the memory location of the text buffer whenever the user inserts, deletes, or moves text. To obtain the current memory location of the text buffer, call the function `ATSUSetTextPointerLocation` (page 107).

If you want to create a copy (essentially, a clone) of this text layout object, call the function `ATSUCreateAndCopyTextLayout` (page 85). You can copy the attributes of this text layout object into another text layout object by calling the function `ATSUCopyLayoutControls` (page 91). You can also copy the attributes of a line of this text layout object into another line in the same or different text layout object by calling the function `ATSUCopyLineControls` (page 98).

SPECIAL CONSIDERATIONS

`ATSUCreateTextLayoutWithTextPtr` allocates memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

ATSUCreateTextLayoutWithTextHandle

Creates a text layout object containing a handle to a Unicode text buffer.

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

`iText` A handle of type `UniCharArrayHandle` (page 197). Pass handle that refers to the Unicode text buffer that you want to assign to the text layout object. You must supply this buffer with a block of Unicode text. You are responsible for making sure there is always text in this buffer as long as the text layout object exists. You are also responsible for allocating the memory associated with this handle. ATSUI functions will dereference the handle before accessing the text, but will leave the handle's state unchanged.

`iTextOffset` A value of type `UniCharArrayOffset` (page 198). Pass the edge offset of the beginning of the range of text that you want to perform layout operations on. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), if you want the range of text to start at the beginning of the text buffer.

`iTextLength` A value of type `UniCharCount` (page 198). Pass the length of the range of text you want to perform layout operations on. You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), if you want the range of text to extend to the end of the text buffer. Note that if the range of text is outside the text buffer, `ATSUCreateTextLayoutWithTextPtr` returns the result code `kATSUInvalidTextRangeErr`.

`iTextTotalLength`
A value of type `UniCharCount` (page 198). Pass the the length of the text buffer. This value should be greater than the range of

text you passed in the `iTextOffset` and `iTextLength` parameters, unless you want to perform layout operations on the entire text buffer.

`iNumberOfRuns` A value of type `ItemCount`. Pass the number of style runs you want to assign to the text layout object.

`iRunLengths` An array of values of type `UniCharCount` (page 198). Pass an array of style run lengths. Each element in the array must correspond to a style object in the `iStyles` array. You can pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 255), for the last style run length if you wish it to extend to end of the text buffer. If the sum of the style run lengths is less than the value you passed in the `iTextLength` parameter, the remaining characters in the range of text are automatically assigned to the last style run.

`iStyles` An array of references of type `ATSUStyle` (page 195). Pass an array of style objects. Each element in the array must reference a valid style object and correspond to a style run length in the `iRunLengths` array.

`oTextLayout` A pointer to a reference of type `ATSUTextLayout` (page 195). On return, `oTextLayout` points to the newly-created text layout object. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCreateTextLayoutWithTextHandle` function creates a text layout object containing a handle to a Unicode text buffer and the default text layout attribute values listed in Table 2-2 (page 251). To set the text layout attributes to non-default values, call the function `ATSUSetLayoutControls` (page 92). To set the attributes of a single line in the text layout object, call the function `ATSUSetLineControls` (page 100).

The newly-created text layout object is initialized (that is, it contains a pointer to a Unicode text buffer and at least one style run). You can also create an initialized text layout object by calling the function `ATSUCreateTextLayoutWithTextPtr` (page 80).

You must specify the range of text you want to perform layout operations on in the `iTextOffset` and `iTextLength` parameters. To include the entire text buffer, pass the constants `kATSUFromTextBeginning` and `kATSUTexttoEnd`, respectively.

If you pass in a range of text that is a subset of the text buffer, the text layout object will scan the remaining text to get the full context for such things as bidirectional processing.

You must supply the text buffer with a block of Unicode text. You are also responsible for updating the memory location of the text buffer whenever the user inserts, deletes, or moves text. To obtain the current memory location of the text buffer, call the function `ATSUSetTextHandleLocation` (page 109).

If you want to create a copy (essentially, a clone) of this text layout object, call the function `ATSUCreateAndCopyTextLayout` (page 85). You can copy the attributes of this text layout object into another text layout object by calling the function `ATSUCopyLayoutControls` (page 91). You can also copy the attributes of a line of this text layout object into another line in the same or different text layout object by calling the function `ATSUCopyLineControls` (page 98).

SPECIAL CONSIDERATIONS

`ATSUCreateTextLayoutWithTextHandle` allocates memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

ATSUCreateAndCopyTextLayout

Creates a copy of a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose contents you want to copy. You cannot pass `NULL` for this parameter.

`oTextLayout`	A pointer to a reference of type `ATSUTextLayout` (page 195). On return, the reference points to a text layout object containing the same text layout attribute values, style runs, and soft line breaks as the text layout object you passed in the `iTextLayout` parameter. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCreateAndCopyTextLayout` function creates a clone of a text layout object that contains the same text layout attribute values, style runs, and soft line breaks. `ATSUCreateAndCopyTextLayout` does not copy reference constants or layout caches.

In order to duplicate the functionality of `ATSUCreateAndCopyTextLayout`, you would have to call the following functions:

- `ATSUGetRunStyle` (page 115) to obtain the style runs from `iTextLayout` and `ATSUSetRunStyle` (page 114) to set these style runs in `oTextLayout`
- `ATSUGetTextLocation` (page 112) to obtain the pointer or handle to a Unicode text buffer from `iTextLayout` and `ATSUSetTextPointerLocation` (page 107) or `ATSUSetTextHandleLocation` (page 109) to set this pointer or handle in `oTextLayout`
- `ATSUGetLayoutControl` (page 94) to obtain the previously set text layout attributes from `iTextLayout` and `ATSUSetLayoutControls` (page 92) to set these text layout attributes in `oTextLayout`
- `ATSUGetLineControl` (page 102) to obtain the previously set line attributes from `iTextLayout` and `ATSUSetLineControls` (page 100) to set these line attributes in `oTextLayout`
- `ATSUGetSoftLineBreaks` (page 160) to obtain the previously set soft line breaks in `iTextLayout` and `ATSUSetSoftLineBreak` (page 159) or `ATSUBreakLine` (page 156) to set these soft line breaks

SPECIAL CONSIDERATIONS

`ATSUCreateAndCopyTextLayout` allocates memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.1.

ATSUSetTextLayoutRefCon

Sets application-specific text layout data.

```
OSStatus ATSUSetTextLayoutRefCon (
                    ATSUTextLayout iTextLayout,
                    UInt32 iRefCon);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose application-specific data you want to set. You cannot pass `NULL` for this parameter.

`iRefCon` A 32-bit value, pointer, or handle to application-specific text layout data.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Note that when you copy a text layout object that contains a reference constant, the reference constant will not be copied. When you dispose of a text layout object that contains a reference constant, you are resposible for freeing any memory allocated for the reference constant. Calling `ATSUDisposeTextLayout` (page 90) will not do so.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetTextLayoutRefCon` (page 88)

ATSUGetTextLayoutRefCon

Obtains application-specific text layout data.

```
OSStatus ATSUGetTextLayoutRefCon (
                    ATSUTextLayout iTextLayout,
                    UInt32 *oRefCon);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose application-specific data you want to obtain. You cannot pass `NULL` for this parameter.

`oRefCon` A pointer to a 32-bit value, pointer, or handle to application-specific text layout data.You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetTextLayoutRefCon` (page 87)

ATSUClearLayoutCache

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

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose layout cache you want to clear. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the line whose layout cache you want to discard. To clear the entire layout

cache of the text layout object, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256).

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUClearLayoutCache` function flushes the layout cache of a single line or an entire text layout object but does not alter previously set text layout attribute values, soft line break positions, or text memory location. Individual lines may be redrawn as before.

You can call `ATSUClearLayoutCache` to free memory associated with the layout results of a text layout object.

The layout cache contains all the layout information ATSUI 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. This information is used when ATSUI redraws text that was recently drawn. It uses information in the layout cache to quickly lay out the text.

You should call `ATSUClearLayoutCache` when a text layout attribute that affects text layout is changed or line breaks are altered (for example, if line justification is set to full justification).

When you flush the cache, you retain the soft line breaks, text layout attribute values, and style runs that were previously set. If you do not care about retaining these values, you should dispose of the text layout object by calling the `ATSUDisposeTextLayout` (page 90) function.

It is not an error if some or all of the lines do not already have layout caches. `ATSUClearLayoutCache` only clears the layout caches it can find.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUDisposeTextLayout` (page 90)

ATSUDisposeTextLayout

Disposes of the memory associated with a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to the text layout whose memory you want to dispose. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUDisposeTextLayout` function to dispose of the memory associated with a text layout object, including text layout attribute settings and style runs. `ATSUDisposeTextLayout` only frees memory associated with the text layout object and its internal structures. It does not dispose of the memory pointed to by custom text layout attributes and reference constants. You are ultimately responsible for doing so.

VERSION NOTES

Available with ATSUI 1.0.

Manipulating Text Layout Attributes

ATSUI provides the following functions for manipulating text layout attributes:

- `ATSUCopyLayoutControls` (page 91) copies both set and unset attribute values from the source into the destination text layout object.
- `ATSUSetLayoutControls` (page 92) sets attribute values of a text layout object.
- `ATSUGetLayoutControl` (page 94) obtains an attribute value from a text layout object.
- `ATSUGetAllLayoutControls` (page 95) obtains attribute values from a text layout object.
- `ATSUClearLayoutControls` (page 97) removes previously set attribute values from a text layout object.

ATSUCopyLayoutControls

Copies both set and unset attribute values from the source into the destination text layout object.

```
OSStatus ATSUCopyLayoutControls (
                    ATSUTextLayout iSource,
                    ATSUTextLayout iDest);
```

`iSource` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose set and unset text layout attribute values you want to copy. You cannot pass `NULL` for this parameter.

`iDest` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose text layout attribute values you want to replace. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCopyLayoutControls` function copies both previously set and unset (that is, default) text layout attribute values from the source into the destination style object.

`ATSUCopyLayoutControls` will not copy the contents of memory referenced by pointers or handles within reference constants. It is your responsibility to ensure that this memory remains valid until the source text layout object is disposed of.

SPECIAL CONSIDERATIONS

`ATSUCopyLayoutControls` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUCopyLineControls` (page 98)

ATSUSetLayoutControls

Sets attribute values of a text layout object.

```
OSStatus ATSUSetLayoutControls (
                    ATSUTextLayout iTextLayout,
                    ItemCount iAttributeCount,
                    ATSUAttributeTag iTag[],
                    ByteCount iValueSize[],
                    ATSUAttributeValuePtr iValue[]);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to a text layout object whose text layout attribute values you want to set. The text layout object can be uninitialized. You cannot pass `NULL` for this parameter.

`iAttributeCount`
A value of type `ItemCount` that represents the number of text layout attributes being set.

`iTag` An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that corresponds to a text layout attribute. See "Text Layout and Line Attribute Tags" (page 250) for a description of Apple-defined tag values. Note that if you pass a style run attribute or an ATSUI-reserved tag value in this parameter, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeTagErr`. You cannot pass `NULL` for this parameter.

`iValueSize` An array of values of type `ByteCount` that represents the size (in bytes) of each text layout attribute value being set. You cannot pass `NULL` for this parameter. Note that if you pass an attribute value size that is less than required, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeSizeErr`.

`iValue` An array of pointers of type `ATSUAttributeValuePtr` (page 187). Each pointer in the array must reference a text layout attribute value that corresponds to a tag in the `iTag` array, and the value referenced by the pointer must be legal for that tag. Note that if

you pass a value that is invalid or undefined in this parameter, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeValueErr`. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUSetLayoutControls` function to set text layout attributes for a text layout object. Unset text layout attributes are assigned their default values listed in Table 2-2 (page 251).

Note that text layout attributes set for a particular line override those set for the entire text layout object, regardless of the order in which these attributes were set. For example, if you call the function `ATSUSetLineControls` (page 100) to set the line width of a particular line in a text layout object and then call `ATSUSetLayoutControls` to set the line width for the entire text layout object, the width of the particular line will have the value set in the call to `ATSUSetLineControls`.

ATSUI functions that operate on a line of text like `ATSUDrawText` (page 163), `ATSUMeasureText` (page 148), `ATSUMeasureTextImage` (page 153), and `ATSUGetGlyphBounds` (page 145) use text layout attributes set for the specified line of text. If no attributes have been set for the line, they use the text layout attributes set for the entire text layout object. If attributes values are not set in a line, ATSUI assigns the attribute values from the text layout object that the line is contained within.

If there is an error, `ATSUSetLayoutControls` will not set any text layout attributes in the line.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUSetLayoutControls` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetLayoutControl` (page 94)

`ATSUClearLayoutControls` (page 97)

`ATSUSetLineControls` (page 100)

ATSUGetLayoutControl

Obtains an attribute value from a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to a text layout object whose text layout attribute value you want to obtain. The text layout object can be uninitialized. You cannot pass `NULL` for this parameter.

`iTag` A value of type `ATSUAttributeTag`. Pass a valid tag that corresponds to the text layout attribute whose value you want to determine. See "Text Layout and Line Attribute Tags" (page 250) for a description of Apple-defined tag values. Note that if you pass a style run attribute or an ATSUI-reserved tag value in this parameter, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeTagErr`.

`iMaximumValueSize`
The size (in bytes) of the memory that you have allocated for the text layout attribute value. You can predetermine this value by first calling `ATSUGetLayoutControl (iTextLayout, iTag, 0, NULL, &oActualValueSize)`. Note that if you pass an attribute value size that is less than required, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeSizeErr`.

`oValue` A pointer of type `ATSUAttributeValuePtr` (page 187). On return, `oValue` points to the desired text layout attribute value. You can predetermine how much memory to allocate for this array by

first calling `ATSUGetLayoutControl (iTextLayout, iTag, 0, NULL, &oActualValueSize)`. Note that if the attribute value has not set, `ATSUGetLayoutControl` passes back its default value in this parameter and returns the result code `kATSUNotSetErr`.

`oActualValueSize`
A pointer to a value of type `ByteCount`. On return, the value represents the size (in bytes) of memory actually required for the attribute's value. This is useful if you are dealing with a custom attribute and don't know how much memory to allocate.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetLayoutControls` (page 92)

`ATSUGetAllLayoutControls` (page 95)

`ATSUGetLineControl` (page 102)

ATSUGetAllLayoutControls

Obtains all attribute values from a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to a text layout object whose text layout attribute values you want to obtain. The text layout object can be uninitialized. You cannot pass `NULL` for this parameter.

`oAttributeInfoArray`

An array of structures of type `ATSUAttributeInfo` (page 186). On return, each structure in the array contains a tag/value-size pair that corresponds to a particular text layout attribute value in the text layout object. You can predetermine how much memory to allocate for this array by first calling `ATSUGetAllLayoutControls (iTextLayout, NULL, 0, &oTagValuePairCount)`.

`iTagValuePairArraySize`

A value of type `ItemCount` that represents the number of `ATSUAttributeInfo` structures you want passed back in the `oAttributeInfoArray` array. You can predetermine this value by first calling `ATSUGetAllLayoutControls (iTextLayout, NULL, 0, &oTagValuePairCount)`.

`oTagValuePairCount`

A pointer to a value of type `ItemCount`. On return, the value represents the actual number of `ATSUAttributeInfo` structures corresponding to the number of text layout attributes set in the text layout object. This may be greater than the value passed in the `iTagValuePairArraySize` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

You can pass the tag and value size data passed back in the `oAttributeInfoArray` parameter to the `ATSUGetLayoutControl` (page 94) function to obtain the value of a particular text layout attribute.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetLayoutControl` (page 94)

`ATSUGetAllLineControls` (page 104)

ATSUClearLayoutControls

Removes previously set attribute values from a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to a text layout object whose previously set text layout attribute values you want to remove. The text layout object can be uninitialized. You cannot pass `NULL` for this parameter.

`iTagCount` A value of type `ItemCount` that represents the number of previously set text layout attribute values you want to remove. To remove all previously set text layout attribute values, pass the constant `kATSUClearAll`.

`iTag` An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that identifies a text layout attribute value you want to remove. See "Text Layout and Line Attribute Tags" (page 250) for a description of Apple-defined tag values. Note that if you pass a style run attribute or an ATSUI-reserved tag value in this parameter, `ATSUClearLayoutControls` returns the result code `kATSUInvalidAttributeTagErr`. If you pass the `kATSUClearAll` constant in the `iTagCount` parameter, the value in this parameter will be ignored.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUClearLayoutControls` function removes those previously set text layout attribute values from an entire text layout object that are identified by the tags in the `iTag` array. `ATSUClearLayoutControls` sets these values to the default values listed in Table 2-2 (page 251).

To remove all the previously set text layout attribute values from a text layout object, pass the constant `kATSUClearAll` in the `iTagCount` parameter. You can remove unset attribute values from a text layout object without a function error.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUClearLineControls` (page 105)

Manipulating Line Attributes

ATSUI provides the following functions for manipulating line attributes:

- `ATSUCopyLineControls` (page 98) copies both set and unset attribute values from one line to another within the same or different text layout object.
- `ATSUSetLineControls` (page 100) sets attribute values of a line within a text layout object.
- `ATSUGetLineControl` (page 102) obtains an attribute value from a line within a text layout object.
- `ATSUGetAllLineControls` (page 104) obtains all attribute values from a line within a text layout object.
- `ATSUClearLineControls` (page 105) removes previously set attribute values from a line within a text layout object.

ATSUCopyLineControls

Copies both set and unset attribute values from one line to another within the same or different text layout object.

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

`iSourceTextLayout`
A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the line whose attribute values you want to copy. You cannot pass `NULL` for this parameter.

`iSourceLineStart`
A value of type `UniCharArrayOffset` (page 198). Pass the edge offset of the beginning of the line whose attribute values you want to copy.

`iDestTextLayout`
A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the line whose attribute values you want to replace. This can be the same text layout object passed in the `iSourceTextLayout` parameter if you want to copy the attributes from a line into another line in the same text layout object. You cannot pass `NULL` for this parameter.

`iDestLineStart`
A value of type `UniCharArrayOffset` (page 198). Pass the edge offset of the beginning of the line whose attribute values you want to replace.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUCopyLineControls` function copies both previously set and unset attribute values from one line to another in the same or different text layout object, depending upon what you pass in the `iDestTextLayout` parameter.

`ATSUCopyLineControls` will not copy the contents of memory referenced by pointers or handles within reference constants. It is your responsibility to ensure that this memory remains valid until the source text layout object is disposed of.

SPECIAL CONSIDERATIONS

`ATSUCopyLineControls` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUCopyLayoutControls` (page 91)

ATSUSetLineControls

Sets attribute values of a line within a text layout object.

```
OSStatus ATSUSetLineControls (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iLineStart,
                    ItemCount iAttributeCount,
                    ATSUAttributeTag iTag[],
                    ByteCount iValueSize[],
                    ATSUAttributeValuePtr iValue[]);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object containing the line whose attribute values you want to set. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 198). Pass the edge offset of the beginning of the line whose attributes you want to set.

`iAttributeCount`
A value of type `ItemCount`. Pass the number of attributes you want to set for the line. This value should equal the number of tags passed in the `iTag` array.

`iTag` An array of values of type `ATSUAttributeTag`. Pass an array of tags that identify the attribute values you wish to set. Each element in the array must contain a valid tag that corresponds to a text layout attribute. See "Text Layout and Line Attribute Tags" (page 250) for a description of Apple-defined tag values. Note that if you pass a style run attribute or an ATSUI-reserved

tag value in this parameter, `ATSUSetLineControls` returns the result code `kATSUInvalidAttributeTagErr`. You cannot pass `NULL` for this parameter.

`iValueSize` An array of values of type `ByteCount`. Pass an array of the number of bytes that represent the size of each attribute you wish to set. Note that if you pass a size that is less than required, `ATSUSetLineControls` returns the result code `kATSUInvalidAttributeSizeErr`. You cannot pass `NULL` for this parameter.

`iValue` An array of pointers of type `ATSUAttributeValuePtr` (page 187). Pass an array of pointers that point to the attribute values you wish to set. Each pointer in the array must reference an attribute value that corresponds to a tag in the `iTag` array, and the value referenced by the pointer must be legal for that tag. Note that if you pass a value that is invalid or undefined in this parameter, `ATSUSetLineControls` returns the result code `kATSUInvalidAttributeValueErr`. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUSetLineControls` function sets attribute values for a single line within a text layout object. When you set the attributes of a line, these override those set in the text layout object the line is contained within. This is true regardless of the order in which these attributes were set. For example, if you first set the line width attribute value of a particular line to `.768` and then set the line width of the text layout object containing the line to `.34`, the line width of the particular line would be `.768`.

If you do not set an attribute value in a line, it will be set to the value of the corresponding attribute in the text layout object, whether it was previously set or is the default value listed in Table 2-2 (page 251).

The functions `ATSUDrawText` (page 163), `ATSUMeasureText` (page 148), `ATSUMeasureTextImage` (page 153), and `ATSUGetGlyphBounds` (page 145) use the text layout attributes set in the line they are operating on.

If there is an error, `ATSUSetLineControls` will not set any attributes in the line.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUSetLineControls` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUGetLineControl` (page 102)

`ATSUGetAllLineControls` (page 104)

`ATSUSetLayoutControls` (page 92)

ATSUGetLineControl

Obtains an attribute value from a line within a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference points to an initialized text layout object that contains the line whose attribute values you want to obtain. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 198). Pass the edge offset of the beginning of the line whose attribute value you want to obtain.

`iTag` A value of type `ATSUAttributeTag`. Pass a valid tag that corresponds to the attribute value you want to obtain. See "Text Layout and Line Attribute Tags" (page 250) for a description of

Apple-defined tag values. Note that if you pass a style run attribute or an ATSUI-reserved tag value in this parameter, `ATSUGetLineControl` returns the result code `kATSUInvalidAttributeTagErr`.

`iExpectedValueSize`
A value of type `ByteCount`. Pass the expected size (in bytes) of the attribute value. Attribute value sizes are listed in Note that if you pass a size that is less than required, `ATSUGetLineControl` returns the result code `kATSUInvalidAttributeSizeErr`.

`oValue` A pointer of type `ATSUAttributeValuePtr` (page 187). Before calling `ATSUGetLineControl`, pass a pointer to memory you have allocated for the attribute value or `NULL` if you don't know how big the attribute value will be. If you pass `NULL`, on return, `oValue` points to the desired text layout attribute value. Note that if you did not previously set the attribute value, `ATSUGetLineControl` passes back its default value in this parameter and returns the result code `kATSUNotSetErr`.

`oActualValueSize`
A pointer to a value of type `ByteCount`. On return, `oActualValueSize` points to actual size (in bytes) of the attribute value.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

You can use the tag and value size data passed back in the `oAttributeInfoArray` parameter of the function `ATSUGetAllLineControls` (page 104) in the `ATSUGetLineControl` function to obtain the value of the corresponding line attribute.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUSetLineControls` (page 100)

`ATSUGetLayoutControl` (page 94)

ATSUGetAllLineControls

Obtains all the text layout attribute values from a line in a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference points to an initialized text layout object that contains the line whose text layout attribute value you want to obtain. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 198) that represents the offset to the beginning of the line whose attribute values you want to obtain.

`oAttributeInfoArray`
An array of structures of type `ATSUAttributeInfo` (page 186). On return, each structure in the array contains a tag/value-size pair that corresponds to a text layout attribute value for a particular line in a text layout object. You can predetermine how much memory to allocate for this array by first calling `ATSUGetAllLineControls (iStyle, iLineStart, NULL, 0, &oTagValuePairCount)`.

`iTagValuePairArraySize`
A value of type `ItemCount` that represents the number of `ATSUAttributeInfo` structures you want passed back in the `oAttributeInfoArray` array. You can predetermine this value by first calling `ATSUGetAllLineControls (iStyle, iLineStart, NULL, 0, &oTagValuePairCount)`.

`oTagValuePairCount`
A pointer to a value of type `ItemCount`. On return, the value represents the actual number of `ATSUAttributeInfo` structures corresponding to the number of text layout attributes set in a single line of a text layout object. This may be greater than the value passed in the `iTagValuePairArraySize` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

You can pass the tag and value size data passed back in the `oAttributeInfoArray` parameter to the `ATSUGetLineControl` (page 102) function to obtain the value of a particular text layout attribute.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUGetAllLayoutControls` (page 95)

ATSUClearLineControls

Removes previously set text layout attribute values from a line in a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the line whose previously set text layout attribute values you want to remove. You cannot pass `NULL` for this parameter.

`iTagCount` A value of type `ItemCount` that represents the number of previously set text layout attributes you want to remove in the line. To remove all previously set text layout attribute values in the line, pass the constant `kATSUClearAll`.

`iTag` An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that identifies a text layout attribute value you want to remove. See "Text Layout and Line Attribute Tags" (page 250) for a description of Apple-defined tag

values. Note that if you pass a style run attribute or an ATSUI-reserved tag value in this parameter, `ATSUClearLineControls` returns the result code `kATSUInvalidAttributeTagErr`. If you pass the `kATSUClearAll` constant in the `iTagCount` parameter, the value in this parameter will be ignored.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUClearLineControls` function removes those previously set text layout attribute values from a single line in a text layout object that are identified by the tags in the `iTag` array. `ATSUClearLineControls` sets these values to the values set for the text layout object that the line is in. If these values were not set in the text layout object, `ATSUClearLayoutControls` sets them to the default values listed in Table 2-2 (page 251).

To remove all the previously set text layout attribute values from a line, pass the constant `kATSUClearAll` in the `iTagCount` parameter. You can remove unset attribute values from a line without a function error.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUClearLayoutControls` (page 97)

Determining and Updating Text Memory Location

ATSUI provides the following functions for determining and updating text memory location:

- `ATSUSetTextPointerLocation` (page 107) updates text accessed with a pointer in a text layout object.
- `ATSUSetTextHandleLocation` (page 109) updates text accessed with a handle in a text layout object.

- `ATSUGetTextLocation` (page 112) calculates the memory location of a text layout object's text buffer, how it is accessed, and the range of text to be operated on.
- `ATSUTextMoved` (page 113) updates text accessed with a handle in a text layout object.

ATSUSetTextPointerLocation

Updates text accessed with a pointer in a text layout object.

```
OSStatus ATSUSetTextPointerLocation (
                    ATSUTextLayout iTextLayout,
                    ConstUniCharArrayPtr iText,
                    UniCharArrayOffset iTextOffset,
                    UniCharCount iTextLength,
                    UniCharCount iTextTotalLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to a valid text layout object in which the user has moved, deleted, or inserted text. If the text layout object is uninitialized, `ATSUSetTextPointerLocation` will initialize it by assigning it the text pointed to by the `iText` parameter.

`iText` A pointer of type `ConstUniCharArrayPtr` (page 197). The pointer contains a text buffer. If the text layout object is uninitialized, `ATSUSetTextPointerLocation` assigns it to the text layout object. If the text layout object already has associated text, `ATSUSetTextPointerLocation` updates the text. Your application is responsible for allocating the memory associated with this pointer.

`iTextOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text that you want ATSUI to perform subsequent lay out operations on. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iTextLength` A value of type `UniCharCount` (page 198). The value represents the length of the range of text that you want to perform layout operations on. You can pass the constant `kATSUTexttoEnd`,

described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUSetTextPointerLocation` returns the result code `kATSUInvalidTextRangeErr`.

`iTextTotalLength`
A value of type `UniCharCount` (page 198). The value represents the length of the entire text buffer. Generally, this is greater than the range of text specified in the `iTextOffset` and `iTextLength` parameters, unless you want to perform layout operations on the entire text buffer. In this case, the value in this parameter should equal `iTextOffset + iTextLength`.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

You should call the `ATSUSetTextPointerLocation` function to update text before displaying it when

- the range of text specified in your text layout object spans the entire text buffer and
- you make changes to text in response to the user moving, deleting, or inserting text

Once you call `ATSUSetTextPointerLocation`, you can call the `ATSUDrawText` (page 163) function to display the updated text.

If the text layout object is uninitialized, the `ATSUSetTextPointerLocation` function assigns it the text buffer pointed to in `iText` the parameter. If the text layout object already has an assigned text buffer, `ATSUSetTextPointerLocation` updates it.

Note that the `ATSUSetTextPointerLocation` function clears drawing caches.

You specify the range of text you want to perform layout operations on in the `iTextOffset` and `iTextLength` parameters. To indicate the entire text buffer, pass the `kATSUFromTextBeginning` constant in the `iTextOffset` parameter and the `kATSUToTextEnd` constant in the `iTextLength` parameter.

If the range of text is a subset of the text buffer, the text layout object will scan the remaining text before and after the range of text in the text buffer to get the full context for bidirectional processing and other information about the text.

If you need to determine the memory location of the text layout object's text buffer, call the `ATSUGetTextLocation` (page 112) function. To update the memory location of the text layout object's text buffer, call the `ATSUTextMoved` (page 113) function.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUSetTextPointerLocation` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetTextHandleLocation` (page 109)

ATSUSetTextHandleLocation

Updates text accessed with a handle in a text layout object.

```
OSStatus ATSUSetTextHandleLocation (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayHandle iText,
                    UniCharArrayOffset iTextOffset,
                    UniCharCount iTextLength,
                    UniCharCount iTextTotalLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to a valid text layout object in which the user has moved, deleted, or inserted text. If the text layout object is uninitialized, `ATSUSetTextHandleLocation` will initialize it by assigning it the text pointed to by the `iText` parameter.

`iText` A handle of type `UniCharArrayHandle` (page 197). The handle contains the address of a text layout object's text buffer. The text layout object expects the buffer to contain a block of Unicode text. Your application is responsible for allocating the handle.

ATSUI functions will dereference the handle before accessing the text, but will leave the handle's state unchanged. If the text layout object is uninitialized, `ATSUSetTextPointerLocation` assigns it to the text layout object. If the text layout object already has associated text, `ATSUSetTextPointerLocation` updates the text.

`iTextOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text that you want to perform layout operations on. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iTextLength` A value of type `UniCharCount` (page 198). The value represents the length of the range of text that you want to perform layout operations on. You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUSetTextHandleLocation` returns the result code `kATSUInvalidTextRangeErr`.

`iTextTotalLength`
A value of type `UniCharCount` (page 198). The value represents the length of the entire text buffer. Generally, this is greater than the range of text specified in the `iTextOffset` and `iTextLength` parameters, unless you want to perform layout operations on the entire text buffer. In this case, the value in this parameter should equal `iTextOffset + iTextLength`.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

You should call the `ATSUSetTextHandleLocation` function to update text before displaying it when

- the range of text specified in your text layout object spans the entire text buffer and
- you make changes to text in response to the user moving, deleting, or inserting text

Once you call `ATSUSetTextHandleLocation`, you can call the `ATSUDrawText` (page 163) function to display the updated text.

If the text layout object is uninitialized, the `ATSUSetTextHandleLocation` function assigns it the text buffer pointed to in `iText` the parameter. If the text layout object already has an assigned text buffer, `ATSUSetTextHandleLocation` updates it.

Note that the `ATSUSetTextHandleLocation` function clears drawing caches.

You specify the range of text you want to perform layout operations on in the `iTextOffset` and `iTextLength` parameters. To indicate the entire text buffer, pass the `kATSUFromTextBeginning` constant in the `iTextOffset` parameter and the `kATSUToTextEnd` constant in the `iTextLength` parameter.

If the range of text is a subset of the text buffer, the text layout object will scan the remaining text before and after the range of text in the text buffer to get the full context for bidirectional processing and other information about the text.

If you need to determine the memory location of the text layout object's text buffer, call the `ATSUGetTextLocation` (page 112) function. To update the memory location of the text layout object's text buffer, call the `ATSUTextMoved` (page 113) function.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUSetTextHandleLocation` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetTextPointerLocation` (page 107)

ATSUGetTextLocation

Calculates the memory location of a text layout object's text buffer, how it is accessed, and the range of text to be operated on.

```
OSStatus ATSUGetTextLocation (
                    ATSUTextLayout iTextLayout,
                    void **oText,
                    Boolean *oTextIsStoredInHandle,
                    UniCharArrayOffset *oOffset,
                    UniCharCount *oTextLength,
                    UniCharCount *oTextTotalLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose text you want to determine.

`oText` A pointer of type `ConstUniCharArrayPtr` (page 197) or a handle of type `UniCharArrayHandle` (page 197). If the value passed back in the `oTextIsStoredInHandle` parameter is `true`, on return the handle contains the address of the text buffer that you want to assign to the text layout object. If the value passed back in the `oTextIsStoredInHandle` parameter is `false`, on return the pointer contains the text buffer that you want to assign to the text layout object.

`oTextIsStoredInHandle`
A pointer to a value of type `Boolean`. On return, the value indicates whether the text buffer in the `oText` parameter is accessed by handle or a pointer. If `true`, the text buffer is accessed by handle; if `false`, the text buffer is accessed by pointer.

`oOffset` A pointer to a value of type `UniCharArrayOffset` (page 198). On return, the value represents the edge offset of the beginning of the range of text to be laid out.

`oTextLength` A pointer to a value of type `UniCharCount` (page 198). On return, the value represents the length of the range of text to be laid out.

`oTextTotalLength`
A pointer to a value of type `UniCharCount` (page 198). On return, the value represents the length of the entire text buffer.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

ATSUTextMoved

Updates the location of text in physical memory.

```
OSStatus ATSUTextMoved (
                    ATSUTextLayout iTextLayout,
                    ConstUniCharArrayPtr iNewLocation);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose text memory location you have moved. You cannot pass `NULL` for this parameter.

`iNewLocation` A pointer of type `ConstUniCharArrayPtr` (page 197). The pointer contains the new base location for the text buffer in memory. The text layout object expects the buffer to contain a block of Unicode text. Your application is responsible for allocating the memory associated with this pointer.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUTextMoved` function updates the memory location of the text buffer associated with a text layout object to the location pointed to by the `iNewLocation` parameter. This parameter reflects the physical location in memory where you moved the text buffer to. Note that the text buffer should be otherwise unchanged. If, for example, the user has moved, deleted, or inserted text in the text layout object, you should instead call the `ATSUSetTextPointerLocation` (page 107) and `ATSUSetTextHandleLocation` (page 109) functions to update the text before displaying it. You are responsible for moving the text.

VERSION NOTES

Available with ATSUI 1.0.

Updating and Determining Style Runs

ATSUI provides the following functions for updating and determining style runs:

- `ATSUSetRunStyle` (page 114) overwrites style run information in a specified range of text.
- `ATSUGetRunStyle` (page 115) determines, for a specified edge offset, the previously set style run information and range of text that shares this information.
- `ATSUGetContinuousAttributes` (page 117) determines the style run information that is continuous for a specified range of text.

ATSUSetRunStyle

Overwrites style run information in a specified range of text.

```
OSStatus ATSUSetRunStyle (
                    ATSUTextLayout iTextLayout,
                    ATSUStyle iStyle,
                    UniCharArrayOffset iRunStart,
                    UniCharCount iRunLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to a valid text layout object whose style run information you want to update. If the text layout object is uninitialized, `ATSUSetRunStyle` will initialize it with the style runs pointed by the `iStyle` parameter.

`iStyle` A reference of type `ATSUStyle` (page 195). Pass a reference to the initialized style object whose style run information you want to copy. You cannot pass `NULL` for this parameter.

`iRunStart` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text whose style run information you want to replace. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iRunLength`	A value of type `UniCharCount` (page 198). The value represents the length of the range of text whose style run information you want to replace. You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUSetRunStyle` returns the result code `kATSUInvalidTextRangeErr`.
function result	A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUSetRunStyle` function completely overwrites the style run attribute, font feature, and font variation values of a specified range of text with those of the passed in style object. This includes values that are set in the specified range of text but not in the passed in style object. In this case, these values are set to their default values. After this call, ATSUI automatically adjusts the lengths of the style runs in the unaffected ranges of text.

You are responsible for disposing of the memory allocated for the new style run when you are done with the style run.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUSetRunStyle` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

ATSUGetRunStyle

Determines, for a specified edge offset, the previously set style run information the range of text that shares this information.

```
OSStatus ATSUGetRunStyle (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iOffset,
                    ATSUStyle *oStyle,
                    UniCharArrayOffset *oRunStart,
                    UniCharCount *oRunLength);
```

`iTextLayout`	A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose style run information over a specific range of text you want to determine. You cannot pass `NULL` for this parameter.
`iOffset`	A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset whose continuous style run information you want to determine. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer. Note that if the offset is outside the text layout object's text buffer, `ATSUGetRunStyle` returns the result code `kATSUInvalidTextRangeErr`.
`oStyle`	A pointer to a reference of type `ATSUStyle` (page 195). On return, `oStyle` points to the previously set style run information at the location specified in the `iOffset` parameter. If no style runs have been set, this value will be `NULL`.
`oRunStart`	A pointer to a value of type `UniCharArrayOffset` (page 198). On return, the value represents the edge offset of the beginning of the range of text that shares the style run information specified in the `oStyle` parameter.
`oRunLength`	A pointer to a value of type `UniCharCount` (page 198). On return, the value represents the length of the range of text that shares the style run information specified in the `oStyle` parameter.
function result	A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUGetRunStyle` function passes back the previously set style run information for a specific text location, and the range of text that shares this information. Unlike `ATSUGetContinuousAttributes` (page 117), `ATSUGetRunStyle` returns style information for a specific text location, not for a range of text.

If there is only one style run in the specified text layout object, `ATSUGetRunStyle` passes back both set and unset style run information for the specified location, and sets the entire text layout object's style information to that of the specified location.

The value passed back in the `oRunStart` parameter represents the edge offset of the beginning of the style run. You can use this value to determine whether you

are at a style run boundary. At a style run boundary, `ATSUGetRunStyle` passes back the style run that follows the boundary.

If you want to determine the style run information that is continuous for a specified range of text, call the `ATSUGetContinuousAttributes` (page 117) function.

ATSUGetContinuousAttributes

Determines the style run information that is continuous for a specified range of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose continuous style run information over a specific range of text you want to obtain. You cannot pass `NULL` for this parameter.

`iOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text whose continuous style run information you want to determine. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iRunLength` A value of type `UniCharCount` (page 198). The value represents the length of the range of text whose continuous style run information you want to determine. You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUGetContinuousAttributes` returns the result code `kATSUInvalidTextRangeErr`.

`oStyle` A reference of type `ATSUStyle` (page 195). Before calling `ATSUGetContinuousAttributes`, allocate enough memory to contain the continuous style information for the range of text.

On return, `oStyle` points to the continuous style run information that is shared by the specified text range. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUGetContinuousAttributes` function passes back the previously set style run attribute, font feature, and font variation values for a specific range of text. Any unset style run attribute, font feature, or font variation values that are continuous over the text range are set to their default values and passed back to indicate that these values were continuous.

Unlike `ATSUGetRunStyle` (page 115), `ATSUGetContinuousAttributes` returns style information for a range of text, not for a specific text location.

The value passed back in the `oRunStart` parameter represents the edge offset of the beginning of the style run. You can use this value to determine whether you are at a style run boundary. At a style run boundary, `ATSUGetRunStyle` passes back the style run that follows the boundary.

You should call `ATSUGetContinuousAttributes` to determine the style run information that remains constant over text that has been selected by the user. For example, the user might select the entire text block associated with a text layout object or a portion of it, then choose a different font family from your menu to render the text. `ATSUGetContinuousAttributes` will determine whether the style is plain, boldfaced, italicized, underlined, condensed, or extended.

If you want to determine the previously set style run information and range of text that shares this information for a specific location in the text, call the `ATSUGetRunStyle` (page 115) function.

Providing Font Substitutions

ATSUI provides the following functions for providing font substitutions in a text layout object:

- `ATSUSetFontFallbacks` (page 119) establishes the search options and the fonts to search when a font does not have all the glyphs for the characters it is trying to draw.

- `ATSUGetFontFallbacks` (page 120) obtains the previously set search order and the fonts to search when a font does not have all the glyphs for the characters it is trying to draw.
- `ATSUMatchFontsToText` (page 122) sequentially scans the valid fonts on a user's system to find the best substitute font.
- `ATSUSetTransientFontMatching` (page 124) enables automatic font substitution in a text layout object.
- `ATSUGetTransientFontMatching` (page 125) determines whether automatic font substitution has been enabled in a text layout object.

ATSUSetFontFallbacks

Establishes the search options and the fonts to search when a font does not have all the glyphs for the characters it is trying to draw.

```
OSStatus ATSUSetFontFallbacks (
                    ItemCount iFontFallbacksCount,
                    ATSUFontID iFontIDs[],
                    ATSUFontFallbackMethod iFontFallbackMethod);
```

`iFontFallbacksCount`
A value of type `ItemCount` that represents the number of fonts in the `iFontIDs` array.

`iFontIDs`
An array of values of type `ATSUFontID` (page 192). The array contains a list of the fonts to be sequentially searched using the search options specifeied in the `iFontFallbackMethod` parameter. The first valid font in the list that can perform the substitution will be used.

`iFontFallbackMethod`
A value of type `ATSUFontFallbackMethod`. The value represents the search options to employ when a font does not have all the glyphs for the characters it is trying to draw. See "Font Fallback Constants" (page 208) for a description of possible values. To specify the default search behavior, pass the constant `kATSUDefaultFontFallbacks`.

function result A result code. The result code `kATSUInvalidFontErr` indicates that the font does not correspond to any installed font. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUSetFontFallbacks` function enables you to customize the search options and the fonts to search when a font does not have all the glyphs for the characters it is trying to draw. If you do not call `ATSUSetFontFallbacks`, all valid fonts on the user's system will be searched when a substitute font is needed. You can change back to this default search behavior by passing the `kATSUDefaultFontFallbacks` constant in the `iFontFallbackMethod` parameter. Specifying the default search behavior is equivalent to calling the `ATSUMatchFontsToText` (page 122) function.

`ATSUSetFontFallbacks` enables you to specify the fonts that ATSUI should use for fallback fonts. By judiciously ordering the chosen fallbacks, the amount of time needed by ATSUI to find a suitable fallback for specific text can be significantly reduced.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUGetFontFallbacks` (page 120)

ATSUGetFontFallbacks

Obtains the previously set search options and the fonts to search when a font does not have all the glyphs for the characters it is trying to draw.

```
OSStatus ATSUGetFontFallbacks (
                    ItemCount iMaxFontFallbacksCount,
                    ATSUFontID oFontIDs[],
                    ATSUFontFallbackMethod *oFontFallbackMethod,
                    ItemCount *oActualFallbacksCount);
```

`iMaxFontFallbacksCount`
A value of type `ItemCount`. This value represents the number of fonts that you want passed back in the `iFontIDs` array. You can predetermine this value by first calling `ATSUGetFontFallbacks (0, NULL, NULL, &oActualFallbacksCount)`.

`oFontIDs` An array of values of type `ATSUFontID` (page 192). On return, the array contains the previously set list of the fonts to be sequentially searched using the search options specified in the `oFontFallbackMethod` parameter. You set this value by calling `ATSUSetTransientFontMatching` (page 124).You can predetermine how much memory to allocate for this array by first calling `ATSUGetFontFallbacks (0, NULL, NULL, &oActualFallbacksCount)`.

`oFontFallbackMethod`
A pointer to a value of type `ATSUFontFallbackMethod`. On return, the value represents the previously set search options to employ when a font does not have all the glyphs for the characters it is trying to draw. You set this value by calling `ATSUSetTransientFontMatching` (page 124). See "Font Fallback Constants" (page 208) for a description of possible values.

`oActualFallbacksCount`
A pointer to a value of type `ItemCount`. On return, the value represents the actual number of fallback fonts set in the text layout object. This may be greater than the value passed in the `iMaxFontFallbacksCount` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUSetFontFallbacks` (page 119)

ATSUMatchFontsToText

Sequentially scans the valid fonts on a user's system to find the best substitute font.

```
OSStatus ATSUMatchFontsToText (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iTextStart,
                    UniCharCount iTextLength,
                    ATSUFontID *oFont,
                    UniCharArrayOffset *oChangedOffset,
                    UniCharCount *oChangedLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains for which you want to find a substitute font. You cannot pass `NULL` for this parameter.

`iTextStart` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text for which you want to find a substitute font. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text buffer.

`iTextLength` A value of type `UniCharCount` (page 198). This value represents the length of the range of text for which you want to find a substitute font. You can pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 255), to represent the end of the text buffer. Note that if any part of the specified text range is outside the text layout object, `ATSUMatchFontsToText` returns the result code `kATSUInvalidTextRangeErr`.

`oFont` A pointer to a value of type `ATSUFontID` (page 192). On return, `oFont` points to the ID of the suggested font to use for the first character that could not be drawn.

`oChangedOffset` A pointer to a value of type `UniCharArrayOffset` (page 198). On return, the value represents the first edge offset containing a character whose font does not have all the glyphs to draw it.

`oChangedLength` A pointer to a value of type `UniCharCount` (page 198). On return, the value represents the length of the first range of text containing a character whose font does not have all the glyphs to draw it.

function result A result code. The result code `noErr` indicates that all the characters can be rendered with their currently assigned fonts. The result code `kATSUFontsMatched` indicates that at least one character could not be rendered with its currently assigned font. In this case, all the characters in the specified text range cannot be drawn with their currently assigned font, but can be drawn with the font passed back in the `oFont` parameter. The result code `kATSUFontsNotMatched` indicates that at least one character could not be rendered with its currently assigned font or with any currently active font. In this case, all the characters in the specified text range can only be rendered by the last resort font, and the value of `oFont` is set to `kATSUInvalidFontID`. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUMatchFontsToText` function scans all valid fonts on the user's system to find the best substitute font. Unlike `ATSUSetTransientFontMatching` (page 124), however, it does not perform the actual font substitution, but simply passes back the recommended font and the first subrange of text whose character(s) could not be drawn with the assigned font.

If you wish to specify the fonts that ATSUI should use for fallback fonts, and the order in which to search these fonts, you should instead call the `ATSUSetFontFallbacks` (page 119) function.

`ATSUMatchFontsToText` looks for characters in a specified range of text that cannot be drawn with the fonts in the style run. It passes back an offset to the first range of text that could not be drawn and suggests an alternative font to use. 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 (that is, the font in the styles for this text layout object) but could be drawn with font F, and the character 'e' could either be drawn with the current font or could not be drawn with font F, then `ATSUMatchFontsToText` will pass back the `ATSUIFontID` of font F in the `oFont` parameter and will set `oChangedOffset` to 2 and `oChangedLength` to 2.

If the function returns the result codes `kATSUFontsMatched` or `kATSUFontsNotMatched`, you should update the input range and call `ATSUMatchFontsToText` again to make sure that all the characters in the range can be drawn.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUMatchFontsToText` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetTransientFontMatching` (page 124)

`ATSUGetTransientFontMatching` (page 125)

ATSUSetTransientFontMatching

Enables automatic font substitution in a text layout object.

```
OSStatus ATSUSetTransientFontMatching (
                    ATSUTextLayout iTextLayout,
                    Boolean iTransientFontMatching);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object for which you want to enable automatic font substitution. You cannot pass `NULL` for this parameter.

`iTransientFontMatching`
A value of type `Boolean` that indicates whether you want ATSUI to perform automatic font substitution. Pass `true` if you want automatic font substitution.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUSetTransientFontMatching` function tells ATSUI to automatically substitute the best font it finds for a particular character. Upon substitution, `ATSUSetTransientFontMatching` does not change the font style run attribute value in the style object. As a result, if you plan to redraw a text layout object, you should instead call the `ATSUSetFontFallbacks` (page 119) or `ATSUMatchFontsToText` (page 122) function.

If you want to ensure that the last resort font will be used if no other fonts are found, you can either call `ATSUSetTransientFontMatching` or pass the `kATSUSequentialFallbacksExclusive` constant in the `iFontFallbackMethod` parameter of `ATSUSetFontFallbacks`. If you do not set the last resort font, glyphs will be denoted by black boxes when a font is not installed on the user's system.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUMatchFontsToText` (page 122)

`ATSUGetTransientFontMatching` (page 125)

ATSUGetTransientFontMatching

Determines whether automatic font substitution has been enabled in a text layout object.

```
OSStatus ATSUGetTransientFontMatching (
                    ATSUTextLayout iTextLayout,
                    Boolean *oTransientFontMatching);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object for which you want to determine whether automatic font substitution has been enabled. You cannot pass `NULL` for this parameter.

`oTransientFontMatching`
A pointer to a value of type `Boolean`. On return, the value indicates whether automatic font substitution is enabled in the specified text layout object. If `true`, it is enabled.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUMatchFontsToText` (page 122)

`ATSUSetTransientFontMatching` (page 124)

Functions for Responding to User Actions

This section describes the functions you can use to respond to actions like text insertion and deletion, cursor movement, line breaking, and highlighting:

- "Hit-Testing" (page 126)
- "Obtaining Cursor Offsets" (page 134)
- "Deleting and Inserting Text" (page 141)
- "Measuring Typographic and Image Bounds" (page 145)
- "Manipulating Line Breaks" (page 156)
- "Drawing Text" (page 162)
- "Highlighting and Unhighlighting Text" (page 165)
- "Performing Background Processing" (page 173)

Hit-Testing

ATSUI provides the following functions for hit-testing:

- `ATSUPositionToOffset` (page 127) obtains the surrounding edge offset(s) of the character whose onscreen glyph or graphic character is nearest a mouse-down event.
- `ATSUOffsetToPosition` (page 131) obtains the caret position(s) corresponding to an edge offset.

ATSUPositionToOffset

Obtains the surrounding edge offset(s) and text direction of the character whose onscreen glyph or graphic character is nearest a mouse-down event.

```
OSStatus ATSUPositionToOffset (
                    ATSUTextLayout iTextLayout,
                    ATSUTextMeasurement iLocationX,
                    ATSUTextMeasurement iLocationY,
                    UniCharArrayOffset *ioPrimaryOffset,
                    Boolean *oIsLeading,
                    UniCharArrayOffset *oSecondaryOffset);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the character whose onscreen glyph or graphic character is nearest the mouse-down event. You cannot pass `NULL` for this parameter.

`iLocationX` A value of type `ATSUTextMeasurement` (page 196) that represents the x-coordinate of the position of the hit point relative to the position of the origin of the line in the current graphics port in which the hit occurred. To determine the position of the hit point relative to the graphics port position of the origin of the line in which the hit occurred, see the discussion below. If you want the position of the hit relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iLocationY` A value of type `ATSUTextMeasurement` (page 196) that represents the y-coordinate of the position of the hit point relative to the position of the origin of the line in the current graphics port in which the hit occurred. To determine the position of the hit point relative to the graphics port position of the origin of the line in which the hit occurred, see the discussion below. If you

want the position of the hit relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`ioPrimaryOffset`

A pointer to a value of type `UniCharArrayOffset` (page 198). Pass a pointer to the edge offset of the beginning of the line that contains the character whose onscreen glyph or graphic character is nearest the mouse-down event. If the line direction is right-to-left, the offset of the beginning of the line corresponds to the lowest edge offset. On return, the value represents the primary edge offset relative to the origin of the line in which the hit occurred that corresponds to the closest edge of the glyph beneath the hit point on the screen. See the discussion below for more information. You cannot pass `NULL` for this parameter.

`oIsLeading` A pointer to a value of type `Boolean`. On return, the value indicates whether the edge offset passed back in the `ioPrimaryOffset` parameter is leading or trailing. If `true`, the primary offset is more closely associated with the following character in the backing-store memory. If `false`, the primary offset is more closely associated with the previous character in the backing-store memory. See the discussion below for more information.

`oSecondaryOffset`

A pointer to a value of type `UniCharArrayOffset` (page 198). On return, if the hit point occurs on a line direction boundary, the value represents the edge offset relative to the origin of the line in which the hit occurred that corresponds to the furthest edge of the glyph beneath the hit point. See the discussion below for more information.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUPositionToOffset` function determines the edge offset(s) in backing-store memory and the text direction of the character whose onscreen glyph or graphics character is nearest a mouse-down event. If the hit occurred on a line direction boundary, it passes back a second edge offset in the `oSecondaryOffset` parameter. A line direction boundary can occur on the trailing edges of two glyphs, the leading edges of two glyphs, or at the beginning or end of a text segment. The primary offset (also known as the high caret offset) corresponds to the closest edge of the glyph beneath the hit point. The secondary offset (also known as the low caret offset) corresponds to the furthest edge of the glyph beneath the hit point.

The user expects that pressing the mouse button correlates to particular actions in an application. You can use the offset(s) passed back in `ATSUPositionToOffset` for providing feedback or performing any actions in response to the user.

For example, if the user presses the mouse button in text, your application should pass the resulting edge offset to `ATSUOffsetToPosition` (page 131) to determine the caret position(s) corresponding to this offset. If the user presses the mouse button while the cursor is on a glyph and drags the cursor across a range of text, then releases the mouse, your application might want to respond by highlighting the text between the mouse-down and mouse-up events. To do this, you would pass the edge offset (s) passed back from `ATSUPositionToOffset` that correspond to the mouse-up and mouse-down event positions to the `ATSUHighlightText` (page 165) function.

If the user then presses the mouse button outside the highlighted area, your application should pass the same edge offsets to the `ATSUUnhighlightText` (page 168) function. If the user double clicks (word selection) or triple clicks (paragraph selection), You can pass the resulting primary edge offset to `ATSUOffsetToPosition` (page 131) to determine the caret position(s) corresponding to this offset.

ATSUI does not keep actual line positions. As a result, the coordinates passed in the `iLocationX` and `iLocationY` parameters are relative to the position in the current graphics port of the origin of the line in which the mouse-down occurred. The passed back back edge offset(s) are thus offsets from the beginning of the line in which the hit occurred, not from the beginning of the text layout object's buffer.

To tranform the hit point's position, you must first call the `GlobalToLocal` function, described in "Basic QuickDraw" in *Inside Macintosh: Imaging with QuickDraw*, to translate the global coordinates passed back in the `where` field of

the event record to local coordinates. For more information about responding to mouse-down events, see the "The Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. You then subtract the hit point (in local coordinates) from the position of the line's origin in the current graphics port.

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 hit that you would pass in the `iLocationX` and `iLocationY` parameters would be (25,0).

`ATSUPositionToOffset` passes back a `Boolean` value in the `oIsLeading` parameter to tell you the text direction of the primary edge offset. This directionality is determined by the Unicode directionality of the original character in backing-store memory. If it passes back `true`, the primary edge offset is more closely associated with the following character in the backing-store memory. If it passes back `false`, the primary offset is more closely associated with the previous character in the backing-store memory.

The following summarizes the possible outcomes of calling `ATSUPositionToOffset`:

- When the input pixel location (that is, the location of the hit point on the screen) is on the leading edge of the glyph, `ATSUPositionToOffset` passes back primary and secondary offsets corresponding to that glyph and an `oIsLeading` flag of `true`. If the glyph represents multiple characters and the style run attribute corresponding to the `kATSUNoLigatureSplitTag` has been set for them, `ATSUPositionToOffset` passes back an edge offset representing the beginning of this group of characters in memory.
- When the input pixel location is on the trailing edge of the glyph, `ATSUPositionToOffset` passes back primary and secondary offsets representing the ending of this group of characters in memory following the character or characters represented by the glyph and an `oIsLeading` flag of `false`.
- When the input pixel location is beyond the leftmost or rightmost caret positions (not taking into account line rotation) such that no glyph corresponds to the location of the hit, `ATSUPositionToOffset` passes back the primary edge 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 the input location is closest to. In this case, the secondary offset is equal to the primary offset, since no glyph was hit.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUPositionToOffset` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

ATSUOffsetToPosition

Obtains the caret position(s) corresponding to a specific edge offset.

```
OSStatus ATSUOffsetToPosition (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iOffset,
                    Boolean iIsLeading,
                    ATSUCaret *oMainCaret,
                    ATSUCaret *oSecondCaret,
                    Boolean *oCaretIsSplit);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the text range whose caret position you want to determine. You cannot pass `NULL` for this parameter.

`iOffset` A pointer to a value of type `UniCharArrayOffset` (page 198) that represents the edge offset whose caret position(s) you wish to determine. To visually respond to a mouse-down event, pass the offset passed back in the `ioPrimaryOffset` parameter of `ATSUPositionToOffset` (page 127).

`iIsLeading` A `Boolean` value that is only relevant if the specific edge offset occurs at a line direction boundary. To determine whether the offset occurs at a line direction boundary, evaluate the `Boolean` value passed back in the `oCaretIsSplit` parameter. If `true`, the edge offset is on a line direction boundary. Pass `true` in this parameter if the edge offset corresponds to the first offset of the line. Pass `false` if the edge offset is the last offset of the line. In this case, the last offset has a value equal to the sum of the starting edge offset and line length.

`oMainCaret` A pointer to a structure of type `ATSUCaret` (page 188). On return, if the value passed back from `oCaretIsSplit` is `true`, the structure contains the starting and ending pen locations of the high caret. If the passed back value is `false`, the structure contains the starting and ending pen locations of the main caret.

`oSecondCaret` A pointer to a structure of type `ATSUCaret` (page 188). On return, if the value passed back from `oCaretIsSplit` is `true`, the structure contains the starting and ending pen locations of the low caret. If the passed back value is `false`, the structure contains the starting and ending pen locations of the main caret contained in the `oMainCaret` parameter.

`oCaretIsSplit` A pointer to a value of type `Boolean`. On return, the value indicates whether the specific edge offset occurs at a line direction boundary. If `true`, the offset occurs at a line direction boundary. In this case, the `oSecondCaret` parameter contains the starting and ending locations of the low caret. If `false`, the offset does not occur at a line direction boundary. In this case, the `oSecondCaret` parameter contains the starting and ending pen locations of the high caret contained in the `oMainCaret` parameter.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The user expects that pressing the mouse button correlates to the display of a caret in text. Your application should call the `ATSUOffsetToPosition` function to find the caret position(s) corresponding to a mouse-down event. To determine caret position(s), you must first pass the hit point relative to the graphics port position of the origin of the line in which the hit occurred to the `ATSUPositionToOffset` (page 127) function. `ATSUPositionToOffset` passes back the edge offset (or offsets, if it falls on a line direction boundary), from the beginning of the line that contains the character whose onscreen glyph or

graphic character is nearest the mouse-down event. Note that the passed back offset is relative to the beginning of the line in which the hit occurred, not from the beginning of the text layout object's buffer.

You should pass the edge offset passed back in the `ioPrimaryOffset` parameter of `ATSUPositionToOffset` into the `iOffset` parameter of `ATSUOffsetToPosition`.

`ATSUPositionToOffset` passes back a `Boolean` value in the `oCaretIsSplit` parameter that indicates whether the edge offset occurs at a line direction boundary. If `true`, the offset occurs at a line direction boundary. In this case, pass a `Boolean` value in the `iIsLeading` parameter indicating whether the edge offset corresponds to the first offset (`true`) or last offset (`false`) of the line. The `oSecondCaret` parameter will contain the starting and ending locations of the low caret.

If `false`, the edge offset does not occur at a line direction boundary. In this case, the `oSecondCaret` parameter contains the starting and ending pen locations of the caret contained in the `oMainCaret` parameter.

Because the edge offset passed back from `ATSUPositionToOffset` is relative to the beginning of the line in which the hit occurred, you must transform the starting and ending pen locations associated with the caret passed back from `ATSUOffsetToPosition` to their actual coordinates in the current graphics port. To do this, you must add the value of the passed back coordinates to the coordinates of the position of the line's origin in the current graphics port.

For example, if you have a straight caret whose starting and ending location (relative to the line origin) is (25,0), you would add this value to 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 actual position that you would draw the caret in the current graphics port would be (75,50).

To draw the caret, you could call `MoveTo(fX, fY)` and `LineTo(fDeltaX, fDeltaY)`, or equivalent functions. The `MoveTo` and `LineTo` functions are described in "Basic QuickDraw" in *Inside Macintosh: Imaging with QuickDraw*. Note that the passed back caret structure(s) contain the positions needed to draw the carets on angled lines and reflect angled carets and leading/trailing split caret appearances.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUOffsetToPosition` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

Obtaining Cursor Offsets

ATSUI provides the following functions for obtaining cursor offsets:

- `ATSUNextCursorPosition` (page 134) obtains the previous edge offset for a specified offset based on cursor movement.
- `ATSUPreviousCursorPosition` (page 136) obtains the previous edge offset for a specified offset based on cursor movement.
- `ATSURightwardCursorPosition` (page 138) obtains the edge offset to the right of the high caret position for a specified offset based on cursor movement.
- `ATSULeftwardCursorPosition` (page 140) obtains the edge offset to the left of the high caret position for a specified offset based on cursor movement.

ATSUNextCursorPosition

Obtains the next edge offset for a specified offset based on cursor movement.

```
OSStatus ATSUNextCursorPosition (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iOldOffset,
                    ATSUCursorMovementType iMovementType,
                    UniCharArrayOffset *oNewOffset);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the cursor whose offset you want to obtain. You cannot pass `NULL` for this parameter.

`iOldOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the initial cursor position. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer. Note that if the offset is outside the text layout object's range of text, `ATSUNextCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.

`iMovementType`	A value of type `ATSUCursorMovementType` that represents the unit distance that the cursor has moved. See "Cursor Movement Constants" (page 207) for a description of possible values.
`oNewOffset`	A pointer to a value of type `UniCharArrayOffset` (page 198). On return, the value represents the edge offset corresponding to the new cursor position. This can be outside the text layout object's specified range of text.
function result	A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUNextCursorPosition` function obtains the next edge offset depending upon the type of cursor movement (by character, cluster, or word). The distances corresponding to these movement types are described in "Cursor Movement Constants" (page 207). Note that because of surrogate pairs, you can not always move the cursor by one character, since doing so might place the cursor in the middle of a surrogate pair. For more information on surrogate pairs, see "ATSUI Implementation of the Unicode Specification" (page 265).

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUNextCursorPosition` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUPreviousCursorPosition` (page 136)

ATSURightwardCursorPosition (page 138)

ATSULeftwardCursorPosition (page 140)

ATSUPreviousCursorPosition

Obtains the previous edge offset for a specified offset based on cursor movement.

```
OSStatus ATSUPreviousCursorPosition (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iOldOffset,
                    ATSUCursorMovementType iMovementType,
                    UniCharArrayOffset *oNewOffset);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the cursor whose offset you want to obtain. You cannot pass `NULL` for this parameter.

`iOldOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the initial cursor position. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer. Note that if the offset is outside the text layout object's range of text, `ATSUPreviousCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.

`iMovementType` A value of type `ATSUCursorMovementType` that represents the unit distance that the cursor has moved. See "Cursor Movement Constants" (page 207) for a description of possible values. Values must be at least one character (2 bytes) and no more than a word in length.

`oNewOffset` A pointer to a value of type `UniCharArrayOffset` (page 198). On return, the value represents the edge offset corresponding to the new cursor position. This can be outside the text layout object's specified range of text.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

You can call the `ATSUPreviousCursorPosition` function to obtain the next edge offset when the cursor is moved one character, cluster, or word. The distances corresponding to these movement types are described in "Cursor Movement Constants" (page 207). Note that because of surrogate pairs, you can not always move the cursor by one character, since doing so might place the cursor in the middle of a surrogate pair. For more information on surrogate pairs, see "ATSUI Implementation of the Unicode Specification" (page 265).

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUPreviousCursorPosition` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUNextCursorPosition` (page 134)

`ATSURightwardCursorPosition` (page 138)

`ATSULeftwardCursorPosition` (page 140)

ATSURightwardCursorPosition

Obtains the edge offset to the right of the high caret position for a specified offset based on cursor movement.

```
OSStatus ATSURightwardCursorPosition (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iOldOffset,
                    ATSUCursorMovementType iMovementType,
                    UniCharArrayOffset *oNewOffset);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the cursor whose offset you want to obtain. You cannot pass `NULL` for this parameter.

`iOldOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the initial cursor position. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer. Note that if the offset is outside the text layout object's range of text, `ATSURightwardCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.

`iMovementType` A value of type `ATSUCursorMovementType` that represents the unit distance that the cursor has moved. See "Cursor Movement Constants" (page 207) for a description of possible values. Values must be at least one character (2 bytes) and no more than a word in length.

`oNewOffset` A pointer to a value of type `UniCharArrayOffset` (page 198). On return, the value represents the edge offset corresponding to the new cursor position. This can be outside the text layout object's specified range of text.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

You can call the `ATSURightwardCursorPosition` function to the edge offset to the right of the high caret position for a specific offset when the use moves the cursor rightward on the screen by one character, cluster, or word. The distances corresponding to these movement types are described in "Cursor Movement Constants" (page 207). Note that because of surrogate pairs, you can not always move the cursor by one character, since doing so might place the cursor in the middle of a surrogate pair. For more information on surrogate pairs, see "ATSUI Implementation of the Unicode Specification" (page 265).

Except in the case of Indic text (and other cases where the font rearranges the glyphs), for left-to-right text, `ATSURightwardCursorPosition` has the same effect as calling `ATSUNextCursorPosition` (page 134). For right-to-left text, `ATSURightwardCursorPosition` has the same effect as calling `ATSUPreviousCursorPosition` (page 136).

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSURightwardCursorPosition` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUNextCursorPosition` (page 134)

`ATSUPreviousCursorPosition` (page 136)

`ATSULeftwardCursorPosition` (page 140)

ATSULeftwardCursorPosition

Obtains the edge offset to the left of the high caret position for a specified offset based on cursor movement.

```
OSStatus ATSULeftwardCursorPosition (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iOldOffset,
                    ATSUCursorMovementType iMovementType,
                    UniCharArrayOffset *oNewOffset);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the cursor whose offset you want to obtain. You cannot pass `NULL` for this parameter.

`iOldOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the initial cursor position. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer. Note that if the offset is outside the text layout object's range of text, `ATSULeftwardCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.

`iMovementType` A value of type `ATSUCursorMovementType` that represents the unit distance that the cursor has moved. See "Cursor Movement Constants" (page 207) for a description of possible values. Values must be at least one character (2 bytes) and no more than a word in length.

`oNewOffset` A pointer to a value of type `UniCharArrayOffset` (page 198). On return, the value represents the edge offset corresponding to the new cursor position. This can be outside the text layout object's specified range of text.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

You can call the `ATSULeftwardCursorPosition` function to the edge offset to the right of the high caret position for a specific offset when the use moves the cursor leftward on the screen by one character, cluster, or word. The distances corresponding to these movement types are described in "Cursor Movement Constants" (page 207). Note that because of surrogate pairs, you can not always move the cursor by one character, since doing so might place the cursor in the middle of a surrogate pair. For more information on surrogate pairs, see "ATSUI Implementation of the Unicode Specification" (page 265).

Except in the case of Indic text (and other cases where the font rearranges glyphs), for left-to-right text, `ATSULeftwardCursorPosition` has the same effect as calling `ATSUPreviousCursorPosition` (page 136). For right-to-left text, `ATSULeftwardCursorPosition` has the same effect as calling `ATSUNextCursorPosition` (page 134).

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSULeftwardCursorPosition` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUNextCursorPosition` (page 134)

`ATSUPreviousCursorPosition` (page 136)

`ATSURightwardCursorPosition` (page 138)

Deleting and Inserting Text

ATSUI provides the following functions for deleting and inserting text:

- `ATSUTextDeleted` (page 142) indicates to a text layout object that text has been deleted.

- `ATSUTextInserted` (page 143) indicates to a text layout object that text has been inserted.

ATSUTextDeleted

Indicates to a text layout object that text has been deleted.

```
OSStatus ATSUTextDeleted (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iDeletedRangeStart,
                    UniCharCount iDeletedRangeLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object in which text has been deleted. You cannot pass `NULL` for this parameter.

`iDeletedRangeStart`
A value of type `UniCharArrayOffset` (page 198) that represents the beginning location in the text layout object where text has been deleted. This can be an offset outside the text layout object's text range, since you will want to alert the text layout object of deletions that occur before or after this subrange. Text preceding the deleted text will be shifted by the appropriate offset (`iDeletedRangeStart` +`iDeletedRangeLength`). You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iDeletedRangeLength`
A value of type `UniCharCount` (page 198) that represents the length of the deleted text. This range can extend beyond the text layout object's own range of text, since you will want to alert the text layout object of deletions that occur before or after this subrange. Text preceding the deleted text will be shifted by the appropriate offset (`iDeletedRangeStart` +`iDeletedRangeLength`). You can pass the constant `kATSUtexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUTextDeleted` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUTextDeleted` function to indicate to a text layout object that text has been deleted. `ATSUTextDeleted` removes any style runs or soft line breaks in the text layout object's text range that have been specified to be deleted.

When you call `ATSUTextDeleted`, drawing caches are updated and style runs are removed as appropriate. You are responsible for making sure that the corresponding text is also removed from the text buffer. `ATSUTextDeleted` does not dispose of the memory used by the removed style objects. You are responsible for doing so.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUTextInserted` (page 143)

ATSUTextInserted

Indicates to a text layout object that text has been inserted.

```
OSStatus ATSUTextInserted (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iInsertionLocation,
                    UniCharCount iInsertionLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object within which text has been inserted. You cannot pass `NULL` for this parameter.

`iInsertionLocation`
A value of type `UniCharArrayOffset` (page 198) that represents the beginning location of inserted text. This can be an offset outside the text layout object's text range, since you will want to

alert the text layout object of insertions that occur before or after this subrange. Text preceding the inserted text will be shifted by the appropriate offset (`iInsertionLocation +iInsertionLocation`). You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iInsertionLength`

A value of type `UniCharCount` (page 198) that represents the length of inserted text. This can be a range outside the text layout object's text range, since you will want to alert the text layout object of insertions that occur before or after this subrange. Text preceding the inserted text will be shifted by the appropriate offset (`iInsertionLocation +iInsertionLocation`). You can pass the constant `kATSUtexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUTextInserted` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUTextInserted` function to indicate to a text layout object that text has been inserted.

When you call `ATSUTextInserted`, drawing caches are updated and text is inserted where specified. `ATSUTextInserted` does not insert style runs or line breaks. To insert these, call the functions `ATSUSetRunStyle` (page 114) and `ATSUSetSoftLineBreak` (page 159), respectively.

When you call `ATSUTextInserted`, the memory location and offset of the text in the text layout object is unchanged, but the total length of the text is extended by the specified amount. The style run containing the insertion point is extended. If the insertion point is the border between two style runs, the first is extended to include the new text. You are responsible for making sure that the corresponding text is inserted into the text buffer.

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUTextDeleted` (page 142)

Measuring Typographic and Image Bounds

- `ATSUGetGlyphBounds` (page 145) obtains the typographic glyph bounds of a range of text.
- `ATSUMeasureText` (page 148) obtains the typographic bounds of a range of .
- `ATSUMeasureTextImage` (page 153) obtains the image bounds.

ATSUGetGlyphBounds

Obtains the typographic glyph bounds of a range of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). The reference points to an initialized text layout object that contains the range of text whose image bounds you want to obtain. You cannot pass `NULL` for this parameter.

`iTextBasePointX`
A value of type `ATSUTextMeasurement` (page 196). This value represents the x-coordinate of the position of the origin of the line in the current graphics port containing the range of text

whose typographic glyph bounds you want to calculate. This enables you to match up the passed back trapezoids to the image so you can draw the bounds. If you just want the length of the bounds but not its onscreen position, pass `0`. If you want to calculate the glyph bounds relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iTextBasePointY`

A value of type `ATSUTextMeasurement` (page 196). This value represents the y-coordinate of the position of the origin of the line in the current graphics port containing the range of text whose typographic glyph bounds you want to calculate. This enables you to match up the passed back trapezoids to the image so you can draw the bounds. If you just want the length of the bounds but not its onscreen position, pass `0`. If you want to calculate the glyph bounds relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iBoundsCharStart`

A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text whose onscreen typographic glyph bounds you want to obtain. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iBoundsCharLength`

A value of type `UniCharCount` (page 198). The value represents the length of the range of text whose onscreen typographic glyph bounds you want to obtain.You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUGetGlyphBounds` returns the result code `kATSUInvalidTextRangeErr`.

`iTypeOfBounds` A value of type `UInt16`. The value specifies whether the width of the resulting typographic glyph bounds will be determined using the caret origin, glyph origin in device space, or glyph origin in fractional absolute positions. See "Glyph Bounds Constants" (page 222) for a description of possible values.

`iMaxNumberOfBounds`
A value of type `ItemCount`. This value represents the number of glyph bounds you want passed back in the `oGlyphBounds` array. You can predetermine this value by first calling `ATSUGetGlyphBounds (iTextLayout, iTextBasePointX, iTextBasePointY, iBoundsCharStart, iBoundsCharLength, iTypeOfBounds, NULL, &oActualNumberOfBounds)`.

`oGlyphBounds` An array of structures of type `ATSTrapezoid` (page 185) that represents the typographic glyph bounds. For information You can predetermine how much memory to allocate for this array by first calling `ATSUGetGlyphBounds (iTextLayout, iTextBasePointX, iTextBasePointY, iBoundsCharStart, iBoundsCharLength, iTypeOfBounds, NULL, &oActualNumberOfBounds)`. You cannot pass `NULL` for this parameter.

`oActualNumberOfBounds`
A pointer to a value of type `ItemCount`. On return, this value represents the actual number of typographic bounding trapezoids of the specified character(s). This may be greater than the value passed in the `iMaxNumberOfBounds` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUGetGlyphBounds` function passes back an array of structures of type `ATSTrapezoid` (page 185) that represent the final laid out line bounding trapezoids. You might want to call `ATSUGetGlyphBounds` to do your own text highlighting using the fractional origin instead of the device origin for the width of the highlight.

Before calculating the typographic glyph bounds of a range of text, `ATSUGetGlyphBounds` 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 no style run at the end of the range of text, `ATSUGetGlyphBounds` assigns the remaining characters to the last style run it can find.

Each `ATSTrapezoid` structure contains the coordinates of a typographic bounding trapezoid. The maximum number of trapezoids is 31, corresponding to 16 bi-directional levels. The coordinates of each trapezoid are offset by the amount you specify in the `iTextBasePointX` and `iTextBasePointY` parameters. If you want to draw the typographic bounds on the screen, pass the position of the origin of the line in current graphics port in these parameters. This enables `ATSUGetGlyphBounds` to match the trapezoids to their onscreen image.

The height of the trapezoid(s) is determined by the line ascent and descent attribute values you previously set for the line. If you have not set these values for the line, `ATSUGetGlyphBounds` will use the values set for the text layout object containing the line. If neither have been set, `ATSUGetGlyphBounds` will use the natural line ascent and descent calculated for the line.

The width of the trapezoid(s) is determined using the caret origin, glyph origin in device space, or glyph origin in fractional absolute positions, depending upon the value you pass in the `iTypeOfBounds` parameter.

VERSION NOTES

Available with ATSUI 1.1.

ATSUMeasureText

Obtains the typographic bounding rectangle of a range of text.

```
OSStatus ATSUMeasureText (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iLineStart,
                    UniCharCount iLineLength,
                    ATSUTextMeasurement *oTextBefore,
                    ATSUTextMeasurement *oTextAfter,
                    ATSUTextMeasurement *oAscent,
                    ATSUTextMeasurement *oDescent);
```

iTextLayout	A reference of type ATSUTextLayout (page 195). The reference points to an initialized text layout object that contains the range of text whose typographic bounds you want to obtain. You cannot pass NULL for this parameter.
iLineStart	A value of type UniCharArrayOffset (page 198). The value represents the edge offset in backing-store of the beginning of line of text you want to measure. You can pass the constant kATSUFromTextBeginning, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text buffer.
iLineLength	A value of type UniCharCount (page 198). The value represents the length of the range of text you want to measure. You can pass the constant kATSUToTextEnd, described in "Text Length Constant" (page 255), to represent the end of the text buffer. Note that if the range of text extends beyond the text buffer, ATSUMeasureText returns the result code kATSUInvalidTextRangeErr.
oTextBefore	A pointer to a value of type ATSUTextMeasurement (page 196). On return, the value represents the starting position of the typographic bounding rectangle relative to the origin of the line in the current graphics port.
oTextAfter	A pointer to a value of type ATSUTextMeasurement (page 196). On return, the value represents the ending location of the typographic bounding rectangle relative to the origin of the line. The value reflects text with no line rotation, justification, or flushness and is independent of the rendering device used to display the text.
oAscent	A pointer to a value of type ATSUTextMeasurement (page 196). On return, the value represents the ascent for the entire line relative to the origin of the line, including cross-stream or baseline shifts. The value reflects text with no line rotation, justification, or flushness and is independent of the rendering device used to display the text.
oDescent	A pointer to a value of type ATSUTextMeasurement (page 196). On return, the value represents the descent for the entire line relative to the origin of the line, including cross-stream or

baseline shifts. The value reflects text with no line rotation, justification, or flushness and is independent of the rendering device used to display the text.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUMeasureText` function obtains the typographic bounds of a set of glyphs. Your application should call the `ATSUMeasureText` function to determine where to place the origin of the lines and leading spaces between lines.

Before calculating the typographic glyph bounds of a range of text, `ATSUMeasureText` 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, `ATSUMeasureText` 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, `ATSUMeasureText` assigns these characters to the first style run it can find. If there no style run at the end of the range of text, `ATSUMeasureText` assigns the remaining characters to the last style run it can find.

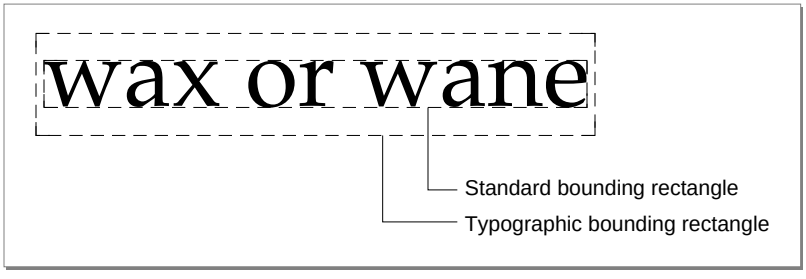
The typographic bounding rectangle does not reflect line rotation, justification, or alignment text layout attributes set for the line or text layout object. The coordinates are independent of the rendering device used to display the text.

The height of the typographic bounding rectangle is determined by the line ascent and line descent text layout attribute values set for the line or text layout object. If these attribute values have been set for the line, `ATSUMeasureText` will use them. Otherwise, it will check to see whether these values have been set in the text layout object. If neither have been set, it will use the natural line ascent and descent calculated for the line.

Figure 2-1 illustrates the difference between the standard and typographic bounding rectangles. The typographic bounding rectangle is the smallest rectangle that encloses the full span of the glyphs from the ascent line to the

descent line, regardless of whether any glyphs extend to those lines. The width of the rectangle extends from the origin of the first glyph through the advance width of the last glyph, including any hanging punctuation and accounting for shifts due to optical alignment.

Figure 2-1 Standard and typographic bounding rectangles



You should call `ATSUMeasureTextImage` (page 153) when you want to calculate the standard bounding rectangle for a block of text. The standard bounding rectangle is the smallest rectangle that completely encloses the filled or framed parts of the line. While the typographic bounding rectangle takes into account the ascent and descent lines for the displayed glyphs, the standard bounding rectangle just encloses the "inked parts" of the displayed glyphs. However, because of the height differences between glyphs—for example, between a small glyph, such as a lowercase "e", and a taller, larger glyph, such an uppercase "M", or even between glyphs of different fonts and point sizes—the standard bounding rectangle may not be sufficient for your application's purposes.

Before measuring the text, `ATSUMeasureText` turns off justification, rotation, and flushness in the text layout object and treats the text as a single line starting at the offset specified in the `iLineStart` parameter.

If text layout attributes have been set for range of text which `ATSUMeasureText` is measuring, it uses these text layout attributes to determine character layout. If no attributes have been set for the line, `ATSUMeasureText` uses the text layout attributes set for the entire text layout object to determine character layout.

ATSUMeasureText will not change or invalidate existing laid-out lines. Its main purpose is to give you feedback on the typographical extrema of a range of characters so you can position lines determine line breaks.

If the line of text is rotated, the sides of the standard bounding rectangle passed back by ATSUMeasureTextImage (page 153) are parallel to the coordinate axes and encompass the rotated line, while the typographic bounding rectangle passed back by ATSUMeasureText reflects an unrotated line of text. You should pass the standard bounding rectangle of a line of text to the function EraseRect to ensure erase all the text.

If the range of text is less than a line, ATSUMeasureText treats the text as a line and ignores the text layout attributes in the text layout object. Note that it doesn't make sense to specify a range of text that is less than a line because of bi-directional text. If the range matches an existing line, ATSUMeasureText still performs post-compensation actions on it. If the range of text goes beyond the line boundary, ATSUMeasureText ignores soft line breaks (that is, it treats the text as a line).

SPECIAL CONSIDERATIONS

Unless you specify otherwise, ATSUMeasureText may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function ATSUCreateMemorySetting (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

ATSUMeasureTextImage (page 153)

ATSUDrawText (page 163)

ATSUMeasureTextImage

Obtains the standard bounding rectangle (that is, image bounds) of a range of text.

```
OSStatus ATSUMeasureTextImage (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iLineOffset,
                    UniCharCount iLineLength,
                    ATSUTextMeasurement iLocationX,
                    ATSUTextMeasurement iLocationY,
                    Rect *oTextImageRect);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). The reference points to an initialized text layout object that contains the range of text whose image bounds you want to obtain. You cannot pass `NULL` for this parameter.

`iLineOffset` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset in backing-store of the beginning of line of text you want to measure. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text buffer.

`iLineLength` A value of type `UniCharCount` (page 198). The value represents the length of the range of text you want to measure. You can pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 255), to represent the end of the text buffer. Note that if the range of text extends beyond the text buffer, `ATSUMeasureText` returns the result code `kATSUInvalidTextRangeErr`.

`iLineLength` A value of type `UniCharCount` (page 198) that represents the length of the text to measure. You can pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 255), to represent the length of an application-specified range of text. Note that if the text range extends beyond the text layout object's text range, `ATSUMeasureTextImage` returns the result code `kATSUInvalidTextRangeErr`.

`iLocationX` A value of type `ATSUTextMeasurement` (page 196). This value represents the x-coordinate of the position of the origin of the line in the current graphics port containing the range of text

whose image bounds you want to calculate. If you want to measure the image bounds relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iLocationY` A value of type `ATSUTextMeasurement` (page 196). This value represents the y-coordinate of the position of the origin of the line in the current graphics port containing the range of text whose image bounds you want to calculate. If you want to measure the image bounds relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`oTextImageRect` A pointer to a structure of type `Rect`. On return, the structure contains the enclosing rectangle of the image bounds of the text offset by the `iLocationX` and `iLocationY` parameters. If the line is rotated, the rectangle's sides are parallel to the coordinate axis. You cannot pass `NULL` for this parameter.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUMeasureTextImage` function to calculate the standard bounding rectangle for a range of text. You should call `ATSUMeasureTextImage` when you want to obtain the standard bounding rectangle of the final line (that is, a line that has been justified, aligned, etc.) to determine the distance of a specific character within the line from the start of line. It reflects the final laid-out line and includes the effects of all text layout attributes that have been set, including hanging punctuation. The rectangle passed back in the `oTextImageRect` parameter is the same rectangle used by `ATSUDrawText` (page 163).

The height of the typographic bounding rectangle is determined by the natural line ascent and descent calculated for the line. `ATSUMeasureTextImage` ignores line ascent and descent text layout attributes set for the line or text layout attribute in which it is measuring typographic bounds.

You should call `ATSUMeasureText` (page 148) when you want to calculate the typographic bounding rectangle for a block of text. This is important for editing and word processing applications, since it enables them to ascertain where to place the origin of the lines and leading spaces between lines. Before measuring the text, `ATSUMeasureText` turns off justification, rotation, and flushness in the text layout object and treats the text as a single line starting at `iLineStart`. For more information on typographic and standard bounding rectangles, see `ATSUMeasureText` (page 148).

If text layout attributes have been set for range of text which `ATSUMeasureTextImage` is measuring, it uses these text layout attributes to determine character layout. If no attributes have been set for the line, `ATSUMeasureTextImage` uses the text layout attributes set for the entire text layout object to determine character layout.

If the line of text is rotated, the sides of the standard bounding rectangle passed back by `ATSUMeasureTextImage` are parallel to the coordinate axes and encompass the rotated line, while the typographic bounding rectangle passed back by `ATSUMeasureText` reflects an unrotated line of text. You should pass the standard bounding rectangle of a line of text to the function `EraseRect` to ensure erase all the text.

The coordinates you pass in the `iLocationX` and `iLocationY` parameters represent the specified location of the line's origin of the text you want to measure. Usually, these are the same values used by `ATSUDrawText` (page 163) for the line of text to be measured.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUMeasureTextImage` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUMeasureText` (page 148)

`ATSUDrawText` (page 163)

Manipulating Line Breaks

ATSUI provides the following functions for manipulating line breaks:

- `ATSUBreakLine` (page 156) calculates the best location for a soft line break in a line and optionally performs the line break.
- `ATSUSetSoftLineBreak` (page 159) sets the position of a specified soft line break in a range of text.
- `ATSUGetSoftLineBreaks` (page 160) obtains the positions of all the set soft line breaks from a range of text.
- `ATSUClearSoftLineBreaks` (page 162) removes all set soft line breaks from a range of text.

ATSUBreakLine

Calculates the best location for a soft line break in a line and optionally performs the line break.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object whose soft line breaks you wish `ATSUBreakLine` to calculate and optionally set. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 198). The first time you call `ATSUBreakLine`, pass the edge offset of the beginning of the line whose soft line breaks you wish to calculate. On subsequent calls, pass the soft line break `ATSUBreakLine`

calculated for the previous line. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the buffer of the text layout object.

`iLineWidth` A value of type `ATSUTextMeasurement` (page 196). Pass the line width that you want `ATSUBreakLine` to use to determine how many characters can fit on the line. You must pass a value must be greater than 0, otherwise `ATSUBreakLine` will pass back the same value in the `oLineBreak` parameter that you passed in the `iLineStart` parameter. You can pass the constant `kATSUUseLineControlWidth`, described in "Line Layout Width Constant" (page 233), if you want `ATSUBreakLine` to use the line width attribute previously set for the text layout object. Note that if the line is outside the text buffer of the text layout object, `ATSUBreakLine` returns the result code `kATSUInvalidTextRangeErr`.

`iUseAsSoftLineBreak`
A value of type `Boolean`. Pass `true` if you want `ATSUBreakLine` to set the line break it calculates. Pass `false` if you want `ATSUBreakLine` to suggest a soft line break but not actually set it.

`oLineBreak` A pointer to a value of type `UniCharArrayOffset` (page 198). On return, `oLineBreak` points to either the suggested or the actual soft line break, depending on the value you passed in the `iUseAsSoftLineBreak` parameter. If this value is the same as that passed in the `iLineStart` parameter, `ATSUBreakLine` did not perform a line break because the `iLineWidth` parameter is not wide enough to fit any characters.

function result A result code. The result code `kATSULineBreakInWord` indicates that `ATSUBreakLine` performed a line break within a word. In this case, `ATSUBreakLine` passes back the location of the soft line break in the `oLineBreak` parameter. Note that this is a status message, not an error code. The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encounters an error rendering or measuring a line of ATSUI text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUBreakLine` function calculates the best location for a soft line break within a specified line width and optionally performs the line break. You should call `ATSUBreakLine` when the user inserts or deletes text or changes text layout attributes that affect how glyphs are laid out. If the user changes attributes that don't affect how glyphs are laid out, it passes back the previously set soft line breaks.

You specify the desired behavior of this function by passing the appropriate value in the `iUseAsSoftLineBreak` parameter. If you want `ATSUBreakLine` to calculate but not set soft line breaks, you should pass `false` in the `iUseAsSoftLineBreak` parameter. You should do this if you want to implement word break hyphenation. To set the soft line break, call the function `ATSUSetSoftLineBreak` (page 159). If you want `ATSUBreakLine` to perform line breaking, pass `true` in `iUseAsSoftLineBreak`.

Before calculating soft line break positions, `ATSUBreakLine` turns off any line justification, line rotation, and line alignment attribute values that you previously set in the text layout object and treats the text as a single line. `ATSUBreakLine` then 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, `ATSUBreakLine` 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, `ATSUBreakLine` assigns these characters to the first style run it can find. If there no style run at the end of the range of text, `ATSUBreakLine` assigns the remaining characters to the last style run it can find.

If text layout attribute values have been set for the line for which `ATSUBreakLine` is calculating a soft line break position, `ATSUBreakLine` uses these text layout attributes to determine where to break the line. If no text layout attribute values have been set for the line, `ATSUBreakLine` uses the values set for the text layout object containing the line.

You should repeatedly call `ATSUBreakLine` until it does not find any more soft line breaks.

`ATSUBreakLine` suggests a soft line break each time it encounters a hard line break character. Examples include carriage returns, line feeds, form feeds, and line separators. However `ATSUBreakLine` does not suggest a soft line break when it encounters a paragraph separator.

If `ATSUBreakLine` does not encounter a hard line break, it uses the line width you specify to determine how many characters can fit on the line. It uses the

calculated soft line break to perform line layout on the characters. `ATSUBreakLine` then determines whether the characters still fit within the line. This is necessary due to end-of-line effects like swashes. When `ATSUBreakLine` sets a soft line break, it clear previously set soft line breaks in the line.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUBreakLine` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

ATSUSetSoftLineBreak

Sets the position of a specified soft line break in a range of text.

```
OSStatus ATSUSetSoftLineBreak (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iLineBreak);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the range of text whose soft line breaks you wish to set. You cannot pass `NULL` for this parameter.

`iLineBreak` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the soft line break you want to set. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer. Note that if the range of text extends beyond the text buffer, `ATSUSetSoftLineBreak` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

The `ATSUSetSoftLineBreak` function enables you to use your own line-breaking algorithm to set soft line break positions in a range of text. You can call the `ATSUBreakLine` (page 156) function if you wish to use its default line-breaking

algorithm to calculate and set line breaks. You should call `ATSUSetSoftLineBreak` to implement word break hyphenation.

Before calculating soft line break positions, `ATSUSetSoftLineBreak` turns off the previously-set line justification, rotation, and alignment text layout attribute values in the line or text layout object. It treats the text as a single line beginning at the offset specified in the `iLineStart` parameter. You must then call the `ATSUMeasureText` (page 148) function to measure the text.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUSetSoftLineBreak` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUGetSoftLineBreaks` (page 160)

ATSUGetSoftLineBreaks

Obtains the positions of all the set soft line breaks from a range of text.

```
OSStatus ATSUGetSoftLineBreaks (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iRangeStart,
                    UniCharCount iRangeLength,
                    ItemCount iMaximumBreaks,
                    UniCharArrayOffset oBreaks[],
                    ItemCount *oBreakCount);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the range of text whose soft line breaks you wish to obtain. You cannot pass `NULL` for this parameter.

`iRangeStart`	A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text whose soft line break positions you want to obtain. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text buffer.
`iRangeLength`	A value of type `UniCharCount` (page 198). The value represents the length of the range of text whose soft line break locations you want to determine. You can pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 255), to represent the end of the text buffer. Note that if the range of text extends beyond the text buffer, `ATSUGetSoftLineBreaks` returns the result code `kATSUInvalidTextRangeErr`.
`iMaximumBreaks`	A value of type `ItemCount` that represents the number of soft line breaks you want passed back in the `oBreaks` array.
`oBreaks`	An array of values of type `UniCharArrayOffset` (page 198). On return, the array contains the positions of all the soft line breaks set in the specified range of text. You can predetermine how much memory to allocate for this array by first calling `ATSUGetSoftLineBreaks (iTextLayout, 0, 0, 0, NULL, &oBreakCount)`.
`oBreakCount`	A pointer to a value of type `ItemCount`. On return, `oBreakCount` points to the actual number of soft line breaks set in the specified range of text. This may be greater than the value passed in the `iMaximumBreaks` parameter. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUSetSoftLineBreak` (page 159)

ATSUClearSoftLineBreaks

Removes all set soft line breaks from a range of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the range of text whose soft line breaks you wish to remove. You cannot pass `NULL` for this parameter.

`iRangeStart` A value of type `UniCharArrayOffset` (page 198). The value represents the edge offset of the beginning of the range of text whose soft line breaks you want to remove. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text buffer.

`iRangeLength` A value of type `UniCharCount` (page 198). The value represents the length of the range of text whose soft line break locations you want to remove. You can pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 255), to represent the end of the text buffer. Note that if the range of text extends beyond the text buffer, `ATSUGetSoftLineBreaks` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.0.

Drawing Text

ATSUI provides the following function for drawing text:

- `ATSUDrawText` (page 163) draws a range of text at a specified location.

ATSUDrawText

Draws a range of text at a specified location.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the range of text you want to draw. You cannot pass `NULL` for this parameter.

`iLineOffset` A value of type `UniCharArrayOffset` (page 198). This value represents the edge offset of the beginning of the line you want to draw. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iLineLength` A value of type `UniCharCount` (page 198). This value represents the length of the line you want to draw. You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUDrawText` returns the result code `kATSUInvalidTextRangeErr`.

`iLocationX` A value of type `ATSUTextMeasurement` (page 196). This value represents the x-coordinate of the position of the origin of the line in the current graphics port containing the range of text that you want to draw. If you want to draw relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iLocationY` A value of type `ATSUTextMeasurement` (page 196). This value represents the y-coordinate of the position of the origin of the line in the current graphics port containing the range of text that you want to draw. If you want to draw relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while rendering a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUDrawText` function to draw a range of text at a specified location. Before drawing the text, `ATSUDrawText` turns off justification, rotation, and flushness in the text layout object and treats the text as a single line starting at `iLineOffset`.

If text layout attributes have been set for the specific line in which `ATSUDrawText` is drawing, it uses these text layout attributes to determine character layout. If no attributes have been set for the line, `ATSUDrawText` uses the text layout attributes set for the entire text layout object to determine character layout.

To draw a range of text that spans multiple lines, you should call `ATSUDrawText` for each line of text that is being drawn, even if all the lines are in the same text layout object and adjust the value in the `iLineOffset` parameter for each new line being drawn.

The transfer mode and resolution used by `ATSUDrawText` are the same values as those set in the graphics port. Text color is taken from the style object and whatever the value is in the graphics port is ignored. If you did not set text color in the style object, you will get the style object's default value for text color (that is, black), regardless of what was set in the graphics port.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUDrawText` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUMeasureTextImage` (page 153)

`ATSUMeasureText` (page 148)

Highlighting and Unhighlighting Text

ATSUI provides the following functions for highlighting and unhighlighting in a text layout object:

- `ATSUHighlightText` (page 165) highlights a range of text using the highlight information in the graphics port.
- `ATSUUnhighlightText` (page 168) removes highlighting from a specified range of text using the highlight information in the graphics port.
- `ATSUGetTextHighlight` (page 170) determines the highlight region for a range of text but doesn't actually highlight the text.

ATSUHighlightText

Highlights a specified range of text using the highlight information in the graphics port.

```
OSStatus ATSUHighlightText (
                    ATSUTextLayout iTextLayout,
                    ATSUTextMeasurement iTextBasePointX,
                    ATSUTextMeasurement iTextBasePointY,
                    UniCharArrayOffset iHighlightStart,
                    UniCharCount iHighlightLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the text range to be highlighted. You cannot pass `NULL` for this parameter.

`iTextBasePointX`
A value of type `ATSUTextMeasurement` (page 196). This value represents the x-coordinate of the position of the origin of the line in the current graphics port containing the range of text that you want to highlight. If you want to highlight relative to the

current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iTextBasePointY`

A value of type `ATSUTextMeasurement` (page 196). This value represents the y-coordinate of the position of the origin of the line in the current graphics port containing the range of text that you want to highlight. If you want to highlight relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iHighlightStart`

A value of type `UniCharArrayOffset` (page 198) that represents the beginning of the text range to highlight. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iHighlightLength`

A value of type `UniCharCount` (page 198) that specifies the length of the text range to highlight. You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUHighlightText` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

To highlight a range of text that spans multiple lines, you must call `ATSUHighlightText` for each line being highlighted (even if all the lines are in the

same text layout object) and adjust the value in the `iHighlightStart` parameter for each new line being highlighted.

To highlight a range of text that spans multiple lines, you must call `ATSUHighlightText` for each line being highlighted (even if all the lines are in the same text layout object) and adjust the value in the `iHighlightStart` parameter for each new line being highlighted. Note that if the offset is outside the text layout object's text range, `ATSUHighlightText` returns the result code `kATSUInvalidTextRangeErr`.

The coordinates you pass in the `iLocationX` and `iLocationY` parameters represent the specified location of the line's origin of the text you want to highlight. Usually, these are the same values used by `ATSUDrawText` (page 163) for the line of text to be highlighted.

The ascent and descent of the resulting highlighted region are the same values passed back in the `oAscent` and `oDescent` parameters of the function `ATSUMeasureText` (page 148) unless the ascent or descent attributes were set for the line or the entire text layout object by calling `ATSUSetLineControls` (page 100) or `ATSUSetLayoutControls` (page 92), respectively.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUHighlightText` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUUnhighlightText` (page 168)

`ATSUGetTextHighlight` (page 170)

ATSUUnhighlightText

Removes highlighting from a specified range of text using the highlight information in the graphics port.

```
OSStatus ATSUUnhighlightText (
                    ATSUTextLayout iTextLayout,
                    ATSUTextMeasurement iTextBasePointX,
                    ATSUTextMeasurement iTextBasePointY,
                    UniCharArrayOffset iHighlightStart,
                    UniCharCount iHighlightLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the text range whose highlighting is to be removed. You cannot pass `NULL` for this parameter.

`iTextBasePointX`

A value of type `ATSUTextMeasurement` (page 196). This value represents the x-coordinate of the position of the origin of the line in the current graphics port containing the range of text that you want to remove highlighting from. If you want to highlight relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iTextBasePointY`

A value of type `ATSUTextMeasurement` (page 196). This value represents the y-coordinate of the position of the origin of the line in the current graphics port containing the range of text that you want to remove highlighting from. If you want to highlight relative to the current pen location in the current graphics port, pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 206).

`iHighlightStart`

A value of type `UniCharArrayOffset` (page 198) that represents the beginning of the text range to remove highlighting from. You can pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

`iHighlightLength`
: A value of type `UniCharCount` (page 198) that represents the length of the text range to remove highlighting from. You can pass the constant `kATSUTexttoEnd`, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, `ATSUBreakLine` returns the result code `kATSUInvalidTextRangeErr`.

function result
: A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

Your application may use the `ATSUUnhighlightText` function to remove highlighting from a specified range of text using the highlight information in the graphics port.

Note that if the text range does not overlap highlighting performed by a previous highlight call, the results are undefined. It can, however, be longer than the highlight region passed back in a previous call to `ATSUHighlightText` (page 165).

If the text range specified in the `iHighlightStart` and `iHighlightEnd` parameters does not overlap highlighting performed by a previous highlight call, the results are undefined. However, the range can be longer than a previously-highlighted region.

The coordinates you pass in the `iLocationX` and `iLocationY` parameters represent the specified location of the line's origin of the text you want to remove highlighting. Usually, these are the same values used by `ATSUDrawText` (page 163) for the line of text whose highlighting is to be removed.

The ascent and descent of the resulting highlighted region are the same values passed back in the `oAscent` and `oDescent` parameters of the function `ATSUMeasureText` (page 148) unless the ascent or descent attributes were set for

the line or the entire text layout object by calling `ATSUSetLineControls` (page 100) or `ATSUSetLayoutControls` (page 92), respectively.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUUnhighlightText` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUHighlightText` (page 165)

ATSUGetTextHighlight

Determines the highlight region for a range of text.

```
OSStatus ATSUGetTextHighlight (
                    ATSUTextLayout iTextLayout,
                    ATSUTextMeasurement iTextBasePointX,
                    ATSUTextMeasurement iTextBasePointY,
                    UniCharArrayOffset iHighlightStart,
                    UniCharCount iHighlightLength,
                    RgnHandle oHighlightRegion);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object that contains the text range whose highlight region you want to determine. You cannot pass `NULL` for this parameter.

`iTextBasePointX`
A value of type `ATSUTextMeasurement` (page 196). This value represents the x-coordinate of the position of the origin of the line in the current graphics port containing the range of text whose highlight region you want to determine. If you want to determine the highlight region relative to the current pen

location in the current graphics port, pass the constant kATSUUseGrafPortPenLoc, described in "Current Pen Location Constant" (page 206).

iTextBasePointY

A value of type ATSUTextMeasurement (page 196). This value represents the y-coordinate of the position of the origin of the line in the current graphics port containing the range of text whose highlight region you want to determine. If you want to determine the highlight region relative to the current pen location in the current graphics port, pass the constant kATSUUseGrafPortPenLoc, described in "Current Pen Location Constant" (page 206).

iHighlightStart

A value of type UniCharArrayOffset (page 198) that represents the beginning of the text range whose highlight region you want to determine. You can pass the constant kATSUFromTextBeginning, described in "Text Offset Constant" (page 256), to represent the edge offset of the beginning of the text layout object's text buffer.

iHighlightLength

A value of type UniCharCount (page 198) that represents the length of the text range whose highlight region you want to determine. You can pass the constant kATSUtexttoEnd, described in "Text Length Constant" (page 255), to represent the end of the text layout object's text buffer. Note that if the specified text range extends beyond the text layout object's text buffer, ATSUHighlightText returns the result code kATSUInvalidTextRangeErr.

oHighlightRegion

A handle of type RgnHandle. Before calling ATSUGetTextHighlight, create a region handle by calling the function NewRgn. On return, RgnHandle points to a RgnPtr which points to a Region structure. The Region structure has two fields, rgnSize and rgnBBox, that represent the highlight region for the text. In the case of discontiguous highlighting, the region consists of multiple components, with the rgnBBox field specifying the bounding box around the entire area of discontiguous highlighting. You cannot pass NULL for this parameter.

function result A result code. The result code `kATSUCoordinateOverflowErr` indicate that the coordinates passed in the `iTextBasePointX` and `iTextBasePointY` parameters caused a coordinate overflow. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in 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). The result code `kATSUQuickDrawTextErr` indicates that the QuickDraw function `DrawText` encountered an error while measuring a line of text. For a list of other ATSUI-specific result codes, see "Result Codes" (page 256).

DISCUSSION

The `ATSUHighlightText` function simply determines the highlight region for a range of text. It does not highlight the text.

When there are discontiguous highlighting regions, the structure passed back in the `oHighlightRegion` parameter is made up of multiple components. The maximum number of components is 31, based on The `rgnBBox` field of the structure represents the bounding box around the entire area of discontiguous highlighting.

The coordinates you pass in the `iLocationX` and `iLocationY` parameters represent the specified location of the line's origin of the text you want to highlight. Typically, these are the same values used by `ATSUDrawText` (page 163) to draw a line of text.

The ascent and descent of the highlighted region is the same as the values passed back in the `oAscent` and `oDescent` parameters of the function `ATSUMeasureText` (page 148) unless the ascent or descent attributes were set for the line or the entire text layout object by calling `ATSUSetLineControls` (page 100) or `ATSUSetLayoutControls` (page 92), respectively.

SPECIAL CONSIDERATIONS

Unless you specify otherwise, `ATSUGetTextHighlight` may allocate memory in your application heap. If you want more control over ATSUI's memory allocation, see the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available with ATSUI 1.0.

SEE ALSO

`ATSUHighlightText` (page 165)

Performing Background Processing

ATSUI provides the following function for performing background processing:

- `ATSUIdle` (page 173) enable ATSUI to perform background processing for a text layout object.

ATSUIdle

Enable ATSUI to perform background processing for a text layout object.

```
OSStatus ATSUIdle (ATSUTextLayout iTextLayout);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 195). Pass a reference to an initialized text layout object in which you want ATSUI to perform background processing. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Although versions 1.0 and 1.1 of ATSUI do not implement background processing for text layout objects, you should call `ATSUIdle` at least once in your main event loop to support the implementation of background processing in future versions of ATSUI.

VERSION NOTES

Available with ATSUI 1.0.

Functions for Manipulating Memory Settings

This section describes the functions you can use to manipulate memory settings in ATSUI:

- `ATSUCreateMemorySetting` (page 174) creates a memory allocation setting that specifies either the specific heap or the application-defined callback functions that ATSUI should use when allocating memory.
- `ATSUSetCurrentMemorySetting` (page 176) establishes a current memory allocation setting.
- `ATSUGetCurrentMemorySetting` (page 176) returns the current memory allocation setting.
- `ATSUDisposeMemorySetting` (page 177) disposes of a specific memory allocation setting.

ATSUCreateMemorySetting

Creates a memory allocation setting that specifies either the specific heap or the application-defined callback functions that ATSUI should use when allocating memory.

```
OSStatus ATSUCreateMemorySetting (
                    ATSUHeapSpec iHeapSpec,
                    ATSUMemoryCallbacks *iMemoryCallbacks,
                    ATSUMemorySetting *oMemorySetting);
```

`iHeapSpec` A value of type `ATSUHeapSpec`. This value represents either the heap or the application-defined functions that you want ATSUI to use when allocating memory. See "Heap Specification Constants" (page 224) for a description of possible values. You must pass a valid value for this parameter.

`iMemoryCallbacks`
A pointer to a union of type `ATSUMemoryCallbacks` (page 193) that contains either the heap or the application-defined functions that ATSUI should use when allocating memory. If you pass the

kATSUUseSpecificHeap constant in the iHeapSpec parameter, you must pass a pointer to a union that contains the correctly-prepared heap in the heapToUse field. If you pass the kATSUUseCallbacks constant in the iHeapSpec parameter, you must pass a pointer to a ATSUMemoryCallbacks union that contains pointers to your application-defined functions. If you pass the kATSUUseCurrentHeap or kATSUUseAppHeap constant in the iHeapSpec parameter, you should pass a NULL pointer.

oMemorySetting

A pointer to a reference of type ATSUMemorySetting (page 194). On return, the reference points to the new memory allocation setting. If you want this setting to be used in subsequenet Memory Manager calls, pass it to the ATSUSetCurrentMemorySetting (page 176) function. You cannot pass NULL for this parameter.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

You must pass the memory setting created by the ATSUCreateMemorySetting function to the ATSUSetCurrentMemorySetting (page 176) function to ensure that it will be used in subsequent Memory Manager calls.

You might want to create different memory settings for different memory allocation operations. For example, you might create two different settings designating different heaps to use for allocating the memory associated wtih style and text layout object creation. Before creating a style or text layout object, you would then make the appropriate setting current by calling ATSUSetCurrentMemorySetting.

If you want the most control possible over memory allocation in ATSUI, you should write your own memory allocation callback functions. In this case, you should pass the kATSUUseCallbacks constant in the iHeapSpec parameter and a pointer to a ATSUMemoryCallbacks union that contains pointers to your callback functions in the iMemoryCallbacks parameter.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUSetCurrentMemorySetting` (page 176)

ATSUSetCurrentMemorySetting

Establishes a current memory allocation setting.

```
OSStatus ATSUSetCurrentMemorySetting (ATSUMemorySetting iMemorySetting);
```

`iMemorySetting`
A reference of type `ATSUMemorySetting` (page 194). Pass a reference to a memory setting created by calling `ATSUCreateMemorySetting` (page 174) that you want to make current. Until another setting is made current, this setting will be used in subsequent Memory Manager calls made within ATSUI.

function result A result code. See "Result Codes" (page 256).

VERSION NOTES

Available with ATSUI 1.1.

ATSUGetCurrentMemorySetting

Returns the current memory allocation setting.

```
ATSUMemorySetting ATSUGetCurrentMemorySetting (
                    void);
```

function result A reference of type `ATSUMemorySetting` (page 194) to the current memory setting. If there is no current memory setting, `ATSUGetCurrentMemorySetting` returns `NULL`.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`ATSUSetCurrentMemorySetting` (page 176)

ATSUDisposeMemorySetting

Disposes of a specific memory allocation setting.

```
OSStatus ATSUDisposeMemorySetting (
                    ATSUMemorySetting iMemorySetting);
```

`iMemorySetting`

A reference of type `ATSUMemorySetting` (page 194). Pass a reference to the memory setting you want to dispose. If you want to dispose of the current memory setting, ATSUI's memory setting will be set to the default setting. The default setting uses the current heap and internal callbacks to perform memory allocation operations.

function result A result code. See "Result Codes" (page 256).

DISCUSSION

Before disposing of a memory setting using the `ATSUDisposeMemorySetting` function, you should dispose of the memory associated with style and text layout objects that were allocated using that memory setting. For example, if you want to dispose of a memory setting that uses your application-defined callback functions to allocate memory, you should dispose of any memory that ATSUI allocated as a result of these callbacks before disposing of the setting. Note that is is an advanced technique and should be used with caution.

VERSION NOTES

Available with ATSUI 1.1.

Application-Defined Functions for Controlling Memory Allocation

This section describes the functions you can provide if you wish to exercise complete control over memory allocation operations in ATSUI:

- `MyATSUCustomAllocFunc` (page 178) manages memory allocation operations typically handled by ATSUI.
- `MyATSUCustomGrowFunc` (page 179) manages memory reallocation typically handled by ATSUI.
- `MyATSUCustomFreeFunc` (page 181) manages memory deallocation operations typically handled by ATSUI.

MyATSUCustomAllocFunc

Manages memory allocation operations typically handled by ATSUI.

This is how you would declare your own memory allocation callback function if you were to name the function `MyATSUCustomAllocFunc`:

```
void * MyATSUCustomAllocFunc (
                    void *refCon,
                    ByteCount howMuch);
```

`refCon` A pointer to arbitrary data for use in your application-defined callback function. ATSUI passes a pointer to data previously supplied by your application in the `memoryRefCon` field of the `ATSUMemoryCallbacks` (page 193) union for your own use.

`howMuch` The number of bytes of memory that ATSUI wants you to allocate.

function result An untyped pointer to the beginning of the block of memory allocated by your `MyATSUCustomAllocFunc` function.

DISCUSSION

If you want total control over memory allocation in ATSUI, including specifying the heap to use and how memory allocation should occur, you should write an application-defined callback function like `MyATSUCustomAllocFunc`.

You can register your application-defined memory allocation function by calling the `ATSUCreateMemorySetting` (page 174) function and passing the `kATSUUseCallbacks` constant in the `iHeapSpec` parameter and a pointer to a `ATSUMemoryCallbacks` (page 193) union in the `iMemoryCallbacks` parameter. You should supply a pointer of type `ATSUCustomAllocFunc` (page 189) to your `MyATSUCustomAllocFunc` function in the `Alloc` field in the `callbacks` structure of the memory callback union.

Note that your `MyATSUCustomAllocFunc` function is expected to return a pointer to the start of the allocated memory, unless it terminates in an application.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`MyATSUCustomAllocFunc` (page 178)

`MyATSUCustomGrowFunc` (page 179)

MyATSUCustomGrowFunc

Manages memory reallocation operations typically handled by ATSUI.

This is how you would declare your own memory reallocation callback function if you were to name the function `MyATSUCustomGrowFunc`:

```
void * MyATSUCustomGrowFunc (
                    void *refCon,
                    void *oldBlock,
                    ByteCount oldSize,
                    ByteCount newSize);
```

`refCon`	A pointer to arbitrary data previously supplied by your application in the `memoryRefCon` field of the `ATSUMemoryCallbacks` (page 193) union.
`oldBlock`	An untyped pointer to the beginning of the block to be reallocated. ATSUI passes your application this value.
`oldSize`	The size (in bytes) of the old block (that is, before the reallocation operation). ATSUI passes this value so your callback function knows how many bytes to copy if you need to allocate memory for the grown block.
`newSize`	The size (in bytes) of the new block (that is, after the reallocation operation).
function result	An untyped pointer to the beginning address of the block of memory reallocated by your `MyATSUCustomGrowFunc` function.

DISCUSSION

If you want total control over memory reallocation in ATSUI, including specifying the heap to use and how memory reallocation should occur, you should write an application-defined callback function like `MyATSUCustomGrowFunc`.

You can register your application-defined memory reallocation function by calling the `ATSUCreateMemorySetting` (page 174) function and passing the `kATSUUseCallbacks` constant in the `iHeapSpec` parameter and a pointer to a `ATSUMemoryCallbacks` (page 193) union in the `iMemoryCallbacks` parameter. The `ATSUMemoryCallbacks` should contain a should supply a pointer of type `ATSUCustomGrowFunc` (page 190) to your `MyATSUCustomGrowFunc` function in the `Grow` field in the `callbacks` structure of the memory callback union.

Note that your `MyATSUCustomGrowFunc` function is expected to return a pointer to the start of the reallocated memory, unless it terminates in an application.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`MyATSUCustomAllocFunc` (page 178)

MyATSUCustomFreeFunc (page 181)

MyATSUCustomFreeFunc

Manages memory deallocation operations typically handled by ATSUI.

This is how you would declare your own memory deallocation callback function if you were to name the function MyATSUCustomFreeFunc:

```
void * MyATSUCustomFreeFunc (
                    void *refCon,
                    void *doomedBlock);
```

refCon	A pointer to arbitrary data for use in your application-defined callback function. ATSUI passes a pointer to data previously supplied by your application in the memoryRefCon field of the ATSUMemoryCallbacks (page 193) union for your own use.
doomedBlock	An untyped pointer to the beginning of the block to dispose.
function result	An untyped pointer to the beginning address of the block of memory freed by your MyATSUCustomFreeFunc function.

DISCUSSION

If you want total control over memory deallocation in ATSUI, including specifying the heap to use and how memory deallocation should occur, you should write an application-defined callback function like MyATSUCustomFreeFunc.

You can register your application-defined memory deallocation function by calling the ATSUCreateMemorySetting (page 174) function and passing the kATSUUseCallbacks constant in the iHeapSpec parameter and a pointer to a ATSUMemoryCallbacks (page 193) union in the iMemoryCallbacks parameter. You should supply a pointer of type ATSUCustomFreeFunc (page 190) to your MyATSUCustomFreeFunc function in the Free field in the callbacks structure of the memory callback union.

Note that your MyATSUCustomFreeFunc function is expected to return a pointer to the start of the deallocated memory, unless it terminates in an application.

VERSION NOTES

Available with ATSUI 1.1.

SEE ALSO

`MyATSUCustomAllocFunc` (page 178)

`MyATSUCustomGrowFunc` (page 179)

Data Types

This section describes the data types that are defined by ATSUI for your application's use:

- `ATSJustPriorityWidthDeltaOverrides` (page 183)
- `ATSTrapezoid` (page 185)
- `ATSUAttributeInfo` (page 186)
- `ATSUAttributeValuePtr` (page 187)
- `ATSUCaret` (page 188)
- `ATSUCustomAllocFunc` (page 189)
- `ATSUCustomGrowFunc` (page 190)
- `ATSUCustomFreeFunc` (page 190)
- `ATSUFontFeatureType` (page 191)
- `ATSUFontFeatureSelector` (page 191)
- `ATSUFontID` (page 192)
- `ATSUFontVariationAxis` (page 192)
- `ATSUFontVariationValue` (page 193)
- `ATSUMemoryCallbacks` (page 193)
- `ATSUMemorySetting` (page 194)
- `ATSUStyle` (page 195)
- `ATSUTextLayout` (page 195)

- ATSUTextMeasurement (page 196)
- BslnBaselineRecord (page 196)
- ConstUniCharArrayPtr (page 197)
- UniChar (page 197)
- UniCharArrayHandle (page 197)
- UniCharArrayOffset (page 198)
- UniCharArrayPtr (page 198)
- UniCharCount (page 198)

ATSJustPriorityWidthDeltaOverrides

Your application can use an array of type ATSJustPriorityWidthDeltaOverrides to override the distribution behavior of a glyph or set of glyphs during justification in a style run. A priority justification override array contains four width delta structures of type ATSJustPriorityWidthDeltaOverrides, one for each justification priority.

To set the priority justification override in a style run, pass a value of type ATSJustPriorityWidthDeltaOverrides and the kATSUPriorityJustOverrideTag tag constant to the ATSUSetAttributes (page 29) function.

A width delta structure contains all the information needed to override the distribution behavior of a glyph or set of glyphs during justification. It is used in both the priority justification override structure and the glyph justification override structure. In each case the width delta structure can specify both a change in priority and a change in distribution behavior.

Each width delta structure in the priority justification override structure specifies overrides for all glyphs of a given justification priority. Thus, for each priority, you can

- Change the priority: assign all glyphs of one priority to another priority.
- Change the behavior: leave the priority the same, but change the priority behavior of all glyphs of that priority.
- Change both: assign all glyphs of one priority to another priority, and change their justification behavior from the defaults of either priority.

Note that if you change one priority to another and change the default behavior of all glyphs of that priority, the behavior of other glyphs already having that priority is not changed. For example, if you change all glyphs with a priority of `kJUSTLetterPriority` to `kJUSTSpacePriority` and give them special behavior, glyphs that already have a priority of `kJUSTSpacePriority` retain their default behavior. Only `kJUSTLetterPriority` glyphs are overridden, and so the overriding behavior applies to only those glyphs.

Unlimited gap absorption is a special case in that it applies across an entire line instead of just to a single style run. If both the `kJUSTOverrideUnlimited` bit and the `kJUSTUnlimited` flag, described in "Justification Override Mask Constants" (page 226), are set in any width delta structure for the glyph justification override structure of any style run on a line, ATSUI distributes the current justification gap among all instances of that glyph in all style runs on the line.

```
struct ATSJustWidthDeltaEntryOverride {
    Fixed               beforeGrowLimit;
    Fixed               beforeShrinkLimit;
    Fixed               afterGrowLimit;
    Fixed               afterShrinkLimit;
    JustificationFlags  growFlags;
    JustificationFlags  shrinkFlags;
};
typedef struct ATSJustWidthDeltaEntryOverride
ATSJustWidthDeltaEntryOverride;
typedef ATSJustWidthDeltaEntryOverride
ATSJustPriorityWidthDeltaOverrides[4];
```

Field descriptions

`beforeGrowLimit` The number of points by which a 1-point 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 of extra space added on the left side (top side for vertical text).

`beforeShrinkLimit` The number of points by which a 1-point glyph can shrink on the left side (top side for vertical text). If specified, this value should be negative.

`afterGrowLimit` The number of points by which a 1-point glyph can expand on the right side (bottom side for vertical text).

`afterShrinkLimit`	The number of points by which a 1-point glyph can shrink on the right side (bottom side for vertical text). If specified, this value should be negative.
`growFlags`	Mask values you can use to check whether bits are set to override which aspects of the normal, font-specified justification behavior for a particular set of glyphs to override. See "Justification Override Mask Constants" (page 226) for a description of possible values. You can use these flags to selectively override the grow case only, while retaining default behavior for other cases.
`shrinkFlags`	Controls which aspects of the normal, font-specified justification behavior for a particular set of glyphs to override. See "Justification Override Mask Constants" (page 226) for a description of possible values. You can use these flags to selectively override the shrink case only, while retaining default behavior for other cases.

VERSION NOTES

Available with ATSUI 1.0.

ATSTrapezoid

The function `ATSUGetGlyphBounds` (page 145) passes back an array of structures of type `ATSTrapezoid` in the `oGlyphBounds` parameter to represent the typographic glyph bounds for a range of text. The maximum number of typographic bounding trapezoids that can be passed back is 31, corresponding to 16 bi-directional levels.

Each `ATSTrapezoid` structure contains the coordinates of the corners of a typographic bounding trapezoid offset by the amount specified in the `iTextBasePointX` and `iTextBasePointY` parameters. If you want the corners to match their image, pass the coordinates of the position of the origin of the line in the current graphics port. If you do not care about the position of the typographic glyph bounds in the current graphics port, pass `(0,0)` for these parameters.

Depending on the value passed in the `iTypeOfBounds` parameter, the width of the glyph bounds will be determined using the caret origin, glyph origin in device space, or glyph origin in fractional absolute positions.

The height of the glyph bounds are determined by the line ascent and line descent text layout attribute values set for the line or text layout object. If these attribute values have been set for the line, `ATSUGetGlyphBounds` will use them. Otherwise, it will check to see whether these values have been set in the text layout object. If neither have been set, it will use the natural line ascent and descent calculated for the line.

```
struct ATSTrapezoid {
    FixedPoint      upperLeft;
    FixedPoint      upperRight;
    FixedPoint      lowerRight;
    FixedPoint      lowerLeft;
};
```

Field descriptions

`upperLeft`	A `FixedPoint` structure that contains the upper left coordinates (assuming a horizontal line of text) of the typographic glyph bounds.
`upperRight`	A `FixedPoint` structure that contains the upper right coordinates (assuming a horizontal line of text) of the typographic glyph bounds.
`lowerRight`	A `FixedPoint` structure that identifies the lower right coordinates (for a horizontal line of text) of the typographic glyph bounds.
`lowerLeft`	A `FixedPoint` structure that identifies the lower left coordinates(for a horizontal line of text) of the typographic glyph bounds.

VERSION NOTES

Available with ATSUI 1.1.

ATSUAttributeInfo

The `ATSUGetAllAttributes` (page 33), `ATSUGetAllLayoutControls` (page 95), and `ATSUGetAllLineControls` (page 104) functions pass back an array of structures of type `ATSUAttributeInfo`. Each structure contains a tag that identifies a particular style run or text layout attribute value, depending on the function called, and the size (in bytes) of the attribute value. You can use this information to query

the ATSUGetAttribute (page 31), ATSUGetLayoutControl (page 94), and ATSUGetLineControl (page 102) functions for the corresponding attribute value in a style object, text layout object, or line in a text layout object, respectively.

```
typedef struct {
    ATSUAttributeTag    fTag;
    ByteCount           fValueSize
}ATSUAttributeInfo;
```

Field descriptions

fTag	A value of type ATSUAttributeTag that identifies a particular style run or text attribute value. See "Style Run Attribute Tags" (page 237) and "Text Layout and Line Attribute Tags" (page 250), respectively, for a description of possible tag values.
fValueSize	The size (in bytes) of the attribute value corresponding to the attribute tag in the fTag parameter.

VERSION NOTES

Available with ATSUI 1.0.

ATSUAttributeValuePtr

Your application passes an array of pointers of type ATSUAttributeValuePtr to the ATSUSetAttributes (page 29), ATSUSetLayoutControls (page 92), and ATSUSetLineControls (page 100) functions. Each pointer references a previously set style run or text layout object attribute value, depending upon the function you called. set the pointer refers to a style run or text layout attribute access to style run and text layout attribute values, which vary in size.

The functions ATSUGetAttribute (page 31), ATSUGetLayoutControl (page 94), and ATSUGetLineControl (page 102) pass back a pointer of type ATSUAttributeValuePtr that references an attribute value corresponding to a particular tag and value-size. You should cast this to the appropriate data type and dereference it to obtain the actual attribute value.

```
typedef void *   ATSUAttributeValuePtr;
```

VERSION NOTES

Available with ATSUI 1.0.

ATSUCaret

The `ATSUOffsetToPosition` (page 131) function passes back two structures of type `ATSUCaret` that contain the caret position(s) corresponding to a specified edge offset. If the caret is not split, both structures contain the starting and ending pen locations of the main caret. If the caret is split, the structure passed back in the `oMainCaret` parameter contains the starting and ending pen locations of the high caret, while that in the `oSecondCaret` parameter contains the starting and ending pen locations of the low caret.

Note that the passed back caret locations are relative to the position of the origin of the line of interest in the graphics port. You can use these positions to draw angled carets, split carets, and carets on angled lines.

In order to draw caret(s) on the screen, you must transform these locations by adding them to the position of the origin of the line in the graphics port in which the hit occurred. For example, if `ATSUOffsettoPosition` passed back the starting and ending pen locations of (25,0), (25,25) in the `oMainCaret` parameter (and the `oSecondCaret` parameter contained 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. If the position of the line origin was at (50,50), then the starting and ending pen locations of the caret on the screen would be (75,50), (75,75).

```
typedef struct {
    Fixed   fX;
    Fixed   fY;
    Fixed   fDeltaX;
    Fixed   fDeltaY;
}ATSUCaret;
```

Field descriptions

`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.
`fDeltaX`	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.

VERSION NOTES

Available with ATSUI 1.0.

ATSUCustomAllocFunc

If you want total control over memory allocation in ATSUI, including specifying the heap to use and how memory allocation should occur, you should write an application-defined callback function like `MyATSUCustomAllocFunc` (page 178).

You can register your callback function by calling the function `ATSUCreateMemorySetting` (page 174) and passing the `kATSUUseCallbacks` constant in the `iHeapSpec` parameter and a pointer to a `ATSUMemoryCallbacks` (page 193) union in the `iMemoryCallbacks` parameter. The union should contain a pointer of type `ATSUCustomAllocFunc` in the `Alloc` field of the `callbacks` structure.

Your memory allocation callback function should have the following type definition:

```
typedef void * (*ATSUCustomAllocFunc)(void *refCon, ByteCount howMuch);
```

See the function `MyATSUCustomAllocFunc` (page 178) for a prototype of this callback function.

VERSION NOTES

Available with ATSUI 1.1.

ATSUCustomFreeFunc

If you want total control over memory allocation in ATSUI, including specifying the heap to use and how memory allocation should occur, you should write an application-defined callback function like `MyATSUCustomFreeFunc` (page 181).

You can register your callback function by calling the function `ATSUCreateMemorySetting` (page 174) and passing the `kATSUUseCallbacks` constant in the `iHeapSpec` parameter and a pointer to a `ATSUMemoryCallbacks` (page 193) union in the `iMemoryCallbacks` parameter. The union should contain a pointer of type `ATSUCustomFreeFunc` in the `Free` field of the `callbacks` structure.

Your memory allocation callback function should have the following type definition:

```
typedef void * (*ATSUCustomFreeFunc)(void *refCon, void *doomedBlock);
```

See the function `MyATSUCustomFreeFunc` (page 181) for a prototype of this callback function.

VERSION NOTES

Available with ATSUI 1.1.

ATSUCustomGrowFunc

If you want total control over memory allocation in ATSUI, including specifying the heap to use and how memory allocation should occur, you should write an application-defined callback function like `MyATSUCustomGrowFunc` (page 179).

You can register your callback function by calling the function `ATSUCreateMemorySetting` (page 174) and passing the `kATSUUseCallbacks` constant in the `iHeapSpec` parameter and a pointer to a `ATSUMemoryCallbacks` (page 193) union in the `iMemoryCallbacks` parameter. The union should contain a pointer of type `ATSUCustomGrowFunc` in the `Grow` field of the `callbacks` structure.

Your memory allocation callback function should have the following type definition:

```
typedef void * (*ATSUCustomGrowFunc)(void *refCon, void *oldBlock,
ByteCount oldSize, ByteCount newSize);
```

See the function `MyATSUCustomGrowFunc` (page 179) for a prototype of this callback function.

VERSION NOTES

Available with ATSUI 1.1.

ATSUFontFeatureType

Your application passes a value of type `ATSUFontFeatureType` to ATSUI functions that manipulate a style object's font features to specify type of feature to employ.

Each feature type has multiple feature selectors that specify the level or style of its employment. For example, the `kLigaturesType` feature type has on/off pairs of feature selectors, including feature selectors `kRequiredLigaturesOnSelector` and `kRequiredLigaturesOffSelector`. For a description of font feature type and selector constants, see "Font Feature Types and Selectors" (page 271).

```
typedef Uint16 ATSUFontFeatureType;
```

VERSION NOTES

Available with ATSUI 1.0.

ATSUFontFeatureSelector

Your application passes a value of type `ATSUFontFeatureSelector` to ATSUI functions that manipulate a style object's font features to specify the level or style of a feature type.

Each feature type has multiple feature selectors that specify the level or style of its employment. For example, the `kLigaturesType` feature type has on/off pairs

of feature selectors, including feature selectors `kRequiredLigaturesOnSelector` and `kRequiredLigaturesOffSelector`. For a description of font feature type and selector constants, see "Font Feature Types and Selectors" (page 271).

```
typedef Uint16   ATSUFontFeatureSelector;
```

VERSION NOTES

Available with ATSUI 1.0.

ATSUFontID

Your application passes a value of type `ATSUFontID` to ATSUI functions that obtain font information to uniquely identify a font to the font management system in ATSUI.

You can use a value of type `ATSUFontID` to set the font ID of all the glyphs in a style run. To set the font ID of all the glyphs in a style run, pass a value of type `ATSUFontID` and the `kATSUFontIDTag` tag constant to the function `ATSUSetAttributes` (page 29).

```
typedef UInt32       ATSUFontID;
```

VERSION NOTES

Available with ATSUI 1.0.

ATSUFontVariationAxis

Your application passes a value of type `ATSUFontVariationAxis` to ATSUI functions that manipulate a style object's font variations to represent the axis for a particular font variation. Fonts that can generate a wide range of stylistic change contain multiple font variation axes. A **font variation axis** describes a particular stylistic attribute and the range of values that the font can use.

```
typedef FourCharCode        ATSUFontVariationAxis;
```

VERSION NOTES

Available with ATSUI 1.0.

ATSUFontVariationValue

Your application passes a value of type `ATSUFontVariationValue` to ATSUI functions that manipulate a style object's font variations to represent the minimum, maximum, and default values of a font variation axis. These values represent the range of values that the font can use.

```
typedef Fixed    ATSUFontVariationValue;
```

VERSION NOTES

Available with ATSUI 1.0.

ATSUMemoryCallbacks

Your application passes a union of type `ATSUMemoryCallbacks` to the function `ATSUCreateMemorySetting` (page 174). The memory callbacks union should either contain pointers to your application-defined callback functions or pointers to the correctly-prepared memory heap that ATSUI should use when allocating memory.

If you want ATSUI to use your application-defined callback functions for allocating memory, supply pointers to your functions in the `callbacks` structure of this union. If you want ATSUI to use a specific heap (other than the current or application heap) and its own internal callback functions for memory allocation, supply a pointer to the correctly-prepared memory heap in the `heapToUse` field of this union.

```
union ATSUMemoryCallbacks {
    struct {
        ATSUCustomAllocFunc     Alloc;
        ATSUCustomFreeFunc      Free;
        ATSUCustomGrowFunc      Grow;
        void *                  memoryRefCon;
    }                           callbacks;
```

```
    THz                         heapToUse;
};
typedef union ATSUMemoryCallbacksATSUMemoryCallbacks;
```

Field descriptions

`callbacks` A structure containing the `Alloc`, `Free`, `Grow`, and `memoryRefCon` fields. These fields contain pointers to your memory allocation callback functions and arbitrary data for use in your callback functions.

The `Alloc` field contains a pointer of type `ATSUCustomAllocFunc` (page 189) that refers to your application-defined memory allocation callback function. See `MyATSUCustomAllocFunc` (page 178) for the prototype of a memory allocation callback function.

The `Free` field contains a pointer of type `ATSUCustomFreeFunc` (page 190) that refers to your application-defined memory deallocation callback function. See `MyATSUCustomFreeFunc` (page 181) for the prototype of a memory deallocation callback function.

The `Grow` field contains a pointer of type `ATSUCustomFreeFunc` (page 190) that refers to your application-defined memory reallocation callback function. See `MyATSUCustomGrowFunc` (page 179) for the prototype of a memory reallocation callback function.

The `memoryRefCon` field contains a pointer to arbitrary data for use in your application-defined callback functions.

`heapToUse` A pointer of type `THz` that refers to a structure of type `Zone`. Set the structure to the correctly-prepared memory heap for ATSUI to use for simple Memory Manager calls.

VERSION NOTES

Available with ATSUI 1.1.

ATSUMemorySetting

Your application passes a reference of type `ATSUMemorySetting` to the functions `ATSUSetCurrentMemorySetting` (page 176) and `ATSUDisposeMemorySetting`

(page 177) to represent the memory setting you want to make current or dispose of, respectively. The function `ATSUGetCurrentMemorySetting` (page 176) passes back a memory setting reference to represent the current memory setting you want to obtain. A memory setting reference is a pointer to a private structure of type `OpaqueATSUMemorySetting` that contains the memory settings you created in a call to the function `ATSUCreateMemorySetting` (page 174).

```
typedef struct OpaqueATSUMemorySetting*  ATSUMemorySetting;
```

VERSION NOTES

Available with ATSUI 1.1.

ATSUStyle

Your application passes a reference of type `ATSUStyle` to ATSUI functions that manipulate style objects. A style object reference is a pointer to a private structure of type `OpaqueATSUStyle` that contains the style run attributes, font features, and font variations that are set in a particular style run. ATSUI functions that create style objects pass back a style object reference that represents the newly-created style object.

```
typedef struct OpaqueATSUStyle* ATSUStyle;
```

VERSION NOTES

Available with ATSUI 1.0.

ATSUTextLayout

Your application passes a reference of type `ATSUTextLayout` to ATSUI functions that manipulate text layout objects. A text layout object reference is a pointer to a private structure of type `OpaqueATSUTextLayout` that contains the text layout attributes, soft line breaks, and style runs that are set in a particular text layout. ATSUI functions that create text layout objects pass back a text layout object reference that represents the newly-created text layout object.

```
typedef struct OpaqueATSUTextLayout* ATSUTextLayout;
```

ATSUTextMeasurement

Your application passes a value of type `ATSUTextMeasurement` to ATSUI functions that draw, measure, hit-test, and get the glyph bounds of onscreen glyphs to represent exact outline metrics and line specifications.

You can use a value of type `ATSUTextMeasurement` to set the imposed width of all glyphs in a style run and to set the line width, ascent, and descent of each line (or a single line) in a text layout object. To set the imposed width of glyphs in a style run, pass a value of type `ATSUTextMeasurement` and the `kATSUImposeWidthTag` tag constant to the function `ATSUSetAttributes` (page 29).

To set the line width, line ascent, and line descent of each line (or a single line) in a text layout object, pass a value of type `ATSUTextMeasurement` and the `kATSULineWidthTag`, `kATSULineAscentTag`, and `kATSULineDescentTag` tag constants to the functions `ATSUSetLayoutControls` (page 92) and `ATSUSetLineControls` (page 100), respectively.

```
typedef Fixed       ATSUTextMeasurement;
```

VERSION NOTES

Available with ATSUI 1.0.

BslnBaselineRecord

The function `ATSUCalculateBaselineDeltas` (page 35) obtains an array of type `BslnBaselineRecord`. The array contains the distances from a specified baseline to each of other baselines in a style object. If you pass in the style object of the dominant style run in a line, the array represents the optimum baseline positions to control placement of all the glyphs on the line.

You can use the obtained values to set the optimum baseline positions for all glyphs in a single line or in each line of a text layout object. To do this, pass the array and the `kATSULineBaselineValuesTag` tag constant to the functions `ATSUSetLineControls` (page 100) and `ATSUSetLayoutControls` (page 92).

```
typedef Fixed       BslnBaselineRecord[32];
```

VERSION NOTES

Available with ATSUI 1.0.

ConstUniCharArrayPtr

Your application passes a pointer of type `ConstUniCharArrayPtr` to the functions `ATSUCreateTextLayoutWithTextPtr` (page 80), `ATSUSetTextPointerLocation` (page 107), and `ATSUTextMoved` (page 113). The pointer contains the text layout object's text buffer. The text layout object expects the buffer to contain a block of Unicode text. Your application is responsible for allocating the memory associated with this pointer.

```
typedef const UniChar *ConstUniCharArrayPtr;
```

VERSION NOTES

Available with ATSUI 1.0.

UniChar

The `UniChar` value represents a 2-byte Unicode character.

```
typedef UInt16      UniChar;
```

VERSION NOTES

Available with ATSUI 1.0.

UniCharArrayHandle

Your application passes a handle of type `UniCharArrayHandle` to the functions `ATSUCreateTextLayoutWithTextHandle` (page 83) and `ATSUSetTextHandleLocation` (page 109). The handle contains the address of a text layout object's text buffer. The text layout object expects the buffer to contain a block of Unicode text. Your application is responsible for allocating and locking this handle. ATSUI functions will dereference the handle before accessing the text, but will leave the handle's state unchanged.

```
typedef UniCharArrayPtr * UniCharArrayHandle;
```

VERSION NOTES

Available with ATSUI 1.0.

UniCharArrayOffset

Your application can use a value of type `UniCharArrayOffset` to indicate an offset into the buffer of a text layout object. This location is an edge offset in backing-store memory that corresponds to the screen position where text is being drawn, highlighted, measured, deleted, inserted, etc.

You use a value of this type with functions that operate on a range of text in a text layout object.

```
typedef UInt32          UniCharArrayOffset;
```

VERSION NOTES

Available with ATSUI 1.0.

UniCharArrayPtr

Your application passes a pointer of type `UniCharArrayPtr` to ATSUI functions that operate on a range of text in a text layout object. This pointer refers to the address of the beginning of a text layout object's text buffer.

```
typedef UniChar *       UniCharArrayPtr;
```

VERSION NOTES

Available with ATSUI 1.0.

UniCharCount

Your application passes a value of type `UniCharCount` to ATSUI functions that operate on a range of text in a text layout object. This value represents the length of the range of text.

```
typedef UInt32              UniCharCount;
```

VERSION NOTES

Available with ATSUI 1.0.

Resource

This section describes the `'ustl'` clipboard data block format that you can use to describe styled text in the clipboard:

- `ustl` (page 199)

ustl

You can use a clipboard data block format of type `'ustl'` to provide clipboard support and to copy and paste styled text between applications or within an application. You should use this format instead of calling the functions `ATSUCopyToHandle` (page 27) and `ATSUPasteFromHandle` (page 28), because they do not produce the correct data format for displaying ATSUI styled text.

You can store styled text information as a resource of type `'ustl'` or in another data format of type `'ustl'`. However, for the purposes of this document, the clipboard data block format will be described as a resource of type `'ustl'`.

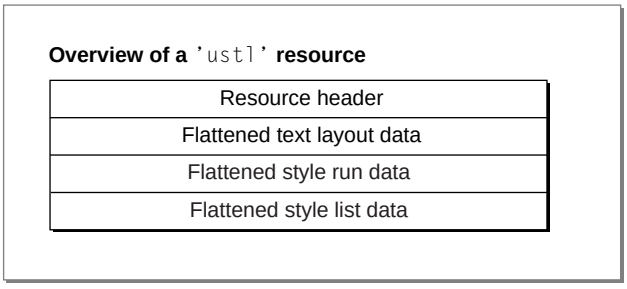
You should use the version of this resource described in this section. Version 0, which was described in previous documentation, should not be used.

Note that all the following fields are 4 bytes except where specified (in parenthesis).

As shown in Figure 2-2, the `'ustl'` resource is composed of four basic elements:

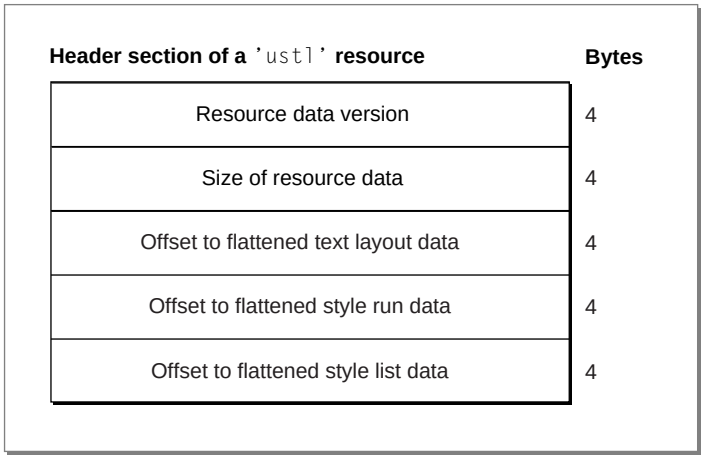
- the resource header
- flattened text layout data
- flattened style run data
- flattened style list data

Figure 2-2 Overview of a `'ustl'` resource



The header section of a `'ustl'` resource contains version and size information for this resource as well as offsets to flattened text layout, style run, and style list data. See Figure 2-3 for an illustration of this section.

Figure 2-3 Header section of a `'ustl'` resource



The elements in the header section of a `'ustl'` resource are

- the version of the data in this resource
- the size of the data in this resource

- an offset to your flattened text layout data
- an offset to your flattened style run data
- an offset to your flattened style list data

After the header section, you must specify either your flattened text layout, style run, or style list data. The order in which you supply these three sets of data is unimportant, although you must follow the order described below within each set.

Flattened Text Layout Data

The elements of the flattened text layout data is

- the number of text layout objects covering the characters that you wish to display on the clipboard
- an array of text layout data.

Each element of the text layout data array should contain the following information:

- the size of the line and text layout attribute data
- the number of characters covered by the text layout object
- an offset to text layout attribute data
- an offset to line attribute data
- an array of text layout attribute data
- an array of line attribute data

Each element of the text layout attribute data array should contain the following information:

- the number of previously set text layout attributes

(for each previously set text layout attribute, including line direction whether or not it has been set):

- the attribute tag
- the size of the attribute value
- the actual attribute value (variable in length)

Each element of the line attribute data array should contain the following information:

- the number of lines in the text layout object

(for each line):

- the line length
- the number of previously set line attributes

(for each previously set line attribute):

- the attribute tag
- the size of the attribute value
- the actual attribute value (variable in length)

Flattened Style Run Data

The elements of the flattened style run data is

- the number of style runs
- an array of style run data

Each element of the style run data array should contain the following information:

- the style run length
- the index of the corresponding style object in the style list

Flattened Style List Data

The elements of the flattened style list data is

- the number of style object
- an array of style run attribute, font feature, and font variation data

Each element of the style run attribute, font feature, and font variation data array should contain:

- the version of the data in this flattened style list
- the size of the style run attribute, font feature, and font variation data
- the number of of previously set style run attributes
- the number of previously set font features
- the number of previously set font variations
- the unique name of font
- the length of font name string

- the font name (variable in length)

(for each previously set style run attribute, and font, text size, and language, even if unset):

- the attribute tag
- the size of the attribute value
- the actual attribute value (variable in length)

(for each previously set font feature):

- the feature type
- the feature selector

(for each previously set font variation):

- the variation axis
- the variation value

VERSION NOTES

Version 0 of this format is available with ATSUI 1.0. Version 1 is available with ATSUI 1.1.

Constants

This section describes the constants defined by ATSUI for your application's use. The constants are organized into the following categories:

Baseline Type Constants

Your application passes a constant of type `kATSUBaselineClassTag` in the `iBaselineClass` parameter of the function `ATSUCalculateBaselineDeltas` (page 35). This value represents the baseline that you want `ATSUCalculateBaselineDeltas` to use in calculating the distances to each of the other baseline types in the style run. If you want to specify that `ATSUCalculateBaselineDeltas` use the standard baseline value from the current font, pass the constant `kBSLNNoBaselineOverride`.

You can also use a constant of this type to set the primary baseline in a style run. To do this, pass the desired baseline value and the `kATSUBaselineClassTag` tag constant to the function `ATSUSetAttributes` (page 29).

```
enum {
    kBSLNRomanBaseline                  = 0,
    kBSLNIdeographicCenterBaseline      = 1,
    kBSLNIdeographicLowBaseline         = 2,
    kBSLNHangingBaseline                = 3,
    kBSLNMathBaseline                   = 4,
    kBSLNLastBaseline                   = 31,
    kBSLNNumBaselineClasses             = kBSLNLastBaseline + 1,
    kBSLNNoBaselineOverride             = 255
};
typedef UInt32              BslnBaselineClass;
```

Constant description

`kBSLNRomanBaseline` Represents the baseline used by most Roman script languages, and in Arabic and Hebrew. This is the default value.

`kBSLNIdeographicCenterBaseline`
Represents the baseline used by Chinese, Japanese, and Korean ideographic scripts, in which ideographs are centered halfway on the line height.

`kBSLNIdeographicLowBaseline`
Represents the baseline used by Chinese, Japanese, and Korean scripts. Similar to `kBSLNIdeographicCenterBaseline`, but with the glyphs lowered. This baseline is most commonly used to align Roman glyphs within ideographic fonts to Roman glyphs in Roman fonts.

`kBSLNHangingBaseline`
Represents the baseline used by Devanagari and related scripts, in which the bulk of most glyphs is below the baseline. This baseline type is also used for drop capitals in Roman scripts.

`kBSLNMathBaseline` Represents the baseline used for setting mathematics. It is centered on symbols such as the minus sign (at half the x-height).

`kBSLNLastBaseline` No baseline type may exceed this value. Application-defined baseline values between `kBSLNMathBaseline` and `kBSLNLastBaseline` are reserved.

`kBSLNNumBaselineClasses`
Represents the total number of baseline types (`kBSLNLastBaseline` + 1).

`kBSLNNoBaselineOverride`
Instructs ATSUI to use the standard baseline value from the current font.

VERSION NOTES

Available with ATSUI 1.0.

Clear All Constant

Your application can pass this constant to several different functions to remove previously set values from a style object: to `ATSUClearAttributes` (page 34) to remove style run attributes, to `ATSUClearFontFeatures` (page 41) to remove font features, and to `ATSUClearFontVariations` (page 46) to remove font variations.

You can also use this constant to remove previously set values from a line or text layout object: to `ATSUClearLineControls` (page 105), to remove text layout attributes from a single line, and to `ATSUClearLayoutControls` (page 97) to remove .

```
enum {
    kATSUClearAll           = (long)0xFFFFFFFF
};
```

Constant description

`kATSUClearAll` Removes all previously set values from a style object, a single line, or a text layout object.

VERSION NOTES

Available with ATSUI 1.0.

Current Pen Location Constant

Your application can pass the `kATSUUseGrafPortPenLoc` constant to ATSUI 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.

```
enum {
    kATSUUseGrafPortPenLoc  = (long)0xFFFFFFFF,
};
```

Constant description

`kATSUUseGrafPortPenLoc`
Indicates that drawing, measuring, or hit-testing should be done relative to the current pen location in the current graphics port. ATSUI looks at the current graphics port location (it knows the last draw location from moves and lineto calls) and uses that.

VERSION NOTES

Available with ATSUI 1.0.

Cursor Movement Constants

Your application can a constant of type `ATSUCursorMovementType` to several different functions to represent the distance the cursor has moved: to `ATSUNextCursorPosition` (page 134) to determine the next cursor position in `ATSUPreviousCursorPosition` (page 136), `ATSURightwardCursorPosition` (page 138), and `ATSULeftwardCursorPosition` (page 140). This value represents the distance that the cursor has moved.

```
enum {
    kATSUByCharacter    = 0,
    kATSUByCluster      = 1,
    kATSUByWord         = 2
};
typedef int ATSUCursorMovementType;
```

Constant descriptions

`kATSUByCharacter` Indicates that the cursorhas moved one Unicode character (that is, 2 bytes). Note that because of surrogate pairs, you can not always move the cursor by one character, since

doing so might place the cursor in the middle of a surrogate pair rahter than between two logical characters. For more information on surrogate pairs, see "ATSUI Implementation of the Unicode Specification" (page 265).

`kATSUByCluster` Indicates that the cursorhas moved one cluster, as defined by Unicode. A group of characters is a cluster based both on the static properties of the characters involved (defined by the Unicode consortium) and the behavior of the specific font you are using with those characters.

`kATSUByWord` Indicates to ATSUI to move the cursor by one word, as defined by Unicode. In Unicode, a word does not include trailing punctuation or white space.

VERSION NOTES

Available with ATSUI 1.0.

Font Fallback Constants

Your application passes a constant of type `ATSUFontFallbackMethod` in the `iFontFallbackMethod` parameter of the function `ATSUSetFontFallbacks` (page 119) the search options to employ when a font does not have all the glyphs for the characters it is trying to draw. The function `ATSUGetFontFallbacks` (page 120) passes back one of these constants to indicate the previously-set search order. To specify the default search behavior, pass the constant `kATSUDefaultFontFallbacks`.

```
enum {
    kATSUDefaultFontFallbacks           = 0,
    kATSULastResortOnlyFallback         = 1,
    kATSUSequentialFallbacksPreferred   = 2,
    kATSUSequentialFallbacksExclusive   = 3
};
typedef UInt16      ATSUFontFallbackMethod;
```

Constant descriptions

`kATSUDefaultFontFallbacks`

Tells ATSUI to sequentially scan the font list and then search all valid fonts in the user's system. This is the

default value and the search order used by the functions `ATSUMatchFontsToText` (page 122) and `ATSUSetTransientFontMatching` (page 124).

`kATSULastResortOnlyFallback`
Tells ATSUI not to scan the font list and instead use the last resort font.

`kATSUSequentialFallbacksPreferred`
Tells ATSUI to sequentially scan the font list, then search the remaining valid fonts on the user's system, and finally use the fallback font if a useful fallback font is not found in either location.

`kATSUSequentialFallbacksExclusive`
Tells ATSUI to sequentially scan the font list, then use the last resort font.

VERSION NOTES

Available with ATSUI 1.1.

Font Name Language Code Constants

The `FontLanguageCode` enumeration defines constants your application can use to identify or obtain the language of a font name string in a font name table. You can use a constant of this type with the functions `ATSUFindFontName` (page 56) and `ATSUFindFontFromName` (page 50) to identify the language of the font name string you want to find in a font name table. You get a platform code constant passed back from the function `ATSUGetIndFontName` (page 54) to represent the language of the font name string corresponding to a passed in font name index.

The font name language code identifies the language of the name string. Specifying the language of a font name string is optional. If you do not care about its language, pass the `kFontNoLanguage` constant, described in "No Font Name Platform, Language, or Script Constants" (page 235). In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first name string in the name table that matches the name code and other parameters that you specified.

```
enum {
    kFontEnglishLanguage            = 0,
    kFontFrenchLanguage             = 1,
    kFontGermanLanguage             = 2,
    kFontItalianLanguage            = 3,
    kFontDutchLanguage              = 4,
    kFontSwedishLanguage            = 5,
    kFontSpanishLanguage            = 6,
    kFontDanishLanguage             = 7,
    kFontPortugueseLanguage         = 8,
    kFontNorwegianLanguage          = 9,
    kFontHebrewLanguage             = 10,
    kFontJapaneseLanguage           = 11,
    kFontArabicLanguage             = 12,
    kFontFinnishLanguage            = 13,
    kFontGreekLanguage              = 14,
    kFontIcelandicLanguage          = 15,
    kFontMalteseLanguage            = 16,
    kFontTurkishLanguage            = 17,
    kFontCroatianLanguage           = 18,
    kFontTradChineseLanguage        = 19,
    kFontUrduLanguage               = 20,
    kFontHindiLanguage              = 21,
    kFontThaiLanguage               = 22,
    kFontKoreanLanguage             = 23,
    kFontLithuanianLanguage         = 24,
    kFontPolishLanguage             = 25,
    kFontHungarianLanguage          = 26,
    kFontEstonianLanguage           = 27,
    kFontLettishLanguage            = 28,
    kFontLatvianLanguage            = kFontLettishLanguage,
    kFontSaamiskLanguage            = 29,
    kFontLappishLanguage            = kFontSaamiskLanguage,
    kFontFaeroeseLanguage           = 30,
    kFontFarsiLanguage              = 31,
    kFontPersianLanguage            = kFontFarsiLanguage,
    kFontRussianLanguage            = 32,
    kFontSimpChineseLanguage        = 33,
    kFontFlemishLanguage            = 34,
    kFontIrishLanguage              = 35,
    kFontAlbanianLanguage           = 36,
```

```
    kFontRomanianLanguage          = 37,
    kFontCzechLanguage             = 38,
    kFontSlovakLanguage            = 39,
    kFontSlovenianLanguage         = 40,
    kFontYiddishLanguage           = 41,
    kFontSerbianLanguage           = 42,
    kFontMacedonianLanguage        = 43,
    kFontBulgarianLanguage         = 44,
    kFontUkrainianLanguage         = 45,
    kFontByelorussianLanguage      = 46,
    kFontUzbekLanguage             = 47,
    kFontKazakhLanguage            = 48,
    kFontAzerbaijaniLanguage       = 49,
    kFontAzerbaijanArLanguage      = 50,
    kFontArmenianLanguage          = 51,
    kFontGeorgianLanguage          = 52,
    kFontMoldavianLanguage         = 53,
    kFontKirghizLanguage           = 54,
    kFontTajikiLanguage            = 55,
    kFontTurkmenLanguage           = 56,
    kFontMongolianLanguage         = 57,
    kFontMongolianCyrLanguage      = 58,
    kFontPashtoLanguage            = 59,
    kFontKurdishLanguage           = 60,
    kFontKashmiriLanguage          = 61,
    kFontSindhiLanguage            = 62,
    kFontTibetanLanguage           = 63,
    kFontNepaliLanguage            = 64,
    kFontSanskritLanguage          = 65,
    kFontMarathiLanguage           = 66,
    kFontBengaliLanguage           = 67,
    kFontAssameseLanguage          = 68,
    kFontGujaratiLanguage          = 69,
    kFontPunjabiLanguage           = 70,
    kFontOriyaLanguage             = 71,
    kFontMalayalamLanguage         = 72,
    kFontKannadaLanguage           = 73,
    kFontTamilLanguage             = 74,
    kFontTeluguLanguage            = 75,
    kFontSinhaleseLanguage         = 76,
    kFontBurmeseLanguage           = 77,
```

```
    kFontKhmerLanguage              = 78,
    kFontLaoLanguage                = 79,
    kFontVietnameseLanguage         = 80,
    kFontIndonesianLanguage         = 81,
    kFontTagalogLanguage            = 82,
    kFontMalayRomanLanguage         = 83,
    kFontMalayArabicLanguage        = 84,
    kFontAmharicLanguage            = 85,
    kFontTigrinyaLanguage           = 86,
    kFontGallaLanguage              = 87,
    kFontOromoLanguage              = kFontGallaLanguage,
    kFontSomaliLanguage             = 88,
    kFontSwahiliLanguage            = 89,
    kFontRuandaLanguage             = 90,
    kFontRundiLanguage              = 91,
    kFontChewaLanguage              = 92,
    kFontMalagasyLanguage           = 93,
    kFontEsperantoLanguage          = 94,
    kFontWelshLanguage              = 128,
    kFontBasqueLanguage             = 129,
    kFontCatalanLanguage            = 130,
    kFontLatinLanguage              = 131,
    kFontQuechuaLanguage            = 132,
    kFontGuaraniLanguage            = 133,
    kFontAymaraLanguage             = 134,
    kFontTatarLanguage              = 135,
    kFontUighurLanguage             = 136,
    kFontDzongkhaLanguage           = 137,
    kFontJavaneseRomLanguage        = 138,
    kFontSundaneseRomLanguage       = 139
};
typedef UInt32          FontLanguageCode;
```

VERSION NOTES

Available with ATSUI 1.1.

Font Name Code Constants

The `FontNameCode` enumeration defines constants your application can use to identify or obtain the type of font name string in a font name table. You can use a constant of this type with the functions `ATSUFindFontName` (page 56) and `ATSUFindFontFromName` (page 50) to identify the type of font name string you want to find in a font name table. You get a name code constant passed back from the function `ATSUGetIndFontName` (page 54) to represent the type of the font name string corresponding to a passed in font name index.

The font name code identifies a name string that provides various kinds of information about the font, including its unique name, font family name, style, version number, and Postscript-legible name. You can specify values between `kFontLicenseInfoURLName` and `kFontLastReservedName` to find other types of information about the font, including the name of a particular font variation axis, feature, tracking setting, or instance.

You must always specify a constant of this type when you are searching for a font name string in a font name table. Optionally, you can also specify the platform, script, and language of the font name string. You should pass these values if you want to find a localized string. This is not the case with the unique name and Postscript-legible name of a font, which only have one font name string regardless of the platform, script, and language you specify.

For all other font names, the actual font name string identified by a name code constant will vary based on the platform, script, and language of the font name. For example, the name code constant `kFontCopyrightName` might identify three different strings of the font manufacturer's copyright notice name, one in Unicode French, one in Macintosh English, and one in Microsoft German.

```
enum {
    kFontCopyrightName          = 0,
    kFontFamilyName             = 1,
    kFontStyleName              = 2,
    kFontUniqueName             = 3,
    kFontFullName               = 4,
    kFontVersionName            = 5,
    kFontPostscriptName         = 6,
    kFontTrademarkName          = 7,
    kFontManufacturerName       = 8,
    kFontDesignerName           = 9,
    kFontDescriptionName        = 10,
    kFontVendorURLName          = 11,
```

```
    kFontDesignerURLName        = 12,
    kFontLicenseDescriptionName = 13,
    kFontLicenseInfoURLName     = 14,
    kFontLastReservedName       = 255
};
typedef UInt32          FontNameCode;
```

Constant descriptions

`kFontCopyrightName` Identifies the font manufacturer's copyright notice name. An example of a font name string that might be accessed by this name code for Mac Roman English is "© Apple Computer, Inc. 1993".

`kFontFamilyName` Identifies the font family name, which is shared by all styles in a font family. An example of a font name string that might be accessed by this name code for Mac Roman English is "Times".

`kFontStyleName` Identifies the font style. An example of a font name string that might be accessed by this name code for Mac Roman English is "Regular", "Italic", "Bold", or "Black".

`kFontUniqueName` Identifies the manufacturer's name for the font. Because the name identified by this constant can be used to uniquely identify the font, you should use it in stored documents and in program interchange to identify fonts. The unique name is used in the standard clipboard format. An example of a font name string that might be accessed by this name code for Mac Roman English is "Apple Computer Times Black 3.0 8/10/92".

`kFontFullName` Identifies the full font name. An example of a font name string that might be accessed by this name code for Mac Roman English is "Times Black".

`kFontVersionName` Identifies the font manufacturer's version number for the font. An example of a font name string that might be accessed by this name code for Mac Roman English is "3.0." (The name does not need to include the word "version").

`kFontPostscriptName`
Identifies the PostScript-legible name of the font. This type of font name can be used to uniquely identify the font. An

example of a font name string that might be accessed by this name code for Mac Roman English is "Times-Black".

`kFontTrademarkName`

Identifies the font trademark name. An example of a font name string that might be accessed by this name code for Mac Roman English is "Palatino is a registered trademark of Linotype AG".

`kFontManufacturerName`

Identifies the font manufacturer's name. An example of a font name string that might be accessed by this name code for Mac Roman English is "Apple Computer, Inc."

`kFontDesignerName` Identifies the font family designer's name.

`kFontDescriptionName`

Identifies the description of the font family.

`kFontVendorURLName` Identifies the uniform resource locator of the font vendor. If a unique serial number is embedded in the URL, it can be used to register the font.

`kFontDesignerURLName`

Identifies the uniform resource locator of the font family designer.

`kFontLicenseDescriptionName`

Identifies the uniform resource locator of the font vendor. If a unique serial number is embedded in the URL, it can be used to register the font.

`kFontLicenseInfoURLName`

Identifies the uniform resource locator of the font vendor. If a unique serial number is embedded in the URL, it can be used to register the font.

`kFontLastReservedName`

No font name code may exceed this value. Name code values between `kFontLicenseInfoURLName` and `kFontLastReservedName` are reserved.

VERSION NOTES

Available with ATSUI 1.0.

Font Name Platform Code Constants

The `FontPlatformCode` enumeration defines constants your application can use to identify or obtain the encoding of a font name string in a font name table. You can use a constant of this type with the functions `ATSUFindFontName` (page 56) and `ATSUFindFontFromName` (page 50) to identify the encoding of the font name string you want to find in a font name table. You get a platform code constant passed back from the function `ATSUGetIndFontName` (page 54) to represent the encoding of the font name string corresponding to a passed in font name index.

The font name platform code identifies the encoding of the name string, which ATSUI uses to determine whether or not it can render the string. A font can support multiple encodings. If the encoding of the name string is not Unicode, you will have to translate it into Unicode using the Text Encoding Converter.

Specifying the encoding of a font name string is optional. If you do not care about its encoding, pass the `kFontNoPlatform` constant, described in "No Font Name Platform, Language, or Script Constants" (page 235). In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first name string in the name table that matches the name code and other parameters that you specified. If you pass `kFontNoPlatform`, you should pass `kFontNoScript` in the script code parameter, since a script code makes no sense without a platform.

IMPORTANT

Due to an error on the part of font developers, Unicode-encoded font name entries have 8-bit instead of the expected 16-bit names. As a result, if you wish to find a Unicode-encoded font name string, you should pass the font platform code constant `kFontMacintoshPlatform` and a script code constant the represents the platform-specific ID of Unicode encoding you wish to find. See "Font Name Script Code Constants" (page 217) for a description of the script codes corresponding to the Unicode platform. ▲

```
enum {
    kFontUnicodePlatform    = 0,
    kFontMacintoshPlatform  = 1,
    kFontReservedPlatform   = 2,
    kFontMicrosoftPlatform  = 3,
    kFontCustomPlatform     = 4
};
typedef UInt32          FontPlatformCode;
```

Constant descriptions

`kFontUnicodePlatform`
: The platform uses the Unicode character code specifications. For more information about the Unicode encodings, see *The Unicode Standard: Worldwide Character Encoding*, volumes 1 and 2, available from Addison-Wesley.

`kFontMacintoshPlatform`
: The platform uses one of the Macintosh character code sets.

`kFontReservedPlatform`
: The platform is reserved for future use.

`kFontMicrosoftPlatform`
: The platform uses one of the Microsoft character code sets.

`kFontCustomPlatform`
: This is a nonstandard platform, specific to the font, in which the encoding of a font does not correspond to a specific standard.

VERSION NOTES

Available with ATSUI 1.0.

Font Name Script Code Constants

The `FontScriptCode` enumeration defines constants your application can use to identify or obtain the platform-specific ID of a font name string in a font name table. You can use a constant of this type with the functions `ATSUFindFontName` (page 56) and `ATSUFindFontFromName` (page 50) to identify the platform-specific ID of the font name string you want to find in a font name table. You get a platform code constant passed back from the function `ATSUGetIndFontName` (page 54) to represent the platform-specific ID of the font name string corresponding to a passed in font name index.

The font name script code identifies the platform version, or in the case of the Macintosh platform, the script ID of the font name. The script ID identifies the writing system being used (for example, MacRoman). A font can support multiple encodings. If the encoding of the name string is not Unicode, you will have to translate it into Unicode using the Text Encoding Converter.

Specifying the version of the platform (or script ID) of a font name string is optional. If you do not care about this value, pass the `kFontNoScript` constant, described in "No Font Name Platform, Language, or Script Constants" (page 235). In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first name string in the name table that matches the name code and other parameters that you specified.

```
enum {
    kFontUnicodeDefaultSemantics    = 0,
    kFontUnicodeV1_1Semantics       = 1,
    kFontISO10646_1993Semantics     = 2,
    kFontUnicodeV2BasedSemantics    = 3
```

Constant descriptions

`kFontUnicodeDefaultSemantics`
The platform uses the default Unicode character code specifications.

`kFontUnicodeV1_1Semantics`
The platform uses version 1.1 of the Unicode character code specifications.

`kFontISO10646_1993Semantics`
The platform uses the ISO/IEC 10646-1993 specifications.

`kFontUnicodeV2BasedSemantics`
The platform uses version 2.0 or later of the Unicode character code specifications.

```
enum {
    kFontRomanScript                = 0,
    kFontJapaneseScript             = 1,
    kFontTraditionalChineseScript   = 2,
    kFontChineseScript              = kFontTraditionalChineseScript,
    kFontKoreanScript               = 3,
    kFontArabicScript               = 4,
    kFontHebrewScript               = 5,
    kFontGreekScript                = 6,
    kFontCyrillicScript             = 7,
    kFontRussian                    = kFontCyrillicScript,
    kFontRSymbolScript              = 8,
    kFontDevanagariScript           = 9,
    kFontGurmukhiScript             = 10,
```

```
    kFontGujaratiScript             = 11,
    kFontOriyaScript                = 12,
    kFontBengaliScript              = 13,
    kFontTamilScript                = 14,
    kFontTeluguScript               = 15,
    kFontKannadaScript              = 16,
    kFontMalayalamScript            = 17,
    kFontSinhaleseScript            = 18,
    kFontBurmeseScript              = 19,
    kFontKhmerScript                = 20,
    kFontThaiScript                 = 21,
    kFontLaotianScript              = 22,
    kFontGeorgianScript             = 23,
    kFontArmenianScript             = 24,
    kFontSimpleChineseScript        = 25,
    kFontTibetanScript              = 26,
    kFontMongolianScript            = 27,
    kFontGeezScript                 = 28,
    kFontEthiopicScript             = kFontGeezScript,
    kFontAmharicScript              = kFontGeezScript,
    kFontSlavicScript               = 29,
    kFontEastEuropeanRomanScript    = kFontSlavicScript,
    kFontVietnameseScript           = 30,
    kFontExtendedArabicScript       = 31,
    kFontSindhiScript               = kFontExtendedArabicScript,
    kFontUninterpretedScript        = 32
};
```

Constant descriptions

`kFontRomanScript` Represents the Roman script on the Macintosh platform.

`kFontJapaneseScript` Represents the Japanese script on the Macintosh platform.

`kFontTraditionalChineseScript` Represents the traditional Chinese script on the Macintosh platform.

`kFontChineseScript` Represents the Chinese script on the Macintosh platform.

`kFontKoreanScript` Represents the Korean script on the Macintosh platform.

`kFontArabicScript` Represents the Arabic script on the Macintosh platform.

`kFontHebrewScript` Represents the Hebrew script on the Macintosh platform.

`kFontGreekScript` Represents the Greek script on the Macintosh platform.

`kFontCyrillicScript` Represents the Cyrillic script on the Macintosh platform.

`kFontRussian` Represents the Russian script on the Macintosh platform.

`kFontRSymbolScript` Represents the right-to-left symbol script on the Macintosh platform.

`kFontDevanagariScript` Represents the Devanagari script on the Macintosh platform.

`kFontGurmukhiScript` Represents the Gurmukhi script on the Macintosh platform.

`kFontGujaratiScript` Represents the Gujarati script on the Macintosh platform.

`kFontOriyaScript` Represents the Oriya font script on the Macintosh platform.

`kFontBengaliScript` Represents the Benagli script on the Macintosh platform.

`kFontTamilScript` Represents the Tamil script on the Macintosh platform.

`kFontTeluguScript` Represents the Telugu script on the Macintosh platform.

`kFontKannadaScript` Represents the Kannada script on the Macintosh platform.

`kFontMalayalamScript` Represents the Malayalam script on the Macintosh platform.

`kFontSinhaleseScript` Represents the Sinhalese script on the Macintosh platform.

`kFontBurmeseScript` Represents the Burmese script on the Macintosh platform.

`kFontKhmerScript` Represents the Khmer script on the Macintosh platform.

`kFontThaiScript` Represents the Thai script on the Macintosh platform.

`kFontLaotianScript` Represents the Laotian script on the Macintosh platform.

`kFontGeorgianScript` Represents the Georgian script on the Macintosh platform.

`kFontArmenianScript` Represents the Armenian script on the Macintosh platform.

`kFontSimpleChineseScript` Represents the simplified Chinese script on the Macintosh platform.

`kFontTibetanScript` Represents the Tibetan script on the Macintosh platform.

`kFontMongolianScript` Represents the Mongolian script on the Macintosh platform.

`kFontGeezScript` Represents the Ge'ez script on the Macintosh platform.

`kFontEthiopicScript`
Represents the Ethiopic script on the Macintosh platform.

`kFontAmharicScript` Represents the Amharic script on the Macintosh platform.

`kFontSlavicScript` Represents the Slavic script on the Macintosh platform.

`kFontEastEuropeanRomanScript`
Represents the East European script on the Macintosh platform.

`kFontVietnameseScript`
Represents the Vietnamese script on the Macintosh platform.

`kFontExtendedArabicScript`
Represents the extended Arabic script on the Macintosh platform.

`kFontSindhiScript` Represents the Sindhi script on the Macintosh platform.

`kFontUninterpretedScript`
Represents an uninterpreted script on the Macintosh platform.

```
enum {
    kFontMicrosoftSymbolScript      = 0,
    kFontMicrosoftStandardScript    = 1
};
```

Constant descriptions

`kFontMicrosoftSymbolScript`
Represents the symbol version of the Microsoft platform.

`kFontMicrosoftStandardScript`
Represents the standard version of the Microsoft platform.

```
enum {
    kFontCustom8BitScript       = 0,
    kFontCustom816BitScript     = 1,
    kFontCustom16BitScript      = 2
};
```

Constant descriptions

`kFontCustom8BitScript`
Represents custom 8-bit encoding.

`kFontCustom816BitScript`
Represents custom mixed 8-/16-bit encoding.

`kFontCustom16BitScript`
Represents custom 16-bit encoding.

Glyph Bounds Constants

Your application passes a glyph bounds constant in the `iTypeOfBounds` parameter of the function `ATSUGetGlyphBounds` (page 145) to indicate whether the width of the resulting typographic glyph bounds will be determined using the caret origin, glyph origin in device space, or glyph origin in fractional absolute positions.

```
enum {
    kATSUseCaretOrigins         = 0,
    kATSUseDeviceOrigins        = 1,
    kATSUseFractionalOrigins    = 2
};
```

Constant descriptions

`kATSUseCaretOrigins`
Specifies that the width of the typographic glyph bounds will be determined using the caret origin. The caret origin is halfway between two characters.

`kATSUseDeviceOrigins`
Specifies that the width of the typographic glyph bounds will be determined using the glyph origin in device space. This is useful for adjusting text on the screen.

`kATSUseFractionalOrigins`
Specifies that the width of the typographic glyph bounds will be determined using the glyph origin in fractional absolute positions, which are uncorrected for device display. This provides the ideal position of laid-out text and is useful for scaling text on the screen. This origin is also used to get the width of the typographic bounding rectangle when you call `ATSUMeasureText` (page 148).

VERSION NOTES

Available with ATSUI 1.1.

Glyph Direction Constants

Your application can use a glyph direction constant to set the direction of glyphs in a text layout object, regardless of their natural direction as specified in the font. To set a glyph direction in a single line or entire text layout object, pass the `kATSULineDirectionTag` tag and a constant of this type to the functions `ATSUSetLineControls` (page 100) and `ATSUSetLayoutControls` (page 92).

```
enum {
    kATSULeftToRightBaseDirection   = 0,
    kATSURightToLeftBaseDirection   = 1
};
```

Constant descriptions

`kATSULeftToRightBaseDirection`
Imposes left-to-right direction on glyphs in a line or in an entire text layout object regardless of their natural direction as specified in the font. For vertical text, imposes top-to-bottom direction onto the glyphs.

`kATSURightToLeftBaseDirection`
Imposes right-to-left direction on glyphs in a line or in an entire text layout object regardless of their natural direction as specified in the font. For vertical text, imposes bottom-to-top direction onto the glyphs.

VERSION NOTES

Available with ATSUI 1.0.

Glyph Orientation Constants

Your application can pass a constant of type `ATSUVerticalCharacterType` to the functions `ATSUCountFontTracking` (page 61) and `ATSUGetIndFontTracking` (page 62) to represent the glyph orientation for the font tracking settings you want to count, and for the tracking setting and name code you want to obtain. It

is necessary to specify this value because there are different font tracking settings for different glyph orientations.

You can also use a constant of type `ATSUVerticalCharacterType` to set the glyph orientation in a style run. To set a style run's glyph orientation, pass the `kATSUVerticalCharacterTag` tag and a constant of this type to the function `ATSUSetAttributes` (page 29).

```
enum {
    kATSUStronglyHorizontal     = 0,
    kATSUStronglyVertical       = 1
};
typdef UInt16   ATSUVerticalCharacterType
```

Constant descriptions

`kATSUStronglyHorizontal`
Specifies non-rotated glyphs that are drawn with horizontal metrics.

`kATSUStronglyVertical`
Specifies that glyphs are rotated 90 degrees and are drawn with vertical metrics.

VERSION NOTES

Available with ATSUI 1.0.

Heap Specification Constants

Your application passes a constant of type `ATSUHeapSpec` to the function `ATSUCreateMemorySetting` (page 174) to specify either the desired heap to use or the callback functions ATSUI should use in performing memory allocation operations.

If you pass the `kATSUUseSpecificHeap` constant in the `iHeapSpec` parameter, you must specify the correctly-prepared heap in the `heapToUse` field of the `ATSUMemoryCallbacks` (page 193) union. If you pass the `kATSUUseCallbacks` constant in the `iHeapSpec` parameter, you must supply pointers to your application in the `callback` structure of the `ATSUMemoryCallbacks` (page 193) union. If you pass the `kATSUUseCurrentHeap` or `kATSUUseAppHeap` constant in the

iHeapSpec parameter, pass a NULL pointer to the ATSUMemoryCallbacks (page 193) union.

```
enum {
    kATSUUseCurrentHeap         = 0,
    kATSUUseAppHeap             = 1,
    kATSUUseSpecificHeap        = 2,
    kATSUUseCallbacks           = 3
};
typedef UInt32          ATSUHeapSpec;
```

Constant descriptions

kATSUUseCurrentHeap
: Indicates to ATSUI to perform memory allocation operations on the heap that is current at the time you call ATSUCreateMemorySetting. If you pass this constant in the iHeapSpec parameter, you must pass a pointer to a ATSUMemoryCallbacks (page 193) union that contains the correctly-prepared heap in the heapToUse field in the parameter. This is the default value if you do not call ATSUCreateMemorySetting.

kATSUUseAppHeap
: Indicates to ATSUI to perform memory allocation operations only on the application heap, whether or not it is the current heap.

kATSUUseSpecificHeap
: Indicates to ATSUI to perform memory allocation operations on the heap identified in the heapToUse field of the ATSUMemoryCallbacks (page 193) union.

kATSUUseCallbacks
: Indicates to ATSUI to use your application-defined memory callback functions pointed to in the Alloc, Grow, and Free fields of the callbacks structure in the ATSUMemoryCallbacks (page 193) union.

VERSION NOTES

Available with ATSUI 1.1.

Invalid Font ID Constant

The functions `ATSUFONDtoFontID` (page 59), `ATSUFindFontFromName` (page 50), and `ATSUMatchFontsToText` (page 122) pass back this constant to indicate that the passed in font ID is invalid.

```
enum {
    kATSUInvalidFontID      = 0
};
```

Constant description

`kATSUInvalidFontID` Indicates that the font ID is invalid.

VERSION NOTES

Available with ATSUI 1.0.

Justification Override Mask Constants

Your application supplies a mask of type `JustificationFlags` in the `growFlags` and `shrinkFlags` fields of a `ATSJustPriorityWidthDeltaOverrides` (page 183) delta structure, described in `ATSJustPriorityWidthDeltaOverrides` (page 183). control which aspects of the normal, font-specified justification behavior for a particular set of glyphs you want to override. This mask applies only to the glyphs of the style run that the width delta structure applies to. You specify this mask in the `growFlags` and `shrinkFlags` fields of a width delta structure, described in `ATSJustPriorityWidthDeltaOverrides` (page 183).

Note that these flags are a specialized override. You should not use them unless you know the details of what is occurring in the font.

```
enum {
    kJUSTOverridePriority           = 0x8000,
    kJUSTOverrideLimits             = 0x4000,
    kJUSTOverrideUnlimited          = 0x2000,
    kJUSTUnlimited                  = 0x1000,
    kJUSTPriorityMask               = 0x0003
};
typedef UInt16              JustificationFlags;
```

Constant descriptions

`kJUSTOverridePriority`
: Determines whether ATSUI should use or override the default priority. If this flag is set, ATSUI uses the justification priority set in the `kJUSTPriorityMask` mask. If this flag is cleared, ATSUI uses the default justification priority for those glyphs. In this case, the `kJUSTPriorityMask` mask bits must also be set to 0.

`kJUSTOverrideLimits`
: Determines whether ATSUI should use or override the default shrink and grow limits. If this flag is set, ATSUI uses the grow and shrink limit values set in the `ATSJustPriorityWidthDeltaOverrides` (page 183) structure. If this flag is cleared, ATSUI uses the default grow and shrink limits for those glyphs. In this case, the limits values in the width delta structure must also be set to 0.

`kJUSTOverrideUnlimited`
: Determines whether ATSUI should apply the `kJUSTUnlimited` flag. If this flag is set, ATSUI takes into account the state of the `kJUSTUnlimited` flag. If this flag is cleared, the `kJUSTUnlimited` flag must also be set to 0.

`kJUSTUnlimited`
: Determines whether ATSUI should distribute all remaining justification gap to the glyphs specified in the `ATSJustPriorityWidthDeltaOverrides` (page 183) structure. If this flag is set, ATSUI distributes all remaining justification gap, even if it violates the grow or shrink limits specified in the structure. If this flag is not zero, you must also set the `kJUSTOverrideUnlimited` bit.

`kJUSTPriorityMask`
: Identifies the new justification priority for the glyphs this width delta structure applies to. See "Justification Priority Constants" (page 228) for a description of possible values. Only a single valid justification priority value is permitted. If this flag is set, the `kJUSTOverrideLimits` bit must also be set. To make the justification priority of white space glyphs the same as intercharacter priority, set this bit to `ATSUInterCharPriority`.

VERSION NOTES

Available with ATSUI 1.0.

Justification Priority Constants

Glyphs in a font can be assigned justification priorities by the font designer. In general, ATSUI applies justification to glyphs on a line in order of glyph priority, from highes to lowest. You can set a new justification priority for the glyphs a particular width delta structure applies to by setting the `kJUSTPriorityMask` flag of the justification flags.

The justification priorities have names that describe the types of glyphs that typically have those priorities, but you can assign any priority to any glyph. The actual kind of justification that ATSUI applies—for example, kashida or whitespace—is defined for each glyph by the font. The priority specifies only the order in which glyphs participate in justification.

```
enum {
    kJUSTKashidaPriority                    = 0,
    kJUSTSpacePriority                      = 1,
    kJUSTLetterPriority                     = 2,
    kJUSTNullPriority                       = 3,
    kJUSTPriorityCount                      = 4
};
typedef UInt16 JustPCActionType;
```

Constant descriptions

`kJUSTKashidaPriority`
: The highest priority. Typically used for kashidas (extension bars) in Arabic. Glyphs with this priority are extended or compressed before all other glyphs in the line.

`kJUSTSpacePriority`
: Typically assigned to whitespace (interword) glyphs. Glyphs with this priority are extended or compressed, usually by the addition or removal of white space, after all glyphs on the line with priority `kJUSTKashidaPriority` have been extended or compressed to the maximum amount permitted.

`kJUSTLetterPriority`
: Assigned to all glyphs that do not have `kJUSTKashidaPriority` or `kJUSTSpacePriority`. Glyphs with this priority are extended or compressed, typically by the addition or removal of white space, after all glyphs on the line with priority `kJUSTSpacePriority` have been extended or compressed to the maximum amount permitted.

`kJUSTNullPriority`	Available as a priority for glyphs that you want to participate in justification last of all.
`kJUSTPriorityCount`	The number of defined justification priorities. You can use this value for range-checking, size allocation, or loop control.

Line Alignment Constants

You use these constants to specify the alignment of text relative to the text margins within a single line or an entire text layout object. You can set the alignment text layout attribute for a line or an entire text layout object by passing the `kATSULineFlushFactorTag` tag to the functions `ATSUSetLineControls` (page 100) and `ATSUSetLayoutControls` (page 92), respectively.

```
enum {
    kATSUStartAlignment     = 0,
    kATSUEndAlignment       = fract1,
    kATSUCenterAlignment    = fract1 / 2
};
```

Constant descriptions

`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.

VERSION NOTES

Available with ATSUI 1.0.

Line Height Constant

You use this constant to specify that ATSUI use the natural line ascent and descent values dictated by the font and pixel size to determine line ascent and descent. 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 100) and `ATSUSetLayoutControls` (page 92), 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 100) and `ATSUSetLayoutControls` (page 92), respectively.

```
enum {
    kATSUseLineHeight= 0x7FFFFFFF,
};
```

Constant description

`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.

VERSION NOTES

Available with ATSUI 1.0.

Line Justification Constants

You use these constants to specify the degree of line justification for a single line or an entire text layout object. 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 100) and `ATSUSetLayoutControls` (page 92), respectively.

```
enum {
    kATSUNoJustification    = 0x00000000L,
    kATSUFullJustification  = 0x40000000L
};
```

Constant descriptions

`kATSUNoJustification`
Indicates no justification.

`kATSUFullJustification`
Full justification between the text margins. White space is "stretched" to make the line extend to both text margins.

VERSION NOTES

Available with ATSUI 1.0.

Line Layout Option Mask Constants

Your application should use one or more of these mask constants to to set or obtain the various line layout options in a line or text layout object. To set the line layout options text layout attribute in a line or text layout object, pass this 32-bit flag and the `kATSULineLayoutOptionsTag` tag to the functions `ATSUSetLineControls` (page 100) and `ATSUSetLayoutControls` (page 92), respectively.

```
enum {
    kATSLineNoLayoutOptions             = 0x00000000,
    kATSLineIsDisplayOnly               = 0x00000001,
    kATSLineHasNoHangers                = 0x00000002,
    kATSLineHasNoOpticalAlignment       = 0x00000004,
    kATSLineKeepSpacesOutOfMargin       = 0x00000008,
    kATSLineNoSpecialJustification      = 0x00000010,
    kATSLineLastNoJustification         = 0x00000020,
    kATSLineFractDisable                = 0x00000040,
    kATSLineImposeNoAngleForEnds        = 0x00000080,
    kATSLineFillOutToWidth              = 0x00000100,
    kATSLineAppleReserved               = (long)0xFFFFFE00
};
typedef UInt32      ATSLineLayoutOptions
```

Constant descriptions

`kATSLineNoLayoutOptions`
Indicates that no bits are set.

`kATSLineIsDisplayOnly`

If the bit specified by this mask is set, ATSUI creates the text layout object without the internal information needed for editing the text layout; it is for display purposes only. This allows ATSUI to display the text layout faster and make the text layout object smaller. When the user edits the text layout object, you must clear this flag.

`kATSLineHasNoHangers`

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`.

`kATSLineHasNoOpticalAlignment`

If the bit specified by this mask is set, the optical alignment of characters at the text margin of the text layout object will not occur. 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`.

`kATSLineKeepSpacesOutOfMargin`

If the bit specified by this mask is set, the trailing white spaces at the end of a line of justified text are placed outside the margin.

`kATSLineNoSpecialJustification`

If the bit specified by this mask is set, postcompensation actions will not be taken, even if 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`.

`kATSLineLastNoJustification`

If this flag is set, the last line of a justified text layout object will not be justified. This flag is meaningless when setting a line's text layout attributes.

`kATSLineFractDisable`
: If the bit specified by this mask is set, the position of the text in the line or text layout object will be 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 on the screen. This origin is also used to get the width of the typographic bounding rectangle when you call `ATSUMeasureText` (page 148).

`kATSLineImposeNoAngleForEnds`
: If the bit specified by this mask is set, carets on the far right and left sides of an unrotated line will always be vertical, no matter what the angle of text.

`kATSLineFillOutToWidth`
: If the bit specified by this mask is set, highlighting extends to either ends of the line, regardless of caret locations. It does not change caret locations. You must set this bit to ensure that highlighting is done correctly, particularly across tab stops. In this case, you should also set the bit specified by the `kATSLineImposeNoAngleForEnds` mask constant.

`kATSLineAppleReserved`
: If the bit specified by this mask is set, line layout mask values (and the bits they specify) between `kATSLineNoLayoutOptions` and `kATSLineAppleReserved` are reserved. In this case, ATSUI will return the `kATSUInvalidAttributeValueErr` result code if you set a reserved bit.

VERSION NOTES

The mask constants `kATSLineFractDisable`, `kATSLineImposeNoAngleForEnds`, `kATSLineFillOutToWidth`, and `kATSLineAppleReserved` are available with ATSUI 1.1. All other mask constants are available with ATSUI 1.0.

Line Layout Width Constant

Your application can pass this constant in the `iLineWidth` parameter of the function `ATSUBreakLine` (page 156) 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.

```
enum {
    kATSUUseLineControlWidth        = 0x7FFFFFFFL
};
```

Constant description

`kATSUUseLineControlWidth`
Indicates `ATSUBreakLine` should use the previously set line width attribute for the current line to determine how many characters can fit on the line. If none has been set, this value will be ignored.

VERSION NOTES

Available with ATSUI 1.0.

Miscellaneous Constants

The following constants are provided for convenience.

```
enum {
    kATSItalicQDSkew                = (1 << 16) / 4,
    kATSRadiansFactor               = 1144.
    kATSUseLineHeight               = 0x7FFFFFFF,
    kATSNoTracking                  = (long)0x80000000,
};
```

Constant descriptions

`kATSItalicQDSkew` A `Fixed` value of `0.25`.

`kATSRadiansFactor` A `Fixed` value of approximately `pi/180 (0.0174560546875)`.

`kATSUseLineHeight` A value that represents the natural ascent or descent of a line.

`kATSNoTracking` A value of type `negativeInfinity` that indicates that font tracking should be off.

VERSION NOTES

Available with ATSUI 1.0.

No Font Name Platform, Language, or Script Constants

This enumeration defines constants your application can use to indicate that you don't care about the platform, language, or script of a font name string. You can use one of these constants with the functions `ATSUFindFontName` (page 56) and `ATSUFindFontFromName` (page 50) if you want the first name string in the name table that matches a given name code and other parameters that you specify. For example, if you passed the constants `kFontMacintoshPlatform`, `kFontRomanScript`, and `kFontNoLanguage`, you would get the name string of the first MacRoman font name in the font name table.

Note that if you pass `kFontNoPlatform`, you should also pass `kFontNoScript`, since the script code is meaningless without a platform.

```
enum {
    kFontNoPlatform = -1,
    kFontNoScript   = -1,
    kFontNoLanguage = -1
};
```

Constant descriptions

`kFontNoPlatform` Indicates that you don't care about the encoding of the font name string. In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first name string in the name table that matches the name code and other parameters that you specified.

`kFontNoScript` Indicates that you don't care about the platform version (or script ID, if the platform is Macintosh) of the font name string. In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first name string in the name table that matches the name code and other parameters that you specified.

`kFontNoLanguage` Indicates that you don't care about the language of the font name string. In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first name string in the name table that matches the name code and other parameters that you specified.

VERSION NOTES

Available with ATSUI 1.0.

Style Comparison Constants

The function `ATSUCompareStyles` (page 18) passes back a constant of type `ATSUStyleComparison` to indicate how the contents of two style objects compare.

```
enum {
    kATUStyleUnequal            = 0,
    ATSUStyleContains           = 1,
    kATSUStyleEquals            = 2,
    kATSUStyleContainedBy       = 3
};
typedef Uint16      ATSUStyleComparison;
```

Constant descriptions

`kATUStyleUnequal` Indicates that the contents of the second style object are not equivalent to, contained by, or containing those of the first.

`ATSUStyleContains` Indicates that the contents of the second style object are contained by those of the first (excluding pointers and handles to reference constants and custom style run attribute tags).

`kATSUStyleEquals` Indicates that the contents of the second style object are equivalent to those of the first (excluding pointers and handles to reference constants and custom style run attribute tags).

`kATSUStyleContainedBy` Indicates that the contents of the second style object are contained by those of the first (excluding pointers and handles to reference constants and custom style run attribute tags).

VERSION NOTES

Available with ATSUI 1.0.

Style Run Attribute Tags

Your application uses tags of this type with ATSUI functions that manipulate and obtain attribue values that control layout in a style run: to `ATSUSetAttributes` (page 29) to set these values, to `ATSUGetAttribute` (page 31) and `ATSUGetAllAttributes` (page 33) to obtain these values, and to `ATSUClearAttributes` (page 34) to remove these values.

Examples of style attributes include text style, text size, text color, font, kerning, tracking, with- and cross-stream shifting, optical alignment, hanging glyph behavior, and control of postcompensation actions.

If you do not set a style run attribute value, it will be set to the default value listed in Table 2-1. Table 2-1 presents the Apple-defined style run attribute tags and the size, data type, and default value of the attributes they identify.

Table 2-1 Apple-defined style run attribute tags and the size, data type, and default value of the attributes they identify

Style run attribute tag	Data type and size (in bytes) of corresponding attribute		Default value of attribute
`kATSUQDBoldfaceTag`	`Boolean`	1	`false`; plain text
`kATSUQDItalicTag`	`Boolean`	1	`false`; plain text
`kATSUQDUnderlineTag`	`Boolean`	1	`false`; plain text
`kATSUQDCondensedTag`	`Boolean`	1	`false`; plain text
`kATSUQDExtendedTag`	`Boolean`	1	`false`; plain text

Table 2-1 Apple-defined style run attribute tags and the size, data type, and default value of the attributes they identify

Style run attribute tag	Data type and size (in bytes) of corresponding attribute		Default value of attribute
`kATSUFontTag`	`ATSUFontID` (page 192)	4	the application font for the current script system; you can determine this value by calling `GetScriptVariable(smSystemScript,smScriptAppFond)`. If the application font cannot be rendered with ATSUI, the default font is Helvetica.
`kATSUSizeTag`	`Fixed`	4	the size of the application font for the current script system; you can determine this value by calling `GetScriptVariable(smSystemScript,smScriptAppFondSize)` and examining the low word of the return value
`kATSUColorTag`	`RGBColor`	6	`(0, 0, 0)`; black text
`kATSULanguageTag`	`RegionCode`	2	the `RegionCode` of the system script; you can determine this value by calling `GetScriptVariable(smSystemScript,smScriptLang)`

Table 2-1 Apple-defined style run attribute tags and the size, data type, and default value of the attributes they identify

Style run attribute tag	Data type and size (in bytes) of corresponding attribute		Default value of attribute
`kATSUVerticalCharacterTag`	`ATSUVerticalCharacterType`	2	`kATSUStronglyHorizontal`; horizontally-oriented glyphs
`kATSUImposeWidthTag`	`ATSUTextMeasurement` (page 196)	4	`kATSUNoImposedWidth`;; use font-defined character width default value
`kATSUBeforeWithStreamShiftTag`	`Fixed`	4	0; use the font-defined with-stream shift default value before glyphs
`kATSUAfterWithStreamShiftTag`	`Fixed`	4	0; use the font-defined with-stream shift default value after glyphs
`kATSUCrossStreamShiftTag`	`Fixed`	4	0; use the font-defined cross-stream shift default value
`kATSUTrackingTag`	`Fixed`	4	0; use the font-defined tracking default value
`kATSUHangingInhibitFactorTag`	`Fract`	4	0; use the font-defined hanging glyph default value
`kATSUKerningInhibitFactorTag`	`Fract`	4	0; use the font-defined default kerning value
`kATSUDecompositionFactorTag`	`Fixed`	4	0; use the font-defined ligature decomposition default value
`kATSUBaselineClassTag`	`ATSUFontID` (page 192)	4	`kBSLNRomanBaseline`; Roman baseline

Table 2-1 Apple-defined style run attribute tags and the size, data type, and default value of the attributes they identify

Style run attribute tag	Data type and size (in bytes) of corresponding attribute		Default value of attribute
`kATSUPriorityJustOverrideTag`	`ATSJustWidthDeltaEntryOverride` structure	20	0's in all fields; apply the font-defined justification priority behavior default values
`kATSUNoLigatureSplitTag`	`Boolean`	1	`false`; treat ligatures as divisible
`kATSUNoCaretAngleTag`	`Boolean`	1	`false`; use inherent angle of text to draw caret and highlighting
`kATSUSuppressCrossKerningTag`	`Boolean`	1	`false`; use the font-defined cross kerning default value
`kATSUNoOpticalAlignmentTag`	`Boolean`	1	`false`; use the font-defined optical alignment default value
`kATSUForceHangingTag`	`Boolean`	1	`false`; glyphs will not extend into the margin, even if they would normally do so

Table 2-1 Apple-defined style run attribute tags and the size, data type, and default value of the attributes they identify

Style run attribute tag	Data type and size (in bytes) of corresponding attribute		Default value of attribute
`kATSUNoSpecialJustificationTag`	`Boolean`	1	`false`; postcompensation actions will occur if they are needed
`kATSUMaxStyleTag`			the maximum Apple-defined style run attribute tag value
`kATSUMaxATSUITagValue`			No Apple-defined tags may exceed this value. Apple-defined values between `kATSUMaxStyleTag` and `kATSUMaxATSUITagValue` are reserved. You can create you own attribute tags with any greater value.

IMPORTANT

The following descriptions assume horizontal text. If you are setting or getting the style run attributes of vertical text, you should interpret these accordingly. ▲

```
enum {
    kATSUQDBoldfaceTag                  = 256L,
    kATSUQDItalicTag                    = 257L,
    kATSUQDUnderlineTag                 = 258L,
    kATSUQDCondensedTag                 = 259L,
    kATSUQDExtendedTag                  = 260L,
    kATSUFontTag                        = 261L,
    kATSUSizeTag                        = 262L,
    kATSUColorTag                       = 263L,
    kATSULanguageTag                    = 264L,
    kATSUVerticalCharacterTag           = 265L,
```

```
    kATSUImposeWidthTag                 = 266L,
    kATSUBeforeWithStreamShiftTag       = 267L,
    kATSUAfterWithStreamShiftTag        = 268L,
    kATSUCrossStreamShiftTag            = 269L,
    kATSUTrackingTag                    = 270L,
    kATSUHangingInhibitFactorTag        = 271L,
    kATSUKerningInhibitFactorTag        = 272L,
    kATSUDecompositionInhibitFactorTag  = 273L,
    kATSUBaselineClassTag               = 274L,
    kATSUPriorityJustOverrideTag        = 275L,
    kATSUNoLigatureSplitTag             = 276L,
    kATSUNoCaretAngleTag                = 277L,
    kATSUSuppressCrossKerningTag        = 278L,
    kATSUNoOpticalAlignmentTag          = 279L,
    kATSUForceHangingTag                = 280L,
    kATSUNoSpecialJustificationTag      = 281L,
    kATSUMaxStyleTag                    = 282L,
    kATSUMaxATSUITagValue               = 65535L
};
typedef UInt32              ATSUAttributeTag;
```

Constant descriptions

`kATSUQDBoldfaceTag` You use this tag to set or get a value of type `Boolean` that indicates whether the text style of the glyphs in a style run is boldfaced. If `true`, text is boldfaced; if `false`, text is plain style. Making text boldfaced causes each glyph to be repeatedly drawn one bit to the right for extra thickness.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, text in the style object will be plain style.

`kATSUQDItalicTag` You use this tag to set or get a value of type `Boolean` that indicates whether the text style of the glyphs in a style run is italicized. If `true`, text is italicized; if `false`, text is plain style. Making text italicized skews glyph bits above the baseline to the right, bits below the baseline to the left.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, text in the style object will be plain style.

`kATSUQDUnderlineTag`

You use this tag to set or get a value of type `Boolean` that

indicates whether the text style of the glyphs in a style run is underlined. If `true`, text is underlined; if `false`, text is plain style. Making text underlined draws the underline through the entire text line, from the pen starting position through the ending position, plus any offsets from the font or italic kerning. If part of a glyph descends below the base line, generally, the underline isn't drawn through the pixel on either side of the descending part.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, text in the style object will be plain style.

`kATSUQDCondensedTag`

You use this tag to set or get a value of type `Boolean` that indicates whether the text style of the glyphs in a style run is condensed. If `true`, text is condensed; if `false`, text is plain style. Making text condensed decreases the horizontal distance between all glyphs, including spaces, by the amount that the Font Manager determines is appropriate.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, text in the style object will be plain style.

`kATSUQDExtendedTag` You use this tag to set or get a value of type `Boolean` that indicates whether the text style of the glyphs in a style run is extended. If `true`, text is extended; if `false`, text is plain style. Making text extended increases the horizontal distance between all glyphs, including spaces, by the amount that the Font Manager determines is appropriate.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, text in the style object will be plain style.

`kATSUFontTag` You use this tag to set or get a value of type `ATSUFontID` (page 192) that uniquely identifies the font in the style run to the font management system in ATSUI.

If you do not set the attribute value corresponding to this tag, the default value is the ID corresponding to the font family number of the application font for the current script system. To determine this value, evaluate the result of the call `GetScriptVariable (smSystemScript,smScriptAppFond)`.

If the application font does not have a corresponding ID, the default value is Helvetica.

`kATSUSizeTag` You use this tag to set or get a value of type `Fixed` that represents the size, in typographic points (72 per inch), of the text in the style run.

If you do not set the attribute value corresponding to this tag, the default value is the application font size for the current script system. To determine this value, evaluate the low word result of the call `GetScriptVariable(smSystemScript,smScriptAppFondSize)`.

`kATSUColorTag` You use this tag to set or get a value of type `RGBColor` that represents the color of the text in the style run.

If you do not set the attribute value corresponding to this tag, the default value is `(0, 0, 0)`. In this case, the color of text will be black.

`kATSULanguageTag` You use this tag to set or get a value of type `RegionCode` that represents the regional language and other region-dependent characteristics for glyphs in the style run.

If you do not set the attribute value corresponding to this tag, the default value is the region code of the system script. To determine this value, evaluate the result of `GetScriptVariable (smSystemScript,smScriptLang)`.

`kATSUVerticalCharacterTag`

You use this tag to set or get a value of type `ATSUVerticalCharacterType` that represents glyph orientation in the style run.

See "Glyph Orientation Constants" (page 223) for a description of possible values. To produce vertical text, you must set the corresponding value to the `kATSUStronglyVertical` constant and the text layout attribute value corresponding to the `kATSULineRotationTag` tag to -90 degrees.

If you do not set the attribute value corresponding to this tag, the default value is `kATSUStronglyHorizontal`. In this case, glyph orientation will be horizontal.

`kATSUImposeWidthTag`

You use this tag to set or get a value of type `ATSUTextMeasurement` (page 196) that represents the

imposed width on glyph in the style run. This width is imposed regardless of the value of other style run attributes. Note that ATSUI ignores negative values.

If you do not set the attribute value corresponding to this tag, the default value is `kATSUNoImposedWidth`. In this case, ATSUI applies the default imposed width value defined for the font.

`kATSUBeforeWithStreamShiftTag`

You use this tag to set or get a value of type `Fixed` that represents the with-stream shift to apply equally before all glyphs in the style run. Positive values increase space before each glyph, while negative values decrease space before each glyph.

If you do not set the attribute value corresponding to this tag, the default value is `0`. In this case, ATSUI applies the default with-stream shift value defined for the font to the space before each glyph.

`kATSUAfterWithStreamShiftTag`

You use this tag to set or get a value of type `Fixed` that represents the with-stream shift to apply equally after all glyphs in the style run. Positive values increase space after each glyph, while negative values decrease space after each glyph.

If you do not set the attribute value corresponding to this tag, the default value is `0`. In this case, ATSUI applies the default with-stream shift value defined for the font to the space after each glyph.

`kATSUCrossStreamShiftTag`

You use this tag to set or get a value of type `Fixed` that represents the cross-stream shift to apply to all glyphs in the style run. This positional shift raises or lowers each glyph in the style run. Positive values shift glyphs upward, while negative values shift glyphs downwards.

If you do not set the attribute value corresponding to this tag, the default value is `0`. In this case, ATSUI uses the default cross-stream shift value defined for the font.

`kATSUTrackingTag` You use this tag to set or get a value of type `Fixed` that represents the tracking to apply to all glyphs in the style run. Tracking is kerning between all glyphs in the style run,

not just the kerning pairs already defined by the font. Positive values loosen tracking, while negative numbers tighten tracking. If you do not want tracking to occur, specify the constant `kATSUNoTracking`.

If you do not set the attribute value corresponding to this tag, the default value is `kATSUNoTracking`. In this case, ATSUI uses the default tracking value defined for the font.

`kATSUHangingInhibitFactorTag`

You use this tag to set or get a value of type `Fract` that represents the amount to inhibit font-defined hanging of glyphs that typically extend beyond the text margins and are not counted when line length is measured. Values can range between `0` and `1`. A value of `0` means you want to use font-defined glyph hanging, while a value of `1` indicates no hanging (that is, total inhibition of font-defined hanging glyphs).

If you do not set the attribute value corresponding to this tag, the default value is `0`. In this case, ATSUI uses the default hanging glyph value defined for the font.

`kATSUKerningInhibitFactorTag`

You use this tag to set or get a value of type `Fract` that represents the amount to inhibit font-defined kerning. Kerning is an adjustment to the normal spacing that occurs between two or more specifically-named glyphs, also known as a kerning pair.

A value of `0` means you want to use font-defined kerning, while a value of `1` indicates no kerning (that is, total inhibition of font-defined kerning).

If you do not set the attribute value corresponding to this tag, the default value is `0`. In this case, ATSUI uses the default kerning value defined for the font.

`kATSUDecompositionInhibitFactorTag`

You use this tag to set or get a value of type `Fixed` that represents the amount to inhibit font-defined ligature decomposition during justification. A ligature is two or more glyphs connected to form a single new glyph. Ligature decomposition is the replacement of ligatures with the glyphs for their component characters during justification.

Values can range from -1.0 to 1.0. Positive values increase the font-defined threshold, while negative values lessen it. For example, a value of 0.5 adds 50 percent to the font-specified threshold, while a value of -0.25 subtracts 25 percent from that threshold.

If you do not set the attribute value corresponding to this tag, the default value is 0. In this case, ATSUI uses the default ligature decomposition value defined for the font.

kATSUBaselineClassTag

You use this tag to set or get a value of type BslnBaselineClass that represents the baseline that you want ATSUCalculateBaselineDeltas to use in calculating the distances to each of the other baseline types in the style run.

See "Baseline Type Constants" (page 204) for a description of possible values. Values can range from 0 (kBSLNRomanBaseline) to 31 (kBSLNLastBaseline). Typically, the values 0 through 4 are used; the remaining are application-defined. If you want to use the standard baseline value defined by the font, set the attribute value to kBSLNLastBaseline.

If you do not set the attribute value corresponding to this tag, the default value is kBSLNRomanBaseline. In this case, ATSUI uses the Roman baseline for the style run.

kATSUPriorityJustOverrideTag

You use this tag to set or get an array of type ATSJustPriorityWidthDeltaOverrides (page 183) that represents the overriding behavior to use in justifying glyphs in a style run. The array contains four structures of type ATSJustPriorityWidthDeltaOverrides, one for each justification priority.

If you do not set the attribute value corresponding to this tag, the default value is 0's in the fields of all four structures. In this case, ATSUI applies the default justification prioritity behavior defined for the font.

kATSUNoLigatureSplitTag

You use this tag to set or get an array of type Boolean that indicates whether ligatures should be treated as their

component glyphs or as an indivisible unit for the purpose of caret positioning.

A value of `true` indicates that ligatures will not be split into their component glyphs. In this case, when the caret position is adjacent to one, ATSUI considers the next valid caret position to be across the entire ligature rather than at any point within it.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, ATSUI treats the ligature as divisible (unless the characters are a surrogate pair or a pre-coded Unicode ligature).

`kATSUNoCaretAngleTag`

You use this tag to set or get an array of type `Boolean` that indicates whether the angle of the caret and the highlight region should be parallel to the inherent angle of the text or perpendicular to the baseline.

A value of `true` indicates that the caret and highlight angles will be perpendicular to the baseline.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, the caret and highlight angles will reflect the inherent angle of the text.

`kATSUSuppressCrossKerningTag`

You use this tag to set or get an array of type `Boolean` that indicates whether to inhibit font-defined cross-stream kerning. Setting this value has no impact on manual cross-stream kerning.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, ATSUI applies font-defined cross-stream kerning to the glyphs in the style run.

`kATSUNoOpticalAlignmentTag`

You use this tag to set or get an array of type `Boolean` that indicates whether to inhibit font-specified optical alignment. Optical alignment is the automatical adjustment of glyph position at the ends of lines to give a more even visual appearance to margins.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, ATSUI applies

font-defined optical alignment to the glyphs in the style run.

`kATSUForceHangingTag`

You use this tag to set or get an array of type `Boolean` that indicates whether glyphs should extend into the margins.

If you set the attribute value identified by this tag to `false`, glyphs will not extend into the text margins, even if they are hanging glyphs that would normally do so as defined by the font or you (if you set the attribute identified by the `kATSUHangingInhibitFactorTag`).

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, glyphs will not extend into the margin.

`kATSUNoSpecialJustificationTag`

You use this tag to set or get an array of type `Boolean` that indicates whether postcompensation actions should occur occur after glyph positions have been calculated.

If you set the attribute value identified by this tag to `true`, postcompensation actions will not be occur, even if they are needed.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, postcompensation actions will occur if they are needed.

`kATSUMaxStyleTag` Represents the maximum Apple-defined style run attribute tag value.

`kATSUMaxATSUITagValue`

No Apple-defined tags may exceed this value. Apple-defined values between `kATSUMaxStyleTag` and `kATSUMaxATSUITagValue` are reserved. You can create you own attribute tags with any greater value.

VERSION NOTES

The tag constants `kATSUMaxStyleTag` and `kATSUMaxATSUITagValue` are available with ATSUI 1.1. All other tag constants are available with ATSUI 1.0.

Text Layout and Line Attribute Tags

Your application uses tags of this type with ATSUI functions that manipulate and obtain the attribute values that control line layout in a text layout object: to `ATSUSetLayoutControls` (page 92) to set these values, to `ATSUGetLayoutControl` (page 94) and `ATSUGetAllLayoutControls` (page 95) to obtain these values, and to `ATSUClearLayoutControls` (page 97) to remove these values.

You can also use these tags with the function `ATSUSetLineControls` (page 100) to override attributes that were previously set in a text layout object in a single line of the text layout object. When you set a line attribute value, it overrides the corresponding attribute value in the text layout attribute containing the line. This is true regardless of the order in which you set these values. You can also use these tags with the functions `ATSUGetLineControl` (page 102), `ATSUGetAllLineControls` (page 104), and `ATSUClearLineControls` (page 105) to obtain and remove attribute values from a line in a text layout object.

Examples of text layout and line attributes include line width, line rotation, line direction, line justification, line alignment, line ascent, line descent, baseline offsets, and special layout options.

If you do not set a text layout or line value, it will be set to the default value listed in Table 2-2. Table 2-2 presents the Apple-defined text layout and line attribute tags and the size, data type, and default values of the attributes they identify.

Table 2-2 Apple-defined text layout and line attribute tags and the size, data type, and default value of the attributes they identify

Text layout and line attribute tag	Data type and size (in bytes) of corresponding attribute		Default value of attribute
kATSULineWidthTag	ATSUTextMeasurement (page 196)	4	0; no imposed line width
kATSULineRotationTag	Fixed	4	0; no line rotation
kATSULineDirectionTag	Boolean	1	derived from the system script; you can determine this value by calling the function GetSysDirection
kATSULineJustificationFactorTag	Fract	4	kATSUNoJustification; no line justification
kATSULineFlushFactorTag	Fract	4	kATSUStartAlignment; text is drawn to the right of the left margin
kATSULineBaselineValuesTag	BslnBaselineRecord (page 196)	4	all 0's; no baseline deltas are applied to the cross-stream shifting of glyphs

Text layout and line attribute tag	Data type and size (in bytes) of corresponding attribute		Default value of attribute
kATSULineLayoutOptionsTag	UInt32	4	kATSUNoLayoutOptions; no special line layout options are set
kATSULineAscentTag	ATSUTextMeasurement (page 196)	4	kATSUseLineHeight; use the maximum line ascent of all the style runs in a line
kATSULineDescentTag	ATSUTextMeasurement (page 196)	4	kATSUseLineHeight; use the maximum line descent of all the style runs in a line

IMPORTANT

The following descriptions assume horizontal text. If you are setting or getting the style run attributes of vertical text, you should interpret these accordingly. ▲

```
enum {
    kATSULineWidthTag                   = 1L,
    kATSULineRotationTag                = 2L,
    kATSULineDirectionTag               = 3L,
    kATSULineJustificationFactorTag     = 4L,
    kATSULineFlushFactorTag             = 5L,
    kATSULineBaselineValuesTag          = 6L,
    kATSULineLayoutOptionsTag           = 7L,
    kATSULineAscentTag                  = 8L,
    kATSULineDescentTag                 = 9L
};
typedef UInt32              ATSUAttributeTag;
```

Constant descriptions

kATSULineWidthTag — You use this tag to set or get a value of type ATSUTextMeasurement (page 196) that represents the line width to impose on a single line or on each line of a text

layout object. Note that if you set this value to 0, ATSUI will act as if you have not set the line width.

If you do not set the attribute value corresponding to this tag, the default value is `0`. In this case, ATSUI does not impose a line width.

`kATSULineRotationTag`

You use this tag to set or get a value of type `Fixed` value that represents the angle of line rotation (in units of degrees) for a single line or for each line in a text layout object. Values can range from `-1.0` to `1` . Negative values rotate the line clockwise. Positive values rotate the line counter-clockwise.

If you do not set the attribute value corresponding to this tag, the default value is `0`. In this case, ATSUI does not impose line rotation.

`kATSULineDirectionTag`

You use this tag to set or get a value of type `Boolean` value that indicates the direction of text in each line of a text layout object. See "Glyph Direction Constants" (page 223) for a description of possible values. You cannot set line direction for a single line.

If you do not set the attribute value corresponding to this tag, the default value is `false`. In this case, text direction is derived from the system script, which you can determine by calling the `GetSysDirection` function.

`kATSULineJustificationFactorTag`

You use this tag to set or get a value of type `Fract` value that represents the justification to impose on a single line or on each line of a text layout object. See "Line Justification Constants" (page 230) for a description of possible values. Values can range from `0` to `1`. A value of `0` represents no justification. A value of `1` represents full justification.

If you do not set the attribute value corresponding to this tag, the default value is `kATSUNoJustification`. In this case, no justification is imposed.

`kATSULineFlushFactorTag`

You use this tag to set or get a value of type `Fract` value that represents the alignment of text in a single line or in each line of a text layout object. See "Line Alignment

Constants" (page 229) for a description of possible values. Intermediate values position the text proportional distances from the left and right margins. If you specify the constants `kATSUEndAlignment` or `kATSUCenterAlignment`, you must also set the line width of the corresponding line(s) of the text layout object.

If you set text alignment, you must also set line justification for the corresponding line(s).

If you do not set the attribute value corresponding to this tag, the default value is `kATSUStartAlignment`. In this case, text is aligned to the right of the left margin.

`kATSULineBaselineValuesTag`

You use this tag to set or get an array of type `BslnBaselineRecord` (page 196) that represents the optimal baseline positions to use in controlling glyph placement in a single line or in each line of a text layout object.

To determine this value, call the function `ATSUCalculateBaselineDeltas` (page 35) and pass the style object of the dominant style run in the line.

If you do not set the attribute value corresponding to this tag, the default value is `0` for each element of the array. In this case, ATSUI applies the font-defined cross-stream shift to each glyph.

`kATSULineLayoutOptionsTag`

You use this tag to set or get a mask value of type `ATSLineLayoutOptions` that controls layout options for a single line or for each line in a text layout object. See "Line Layout Option Mask Constants" (page 231) for a description of possible values.

If you do not set the attribute value corresponding to this tag, the default value is the mask value `kATSUNoLayoutOptions`. In this case, no special layout options are set.

`kATSULineAscentTag` You use this tag to set or get a value of type `ATSUTextMeasurement` (page 196) that represents line ascent. You can specify any nonnegative value to reflect the distance above the line's baseline. Note that if you set this value to 0, ATSUI will act as if you have not set the line ascent.

Line ascent is only taken into consideration by the functions `ATSUHighlightText` (page 165) and `ATSUUnhighlightText` (page 168) in order to calculate the ascent of the highlight region.

If you do not set the attribute value corresponding to this tag, the default value is `kATSUseLineHeight`. In this case, ATSUI uses the calculated line ascent from the maximum ascent along the line of all the style runs.

`kATSULineDescentTag`

You use this tag to set or get a value of type `ATSUTextMeasurement` (page 196) that represents line descent. You can specify any nonnegative value to reflect the distance below the line's baseline. Note that if you set this value to 0, ATSUI will act as if you have not set the line descent.

Line descent is only taken into consideration by the functions `ATSUHighlightText` (page 165) and `ATSUUnhighlightText` (page 168) in order to calculate the descent of the highlight region.

If you do not set the attribute value corresponding to this tag, the default value is `kATSUseLineHeight`. In this case, ATSUI uses the calculated line descent from the maximum descent along the line of all the style runs.

VERSION NOTES

The tag constants `kATSULineAscentTag` and `kATSULineDescentTag` are available with ATSUI 1.1. All other tag constants are available with ATSUI 1.0.

Text Length Constant

Your application can pass a `kATSUToTextEnd` constant to ATSUI functions that operate on a range of text in a text layout object to represent the end of a text layout object's text buffer. If you want to specify the entire text text buffer, pass this constant in conjunction with the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 256).

```
enum {
    kATSUToTextEnd      = (long)0xFFFFFFFF
};
```

Constant descriptions

`kATSUToTextEnd` Represents the end of the text layout object's text buffer.

VERSION NOTES

Available with ATSUI 1.0.

Text Offset Constant

Your application can pass a `kATSUFromTextBeginning` constant to ATSUI functions that operate on a range of text in a text layout object to represent the edge offset of the beginning of a text layout object's text buffer. If you want to specify the entire text text buffer, pass this constant in conjunction with the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 255).

```
enum {
    kATSUFromTextBeginning      = (long)0xFFFFFFFF,
};
```

Constant descriptions

`kATSUFromTextBeginning`
Represents the edge offset of the beginning of the text layout object's text buffer.

VERSION NOTES

Available with ATSUI 1.0.

Result Codes

All ATSUI functions return result codes of type `OSStatus`. This includes general result codes such as `noErr` (function completed successfully) and `paramErr` (one or more of the input parameters has an invalid value). In addition, ATSUI

functions that allocate memory may return `memFullErr` if there is not enough memory in the designatedheap.

The result codes specific to ATSUI are listed below in Table 2-3 (page 258). In some cases, the function result section for a particular function provides more detail about the meaning of the result code specific to that function.

Table 2-3 ATSUI-specific result codes

Result code constant	Value	Description
kATSUInvalidTextLayoutErr	-8790	Text layout object not previously initialized or in an otherwise invalid state
kATSUInvalidStyleErr	-8791	Style object not previously initialized or in an otherwise invalid state
kATSUInvalidTextRangeErr	-8792	Text range extends beyond the limits of the text layout object's text range
kATSUFontsMatched	-8793	Character could not be rendered with its assigned font
kATSUFontsNotMatched	-8794	Character could not be rendered with its assigned font or any currently active font
kATSUNoCorrespondingFontErr	-8795	Font ID corresponds to an existing font that isn't available to ATSUI
kATSUInvalidFontErr	-8796	Font ID does not correspond to any installed font
kATSUInvalidAttributeValueErr	-8797	Invalid or undefined attribute value
kATSUInvalidAttributeSizeErr	-8798	Allocated attribute value size is less than required
kATSUInvalidAttributeTagErr	-8799	ATSUI-reserved tag value or wrong type of attribute tag (that is, style run attribute tag instead of text layout attribute tag and vice versa)
kATSUInvalidCacheErr	-8800	Attempt to read in 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)
kATSUNotSetErr	-8801	Style object's attribute, font feature, font variation not set; text layout object or single line's attribute not set; or font name not set
kATSUNoStyleRunsAssignedErr	-8802	No style runs assigned to text layout object

kATSUQuickDrawTextErr	-8803	QuickDraw function DrawText encountered an error rendering or measuring a line of text
kATSULowLevelErr	-8804	Error encountered in Apple Type Solution (ATS) while performing an operation requested by ATSUI
kATSUNoFontCmapAvailableErr	-8805	'CMAP' table cannot be accessed or synthesized for a font set by the function ATSUSetAttributes (page 29)
kATSUNoFontScalerAvailableErr	-8806	No font scaler available for a font set by the function ATSUSetAttributes (page 29)
kATSUCoordinateOverflowErr	-8807	Input coordinates caused coordinate overflow
kATSULineBreakInWord	-8808	Returned by ATSUBreakLine (page 156) to indicate that ATSUBreakLine performed a line break within a word.
kATSULastErr	-8809	No ATSUI-specific result code may exceed this value. Result code values between kATSUInvalidTextLayoutErr and kATSULastErr are reserved.

VERSION NOTES

The result code constants kATSUNoStyleRunsAssignedErr, kATSUQuickDrawTextErr, kATSULowLevelErr, kATSUNoFontCmapAvailableErr, kATSUNoFontScalerAvailableErr, and kATSUCoordinateOverflowErr are available with ATSUI 1.1. All other result code constants are available with ATSUI 1.0.

APPENDIX A

Document Revision History

This document has had the following releases:

Table A-1 *Apple Type Services for Unicode Imaging Reference* revision history

Version	Notes
May 7, 1999	Updated document to cover ATSUI 1.1. For a listing of all new 1.1 functions, as well as any ATSUI 1.0 functions whose implementation has changed with ATSUI 1.1, see "Functions New to ATSUI 1.1 and Changed From ATSUI 1.0" (page 263).
	Updated, expanded, and corrected descriptions of ATSUI 1.0 API.
Mar. 12, 1999	Initial public release of *Apple Type Services for Unicode Imaging Reference* covering ATSUI 1.0.
Sep. 23, 1998	First seed draft release of ATSUI 1.0 API documentation. Document title: *Rendering Apple Type Services for Unicode Imaging (ATSUI)*.

Functions New to ATSUI 1.1 and Changed From ATSUI 1.0

Table B-1 alphabetically lists all ATSUI 1.1 functions.

Table B-1 Functions new to ATSUI 1.1

Function name
`ATSUClearLayoutCache` (page 88)
`ATSUClearLineControls` (page 105)
`ATSUCopyLineControls` (page 98)
`ATSUCountFontTracking` (page 61)
`ATSUCreateAndCopyTextLayout` (page 85)
`ATSUCreateMemorySetting` (page 174)
`ATSUDisposeMemorySetting` (page 177)
`ATSUGetAllLineControls` (page 104)
`ATSUGetCurrentMemorySetting` (page 176)
`ATSUGetFontFallbacks` (page 120)
`ATSUGetIndFontTracking` (page 62)
`ATSUGetLineControl` (page 102)
`ATSUSetCurrentMemorySetting` (page 176)
`ATSUSetFontFallbacks` (page 119)
`ATSUSetLineControls` (page 100)
`MyATSUCustomAllocFunc` (page 178)
`MyATSUCustomFreeFunc` (page 181)
`MyATSUCustomGrowFunc` (page 179)

Table B-2 alphabetically lists any ATSUI 1.0 functions whose implementation has changed with ATSUI 1.1.

Table B-2 Functions whose implementaion has changed in ATSUI 1.1

Function name	Changed from 1.0
`ATSUBreakLine` (page 156)	Now returns the result code `kATSULineBreakInWord` to indicate that `ATSUBreakLine` performed a line break within a word. Note that this is a status message, not an error code.
`ATSUCopyToHandle` (page 27)	No longer used. Instead, use the `'ustl'` resource, described in `ustl` (page 199), to format styled text in the clipboard.
`ATSUHighlightText` (page 165)	Now can extend highlighting across tab stops using the mask constants `kATSLineFillOutToWidth` and `kATSLineImposeNoAngleForEnds`, described in "Line Layout Option Mask Constants" (page 231).
`ATSUPasteFromHandle` (page 28)	No longer used. Instead, use the `'ustl'` resource, described in `ustl` (page 199), to format styled text in the clipboard.
`ATSUSetAttributes` (page 29)	New style run attribute tag constants now available. See "Style Run Attribute Tags" (page 237) for details.
`ATSUSetLayoutControls` (page 92)	New text layout and line attribute tag constants now available. See "Text Layout and Line Attribute Tags" (page 250) for details.
`ATSUUnhighlightText` (page 168)	Now can erase highlighting across tab stops using the mask constants `kATSLineFillOutToWidth` and `kATSLineImposeNoAngleForEnds`, described in "Line Layout Option Mask Constants" (page 231).

ATSUI Implementation of the Unicode Specification

This Appendix describes ATSUI's implementation of the Unicode specification. ATSUI provides support for all the text-drawing features required by scripts included with version 2.1 of the Unicode Standard or later. It does not provide other Unicode-related text processing services such as date and time formatting, collation, or string matching. The ability of ATSUI to render Unicode text is only limited by the available fonts the user has installed.

With ATSUI, Unicode text rendering includes support for accents and ligatures, bidirectional text, contextual forms and vowel reordering, vertical text, and surrogates. These capabilities are described and illustrated in subsequent sections of this document.

The **Unicode Standard** is a 2-byte character encoding system designed to support the interchange, processing, and display of all the written texts of the diverse languages of the modern world. It provides a single model for text-related activities, including text display and editing. Unicode simplifies the handling of bidirectional text and characters that change according to their position in the sentence.

Unicode is fully defined in *The Unicode Standard, Version 2.0*, published by Addison-Wesley (1996) and the Unicode 2.1 addendum available from the Unicode Consortium. For more complete information, see the Unicode web site at <http://www.unicode.org/>. You should use Unicode text conforming to the current version of the Unicode Standard.

The correct handling of many Unicode characters requires that the current font supports those characters properly. For example, correct ligature formation requires that the font supports those features using Apple Advanced Typography (AAT) tables. If there is more than one equivalent combining character sequence for a given glyph, the font is responsible for mapping all such sequences to the correct glyph. For example, ATSUI will not automatically support conjoining jamos in a Korean font that specifies pre-composed glyphs only. For more details on the required AAT font tables and tools for creating them, see the description of Apple Advanced Typography at <http://fonts.apple.com>.

Character Size

All characters in Unicode are 16 bits in size. In this, it differs from other character encodings such as those used in East Asia, where some characters are 8-bits in size and others 16-bits.

Moreover, the interpretation of any character in Unicode is not dependent on the character or characters surrounding it. The meaning of a character doesn't change depending on its context (although its visual appearance might). Again, this is not true in general of the character sets used in East Asia. Surrogates, described below, are a partial exception to these rules.

As a result, ATSUI functions that are intended only for Unicode take arrays of UniChars (data type UInt16) as arguments. There is no need to use a void * or char * to hide the size of individual characters.

Control Characters

ATSUI 1.1 does not support the following control characters:

- U+00AD (soft hyphen)
- U+206A (inhibit symmetric swapping)
- U+206B (activate symmetric swapping)
- U+206C (inhibit symmetric swapping)
- U+206C (inhibit symmetric swapping)
- INHIBIT ARABIC FORM SHAPING
- U+206D, ACTIVATE ARABIC FORM SHAPING
- U+206E, NATIONAL DIGIT SHAPES; U+206F
- NORMAL DIGIT SHAPES
- the underline character

You can, however, achieve similar effects achieved by these control characters by setting certain style run attributes, described in "Style Run Attribute Tag

Constants". In addition, ATSUI currently treats the following characters as hard line breaks: U+000A, LINE FEED; U+000C, FORM FEED; U+000D, CARRIAGE RETURN; U+2028, LINE SEPARATOR; and U+2029, PARAGRAPH SEPARATOR.

ATSUI fully renders non-spacing marks, though correct font tables are required to render and process non-spacing marks correctly. To locate text element boundaries, ATSUI defines a cluster as a run consisting of a base character plus zero or more non-base characters, where a base character is defined as one whose combining class is 0 and whose glyph is not deleted. Whether or not a set of characters is a cluster is also dependent upon the behavior of the specific font you are using with those characters. See "Cursor Movement Constants" (page 400) for more information about clusters.

ATSUI fully supports the Unicode bidirectional algorithm, including the bidirectional ordering codes. Correct bidirectional processing requires that the font have the correct glyph properties set (for example, mirrored punctuation). Other characters that require font support for correct processing include invisible characters such as U+FEFF, ZERO WIDTH NO-BREAK SPACE. There are some characters that ATSUI will map to either a zero-width glyph or a non-marking return.

Combining Characters

Unicode has a number of characters defined to be combining characters. Combining characters are two glyphs that can be treated as either one or two text elements, depending upon user preference. For example, in Dutch, the characters "ij" are typically considered a single letter and yet can be viewed as two separate text elements.

Rather than encode all the possible combinations of letter and accent that users might want or limit the user to only those pre-defined and approved letter/accent combinations in the standard, Unicode allows users to create new letter-accent combinations on the fly, as they may need them.

Support for combining characters is necessary for computer implementation of certain scripts, like Arabic or the various south Asian scripts. Combining characters are used to represent indivisible letter-with-accent in various Latin-derived writing systems.

Whether a particular symbol is an accented letter or an indivisible letter-with-accent will vary from culture to culture and from user to user. For example, English-speakers would generally consider "é" to be a letter with an accent on it, whereas French-speakers would generally consider it a letter-with-accent. An English-speaker is more likely to want to delete an "é" with two presses of the delete key (one to remove the accent and the other the base character), and a French-speaker with one.

The interpretation of the combining character does not change. Even if a "´" is a part of "é" in one instance and part of "á" in another, it is always an "´". Your application might need to vary how it handles the character depending on character context and user preferences, but the interpretation of the character does not vary.

The functions `ATSUNextCursorPosition`, `ATSUPreviousCursorPosition`, `ATSURightwardCursorPosition`, and `ATSULeftwardCursorPosition` can treat individual combining characters either as part of an indivisible larger unit (that is, a cluster) or as characters in their own right, depending on the user's preference. You can use the style run attribute tag `kATSUNoLigatureSplitTag` to make combining characters that are due to ligature formation one or two text elements.

Surrogates

To accommodate the encoding of more characters than there were code points in Unicode, surrogates were added to version 2.0 of the Unicode Standard. Currently, there are no characters formally encoded using surrogates, though six scripts have been approved for encoding with surrogates (namely, the Deseret Alphabet, Shavian, Etruscan, Gothic, Linear B, and Cypriot).

In addition, a large number of scripts, including Egyptian hieroglyphics, cuneiform, and rare Han ideographs, are earmarked for encoding with surrogates if they are ever encoded in Unicode. Apple intends to provide full support for surrogates as it becomes required. the Like font features, surrogates cannot be implemented without font support.

Surrogates break some of the assumptions about Unicode characters. In a sense, U+D800 is only half a character and not a whole character, and the exact meaning of the character (and its character properties) depend on what follows it. In text insertion, deletion, selection, hit-testing, and cursor movement, you

should treat surrogate pairs as single entities. In other areas like ligatures or accented letters, you may treat them as a single or multiple entities.

There are two sets of surrogates: U+D800 through U+DBFF are “high surrogates” and D+DC00 through U+DFFF are “low surrogates.” A valid surrogate pair is a high surrogate followed by a low surrogate. Each valid surrogate pair indicates a different character. Thus, a total of 1024 x 1024 characters (over one million) can be represented by surrogates.

Character Properties

The definition of character properties used by ATSUI is derived from version 2.1.2 of the Unicode character properties database, definitions for Apple characters from the corporate private use zone, and changes for certain characters to make implementation easier. ATSUI uses the bidirectional character properties associated with version 2.1 of the Unicode Standard.

Specifically, ATSUI requires that whitespace and symmetric swapping-related properties be correctly set in the font. Characters not defined in the standard are assumed to be directionally neutral.

Font Feature Types and Selectors

This Appendix describes the Apple-defined font feature type and selector constants. Font features are grouped into categories called feature types, within which individual feature selectors are used to define particular feature settings or selections. You can use these feature types and selectors with the functions `ATSUSetFontFeatures` (page 37), `ATSUGetFontFeature` (page 38), `ATSUGetAllFontFeatures` (page 39), and `ATSUClearFontFeatures` (page 41) to set, obtain, or clear the font features of a style object.

Font vendors create tables that implement a set of font features from which your application can pick and choose. The architecture of font features is open-ended; as font vendors create new kinds of features, ATSUI automatically takes advantage of them. The initially-defined standard set of features is described in this chapter; as new fonts add new features, the defined set of font features will be expanded to accommodate them. For the most current list of Apple-defined font feature type and selector constants, you can see the Font Feature Registry at the Apple font web site:

<http://fonts.apple.com/Registry>

Table D-1 (page 272) lists examples of some of the feature types that a font can support and that your application can choose among when laying out text.

Note that unless the feature is defaulted differently in different fonts, the zero value for the selectors represents the default value.

Table D-1 Examples of feature types

Constant	Explanation
`kAllTypographicFeaturesType`	Specifies whether or not any font features are to be applied at all. Table D-2 (page 275) lists the feature selectors related to this feature type.
`kLigaturesType`	Specifies the use of required ligatures and other categories of optional ligatures. Table D-3 (page 276) lists the feature selectors related to this feature type.
`kCursiveConnectionType`	Specifies whether or not cursive connections are to be used between glyphs. Table D-4 (page 278) lists the feature selectors related to this feature type.
`kLetterCaseType`	Specifies case changes, such as all uppercase, all lowercase, and small caps, for scripts in which case has meaning. Table D-5 (page 278) lists the feature selectors related to this feature type.
`kVerticalSubstitutionType`	Allows substitution of vertical forms of particular glyphs (such as parentheses) in vertical runs of text. Table D-6 (page 279) lists the feature selectors related to this feature type.
`kLinguisticRearrangementType`	Either permits or inhibits linguistic (Indic-style) rearrangement of glyphs. Table D-7 (page 280) lists the feature selectors related to this feature type.
`kNumberSpacingType`	Specifies whether to use fixed-width or proportional-width glyphs for numerals. Table D-14 (page 286) lists the feature selectors related to this feature type.
`kSmartSwashType`	Controls whether swash variants of glyphs are to be substituted in specific places in the text, such as at the beginnings or ends of words or lines. Table D-8 (page 280) lists the feature selectors related to this feature type.
`kDiacriticsType`	Controls whether diacritical marks are shown or hidden. Table D-9 (page 281) lists the feature selectors related to this feature type.

Table D-1 Examples of feature types (continued)

Constant	Explanation
`kVerticalPositionType`	Controls the selection of superscript and subscript glyph sets. Table D-10 (page 282) lists the feature selectors related to this feature type.
`kFractionsType`	Controls automatic substitution or formation of fractions. Table D-11 (page 283) lists the feature selectors related to this feature type.
`kOverlappingCharactersType`	Controls whether long tails on glyphs are permitted to collide with other glyphs. Table D-12 (page 284) lists the feature selectors related to this feature type.
`kTypographicExtrasType`	Controls several effects, such as substitution of en dashes for hyphens, that are associated with sophisticated typography. Table D-23 (page 292) lists the feature selectors related to this feature type.
`kMathematicalExtrasType`	Controls several features, such as changing asterisks to multiplication symbols, used for typesetting mathematical expressions. Table D-24 (page 293) lists the feature selectors related to this feature type.
`kOrnamentSetsType`	Specifies certain sets of non-alphanumeric glyphs, such as decorative borders or musical symbols. Table D-25 (page 294) lists the feature selectors related to this feature type.
`kCharacterAlternativesType`	Specifies, by number, any font-specific set of alternate glyph forms. Table D-26 (page 295) lists the feature selector related to this feature type.
`kDesignComplexityType`	Specifies an overall complexity of appearance, as defined by the font. Table D-27 (page 295) lists the feature selectors related to this feature type.
`kStyleOptionsType`	Specifies any of several named alternative forms that may be available in the font, such as engraved or cursive. Table D-22 (page 291) lists the feature selectors related to this feature type.
`kCharacterShapeType`	Specifies for languages such as Chinese that have both sets whether traditional or simplified characters are to be used. Table D-13 (page 285) lists the feature selectors related to this feature type.

Table D-1 Examples of feature types (continued)

Constant	Explanation
`kNumberCaseType`	Specifies whether to use numerals that do, or do not, extend below the baseline. Table D-15 (page 286) lists the feature selectors related to this feature type.
`kTextSpacingType`	Specifies whether to use proportional, monospaced and half-width forms of characters in a font. Table D-16 (page 287) lists the feature selectors related to this feature type.
`kTransliterationType`	Allows text in one format to be displayed using another format. Table D-18 (page 288) lists the feature selectors related to this feature type.
`kAnnotationType`	Specifies annotations (or adornments) to basic lettershapes. For instance, most Japanese fonts include versions of numbers that are circled, parenthesized, have periods after them, and so on. Table D-17 (page 287) lists the feature selectors related to this feature type.
`kKanaSpacingType`	Specifies widths for Japanese Hiragana and Katakana characters. Table D-19 (page 289) lists the feature selectors related to this feature type.
`kIdeographicSpacingType`	Specifies whether to use proportional or full-width forms of ideographs (that is, Han-derived characters). Table D-20 (page 290) lists the feature selectors related to this feature type.
`kCJKRomanSpacingType`	Specifies whether to use proportional or half-width forms of Roman characters in a CJK (that is, Chinese, Japanse, and Korean) font. Table D-21 (page 290) lists the feature selectors related to this feature type.
`kUnicodeDecompositionType`	Table D-28 (page 296) lists the feature selectors related to this feature type.
`kLastFeatureType`	Represents the last Apple-reserved font feature type value.

Within some feature types, you can choose only one of the available feature selectors, such as whether numbers are to be proportional or fixed-width. With other feature types you can turn “on” or “off” any number of feature selectors

at once; for example, under ligatures you can choose any combination of the available classes of ligatures that the font supports.

Your application can select a group of features, place selectors for them in a style object's font features array, and assign that array to a text layout object's style object. ATSUI will then use those features, plus any font-specified features not overridden by your feature selections, when it draws the text layout object.

You can also turn font features on or off. Table D-2 lists the feature selectors for the `kAllTypographicFeaturesType` feature type; by specifying the selector `kAllTypeFeaturesOnSelector` or `kAllTypeFeaturesOffSelector` for that feature type, you can turn the entire set of features on or off. Note that if you turn all font features off this way, you turn off **all** font features, including all the font-specified defaults. (That may result in linguistically incorrect display.) If you turn font features on, you turn on the font-specified defaults, modified by whatever feature settings you have specified in the run-features array.

Table D-2 Feature selectors for the `kAllTypographicFeaturesType` font feature type

Constant	Explanation
`kAllTypeFeaturesOnSelector`	Tells ATSUI to use the font features specified in this style run's run-features array and the defaults specified by the font.
`kAllTypeFeaturesOffSelector`	Tells ATSUI to ignore all font features specified either by the font or in this style run's run-features array.

The rest of this section gives examples of the kinds of feature selectors that may be available for some of the feature types listed in Table D-1 (page 272). Please consult the font feature registry for more up-to-date information.

Contextual Font Features

One class of font features is contextual, meaning that how (or if) the feature is applied to a given glyph depends on the glyph's position compared to adjacent

glyphs. Much of ATSUI's text layout power results from its ability to apply sophisticated contextual processing.

ATSUI's ability to automatically substitute one or more glyphs for one or more other glyphs is called **automatic form substitution.** ATSUI supports several kinds of automatic form substitution, including ligatures, cursive contextual forms, contextual case substitution, vertical substitution, rearrangement, automatic fraction generation, and others.

A **ligature** is a rendering form that represents a combination of two or more individual characters. Examples include the "fi" ligature in English and the miim-miim ligature in Arabic.

A **contextual form** is an alternate appearance of a glyph that is used in certain contexts. Arabic, for example, has different contextual forms of characters, depending on whether they are at the beginning, the middle, or the end of a word. The same character code is used in each case; ATSUI chooses the correct glyph when laying out the text.

Ligatures

If the font supports the ligatures feature type, you can select features related to ligature formation, such as those shown in Table D-3.

Table D-3 Feature selectors for the `kLigaturesType` feature type

Constant	Explanation
`kRequiredLigaturesOnSelector` `kRequiredLigaturesOffSelector`	Allows or prevents the use of ligatures that the font designates as required by the language (such as certain Arabic ligatures).
`kCommonLigaturesOnSelector` `kCommonLigaturesOffSelector`	Allows or prevents the use of ligatures that the font designates as "common," or normally used (such as the "fi" ligature in Roman text).
`kRareLigaturesOnSelector` `kRareLigaturesOffSelector`	Allows or prevents the use of ligatures that the font designates as "rare" (such as "ct" or "ss" ligatures).

Table D-3 Feature selectors for the `kLigaturesType` feature type

Constant	Explanation
`kLogosOnSelector` `kLogosOffSelector`	Allows or prevents the use of ligatures that the font designates as logotypes (typically used for trademarks or other special display text).
`kRebusPicturesOnSelector` `kRebusPicturesOffSelector`	Allows or prevents the use of rebuses (pictures that represent words or syllables).
`kDiphthongLigaturesOnSelector` `kDiphthongLigaturesOffSelector`	Specifies whether or not to replace diphthong sequences, such as “AE” and “oe”, with their equivalent ligatures (“Æ” and “œ” in this case).
`kSquaredLigaturesOnSelector` `kSquaredLigaturesOffSelector`	Allows or prevents the use of ligatures where the component letters are arranged in a lattice, such that the ligature fits into the space of a single letter. For examples, see Unicode characters `U+3300` through `U+3357` and `U+337B` through `U+337F`.
`kAbbrevSquaredLigaturesOnSelector` `kAbbrevSquaredLigaturesOffSelector`	Allows or prevents the use of ligatures similar to the previously described ligatures, but in abbreviated form.

Cursive Connection

All Arabic fonts use cursive connection, and some Roman fonts may also support cursive connection. If a font supports the cursive connection feature type, you may be able to select features that either disable cursive connection completely, enable letterforms that connect in a noncontextual manner, or enable completely contextual, cursively connected letterforms (as in Arabic). Table D-4 lists the feature selectors for cursive connection.

Letter Case

In fonts for languages in which case is significant, ATSUI allows you to specify certain automatic case changes. If the font supports the letter case feature type,

Table D-4 Feature selectors for the `kCursiveConnectionType` feature type

Constant	Explanation
`kUnconnectedSelector`	Disables cursive connection.
`kPartiallyConnectedSelector`	Specifies noncontextual cursive connection.
`kCursiveSelector`	Specifies fully contextual cursive connection. For Arabic fonts, this selector is set by default.

you can select features that specify case changes such as those shown in Table D-5.

Table D-5 Feature selectors for the `kLetterCaseType` feature type

Constant	Explanation
`kUpperAndLowerCaseSelector`	Specifies no case conversion.
`kAllCapsSelector`	Specifies conversion of all letters to uppercase. (This feature is noncontextual.)
`kAllLowerCaseSelector`	Specifies conversion of all letters to lowercase. (This feature is noncontextual.)
`kSmallCapsSelector`	Specifies conversion of all lowercase letters to small caps. (This feature is noncontextual.)
`kInitialCapsSelector`	Specifies conversion of all lowercase letters at the beginnings of words to uppercase. (This feature is contextual.)
`kInitialCapsAndSmallCapsSelector`	Specifies conversion of all lowercase letters at the beginnings of words to uppercase, and all other lowercase letters to small caps. (This feature is contextual.)

Note
Contrary to common perception, the small caps style is not simply the use of capital letters in a smaller point size. If the font contains true small caps glyphs, you can specify them with a letter case feature selector, and ATSUI will use them. ◆

Vertical Substitution

Vertical substitution is a glyph substitution in which the glyph for a given glyph code is replaced by an alternate form in a vertical line. (This is not the same as rotating the glyph.) Table D-6 shows the feature selectors for vertical substitution.

Table D-6 Feature selectors for the `kVerticalSubstitutionType` feature type

Constant	Explanation
`kSubstituteVerticalFormsOnSelector` `kSubstituteVerticalFormsOffSelector`	Allows or prevents the substitution of alternate glyph forms in vertical lines.

For vertical substitution to happen, the vertically rotated forms must exist in the font and must be indicated as such in the font's tables; otherwise, no characters are substituted. If the font supports the vertical substitution feature type, its default behavior is to perform such substitutions; you may either prevent the substitution or allow it to occur.

Linguistic Rearrangement

Linguistic (Indic-style) rearrangement is a standard feature of Devanagari and other South Asian scripts. However, users may not always want it to occur, preferring instead to enter characters in an "already reversed" order. If a font supports the rearrangement feature type, you can either allow the default

behavior (which is to perform rearrangement) or you can prevent it. Table D-7 shows the feature selectors for rearrangement.

Table D-7 Feature selectors for the `kLinguisticRearrangementType` feature type

Constant	Explanation
`kLinguisticRearrangementOnSelector` `kLinguisticRearrangementOffSelector`	Allows or prevents the automatic rearrangement of certain glyphs as required by language rules.

Swashes and Smart Swashes

A **swash** is a variation, often ornamental, of an existing glyph. Using font tables, ATSUI can identify and automatically substitute swashes for existing glyphs. Alternatively, your application can allow the user to choose swash forms at the time the text layout object is created.

Collections of swash forms called **smart swashes** can be designated by the font designer and put in swash tables. Smart swashes are contextual and swashes are not. If the font supports the smart swashes feature type, you can select features that allow you to specify sets of swashes, such as shown in Table D-8.

Table D-8 Feature selectors for the `kSmartSwashType` feature type

Constant	Explanation
`kWordInitialSwashesOnSelector` `kWordInitialSwashesOffSelector`	Allows or prevents the substitution of swash variants that begin words.
`kWordFinalSwashesOnSelector` `kWordFinalSwashesOffSelector`	Allows or prevents the substitution of swash variants that end words.

Table D-8 Feature selectors for the `kSmartSwashType` feature type

Constant	Explanation
`kLineInitialSwashesOnSelector` `kLineInitialSwashesOffSelector`	Allows or prevents the substitution of swash variants that begin lines.
`kLineFinalSwashesOnSelector` `kLineFinalSwashesOffSelector`	Allows or prevents the substitution of swash variants that end lines.
`kNonFinalSwashesOnSelector` `kNonFinalSwashesOffSelector`	Allows or prevents the substitution of swash variants that can occur at the beginnings or interiors of words

Note
If you want your application to define its own set of swashes, it can use glyph substitutions to replace the ATSUI glyph choices with its own. ◆

Diacritical Marks

A glyph with a diacritical mark is a form of ligature. For fonts whose glyphs can take diacritical marks, ATSUI allows you several display options. If the font supports the diacritical marks feature type, you can specify that ATSUI should show, hide, or decompose diacritical marks, as shown in Table D-9.

Table D-9 Feature selectors for the `kDiacriticsType` feature type

Constant	Explanation
`kShowDiacriticsSelector`	Specifies that ATSUI is to form accent ligatures on the glyphs they apply to.
`kHideDiacriticsSelector`	Specifies that ATSUI is not to form any accent ligatures.
`kDecomposeDiacriticsSelector`	Specifies that ATSUI is to display marked glyphs as unmarked, followed by the accent ligatures as stand-alone glyphs.

For Roman fonts the default setting is to show diacritical marks. In text for scripts in which vowel marks are not normally shown, you can specify that marks be visible in certain instances, such as for children's text, or for pronunciation guides on rare words.

Vertical Position

For fonts that support the vertical position feature type, you can select features that allow you to specify glyph variants related to vertical position, as shown in Table D-10.

Table D-10 Feature selectors for the `kVerticalPositionType` feature type

Constant	Explanation
`kNormalPositionSelector`	Specifies use of normally positioned glyph set.
`kSuperiorsSelector`	Specifies use of superiors: glyph variants that are positioned above the baseline, used typically for superscripts.
`kInferiorsSelector`	Specifies use of inferiors: glyph variants that are positioned below the baseline, used typically for subscripts.
`kOrdinalsSelector`	Specifies contextual substitution of glyphs that replace ordinal designations attached to numerals (such as "1st" substituting for "1st").

Fractions

There are several ways to generate fractions with ATSUI. For a font that supports the fractions feature type, you may be able to select between two different types of automatic fraction generation, as shown in Table D-11.

Table D-11 Feature selectors for the `kFractionsType` feature type

Constant	Explanation
`kNoFractionsSelector`	Specifies no substitution or construction of fractions.
`kVerticalFractionsSelector`	Specifies replacement of slash-separated numeric sequences with pre-drawn fraction glyphs, if present in the font.
`kDiagonalFractionsSelector`	Specifies replacement of slash-separated numeric sequences with pre-drawn fraction glyphs, or else construction of fractions with numerators and denominators, or superiors and inferiors.

Note
To use the automatic fraction-generation capability, make sure that the slash separating the numerator and denominator is the fraction slash (character code 0xDA in the Standard Roman character set), not the normal slash character (0x2F). Automatic fraction generation does not occur unless the slash is a fraction slash. ◆

Prevention of Glyph Overlap

Some glyphs, especially certain initial swashes, have parts that extend well beyond their advance widths. An initial “Q”, for example, may have a tail that extends underneath the following “u”.

For fonts that support the glyph overlap feature type, you can specify that no glyph may overlap the outline of the following glyph. If it does, a

non-overlapping form of the glyph is substituted. Table D-12 lists the selectors for this feature.

Table D-12 Feature selectors for the `kOverlappingCharactersType` feature type

Constant	Explanation
`kPreventOverlapOnSelector` `kPreventOverlapOffSelector`	Prevents or allows the collision of an extended part of one glyph with an adjacent glyph.

Noncontextual Font Features

Noncontextual font features include the selection of alternate glyph sets to give text a different appearance, and glyph substitution for purposes of mathematical typesetting or enhancing typographic sophistication.

Character Shape

The Chinese language can be represented with both a traditional and a simplified character set. Chinese fonts that support the character shape feature type allow you to select either set.

Note
Historically on the Macintosh, the difference has been handled by having separate script systems for traditional Chinese and simplified Chinese; while that is still the case, this font feature makes it possible to have both glyph repertoires present in a single font. ◆

Table D-13 lists the selectors for this feature.

Table D-13 Feature selectors for the `kCharacterShapeType` feature type

Constant	Explanation
`kTraditionalCharactersSelector`	Specifies the use of traditional characters.
`kSimplifiedCharactersSelector`	Specifies the use of simplified characters.
`kJIS1978CharactersSelector`	Use character shapes for Japanese characters as defined by the JIS (Japanese Industrial Standard) C 6226-1978 document.
`kJIS1983CharactersSelector`	Use character shapes for Japanese characters as defined by the JIS X 0208-1983 document.
`kJIS1990CharactersSelector`	Use character shapes for Japanese characters as defined by the JIS X 0208-1990 document.
`kTraditionalAltOneSelector`	Use alternate set 1 of traditional forms for characters.
`kTraditionalAltTwoSelector`	Use alternate set 2 of traditional forms for characters.
`kTraditionalAltThreeSelector`	Use alternate set 3 of traditional forms for characters.
`kTraditionalAltFourSelector`	Use alternate set 4 of traditional forms for characters.
`kTraditionalAltFiveSelector`	Use alternate set 5 of traditional forms for characters.
`kExpertCharactersSelector`	Use "expert" forms of ideographs, such as are defined in the Fujitsu FMR character set.

Number Width

Many fonts support both proportional-width and fixed-width numeral. In proportional-width numerals the “1” is narrower than the “0”, whereas in

fixed-width numerals they (and all the other numerals) have identical widths. Fixed-width numerals are also called **columnating** because they align well in text that consists of columns of numerical data. For fonts that support the number spacing feature type, you can select either fixed-width or proportional-width numerals. Table D-14 lists the selectors for this feature.

Table D-14 Feature selectors for the `kNumberSpacingType` feature type

Constant	Explanation
`kMonospacedNumbersSelector`	Specifies the use of fixed-width (columnating) numerals.
`kProportionalNumbersSelector`	Specifies the use of proportional-width numerals.

Number Case

Some fonts support both lowercase (also called traditional or old-style) numerals, in which some glyphs extend below the baseline, and uppercase (also called **lining**) numerals, in which no glyphs extend below the baseline. For fonts that support the number case feature type, you can select either kind of numeral. Table D-15 lists the selectors for this feature.

Table D-15 Feature selectors for the `kNumberCaseType` feature type

Constant	Explanation
`kLowerCaseNumbersSelector`	Specifies the use of lowercase (old-style) numerals.
`kUpperCaseNumbersSelector`	Specifies the use of uppercase (lining) numerals.

Text Width

The text spacing feature type is used to select between the proportional, monospaced and half-width forms of characters in a font. Use of this feature type is optional; for more precise control see “Kana Spacing” (page 289) and

"Ideographic Spacing" (page 290). This is an exclusive feature type. Table D-16 lists the selectors for this feature.

Table D-16 Feature selectors for the `kTextSpacingType` feature type

Constant	Explanation
`kProportionalTextSelector`	Selects the proportional forms of letters.
`kMonospacedTextSelector`	Selects the monospace forms of letters.
`kHalfWidthTextSelector`	Selects the half-width forms of letters.
`kNormallySpacedTextSelector`	Selects the default forms of letters.

Annotation

The annotation feature type specifies annotations (or adornments) to basic lettershapes. For instance, most Japanese fonts include versions of numbers that are circled, parenthesized, have periods after them, and so on. This is an exclusive feature type. Table D-17 lists the selectors for this feature.

Table D-17 Feature selectors for the `kAnnotationType` feature type

Constant	Explanation
`kNoAnnotationSelector`	Indicates that characters should appear without annotation.
`kBoxAnnotationSelector`	Use the forms of characters surrounded by a box cartouche.
`kRoundedBoxAnnotationSelector`	Use the forms of characters surrounded by a box cartouche with rounded corners.
`kCircleAnnotationSelector`	Use the forms of characters surrounded by a circle. For instance, see Unicode characters `U+3260` through `U+326F`.

Table D-17 Feature selectors for the `kAnnotationType` feature type

Constant	Explanation
`kInvertedCircleAnnotationSelector`	Same as circle annotation, but with white and black reversed. For instance, see Unicode characters `U+2776` through `U+277F`.
`kParenthesisAnnotationSelector`	Use the forms of characters surrounded by parentheses. For instance, see Unicode characters `U+2474` through `U+2487`.
`kPeriodAnnotationSelector`	Use the forms of characters followed by a period. For instance, see Unicode characters `U+2488` through `U+249B`.
`kRomanNumeralAnnotationSelector`	Display characters in their Roman numeral form.
`kDiamondAnnotationSelector`	Display the text surrounded by a diamond.

Transliteration

The transliteration feature types allows text in one format to be displayed using another format. An example is taking a Hiragana string and displaying it as Katakana. Table D-18 lists the selectors for this feature.

Table D-18 Feature selectors for the `kTransliterationType` feature type

Constant	Explanation
`kNoTransliterationSelector`	Allows no transliteration.
`kHanjaToHangulSelector`	Allows text in Hanja to be displayed using Hangul.
`kHiraganaToKatakanaSelector`	Allows text in Hiragana to be displayed using Katakana.
`kKatakanaToHiraganaSelector`	Allows text in Katakana to be displayed using Hiragana.

Table D-18 Feature selectors for the `kTransliterationType` feature type

Constant	Explanation
`kKanaToRomanizationSelector`	Allows text in Kana to be displayed using Romanization.
`kRomanizationToHiraganaSelector`	Allows text in Romanization to be displayed using Hiragana.
`kHanjaToHangulAltOneSelector`	Allows text in Hanja to be displayed using Hangul, Alternative Set 1.
`kHanjaToHangulAltTwoSelector`	Allows text in Hanja to be displayed using Hangul, Alternative Set 2.
`kHanjaToHangulAltThreeSelector`	Allows text in Hanja to be displayed using Hangul, Alternative Set 3.

Kana Spacing

The Kana Spacing feature type is used to select widths specifically for Japanese Hiragana and Katakana characters. Table D-19 lists the selectors for this feature.

Table D-19 Feature selectors for the `kKanaSpacingType` feature type

Constant	Explanation
`kFullWidthKanaSelector`	Selects the full width forms of kana.
`kProportionalKanaSelector`	Selects the proportional forms of kana.

Ideographic Spacing

The ideographic spacing feature type is used to select between full-width and proportional forms of ideographs (that is, Han-derived characters). Table D-20 lists the selectors for this feature.

Table D-20 Feature selectors for the `kIdeographicSpacingType` feature type

Constant	Explanation
`kFullWidthIdeographsSelector`	Selects the full width forms of ideographs.
`kProportionalIdeographsSelector`	Selects the proportional forms of ideographs.

CJK Roman Width

The CJK Roman spacing feature type is used to select between the proportional and half-width forms of Roman characters in a CJK (that is, Chinese, Japanese, Korean) font. Table D-21 lists the selectors for this feature.

Table D-21 Feature selectors for the `kCJKRomanSpacingType` feature type

Constant	Explanation
`kHalfWidthCJKRomanSelector`	Selects the half-width forms of letters.
`kProportionalCJKRomanSelector`	Selects the proportional forms of letters.
`kDefaultCJKRomanSelector`	Selects the default Roman forms of letters.
`kFullWidthCJKRomanSelector`	Selects the full-width Roman forms of letters.

Style Options

An ATSUI-compatible font may offer named sets of noncontextual glyph substitutions that give the text a specific style or appearance. You can select among sets, using selectors such as those listed in Table D-22.

Table D-22 Feature selectors for the `kStyleOptionsType` feature type

Constant	Explanation
`kNoStyleOptionsSelector`	Specifies the use of the standard glyph set.
`kDisplayTextSelector`	Specifies the use of a glyph set that is designed for best display at large sizes (over 24 point).
`kEngravedTextSelector`	Specifies the use of a glyph set that has contrasting strokes parallel to the main stroke, giving an engraved effect.
`kIlluminatedCapsSelector`	Specifies the use of a glyph set with complex decoration surrounding the glyphs of capital letters.
`kTitlingCapsSelector`	Specifies the use of a glyph set in which capital letters have a special form for display in titles.
`kTallCapsSelector`	Specifies the use of a glyph set in which capital letters have a taller form than is typical.

You may be able to select more than one feature at a time from the list of alternate forms. For example, a font may offer display, engraved, and engraved-display style options.

Typographic Extras

Fonts that support the typographic extras feature type allow you to specify certain small-scale typographic conventions, using selectors such as those shown in Table D-23.

Table D-23 Feature selectors for the `kTypographicExtrasType` feature type

Constant	Explanation
`kHyphensToEmDashOnSelector` `kHyphensToEmDashOffSelector`	Allows or prevents the automatic replacement of two adjacent hyphens with an em dash.
`kHyphenToEnDashOnSelector` `kHyphenToEnDashOffSelector`	Allows or prevents the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen- numeral) with an en-dash.
`kSlashedZeroOnSelector` `kSlashedZeroOffSelector`	Allows or prevents the forced use of the un-slashed zero glyph, regardless of whether the font specifies the slashed zero as the default.
`kFormInterrobangOnSelector` `kFormInterrobangOffSelector`	Allows or prevents the automatic replacement of the sequence “?!” or “!?” with the font’s interrobang glyph.
`kSmartQuotesOnSelector` `kSmartQuotesOffSelector`	Allows or prevents the automatic contextual replacement of straight quotation marks with curly ones.
`kPeriodsToEllipsisOnSelector` `kPeriodsToEllipsisOffSelector`	Allows or prevents the automatic replacement of two adjacent periods with an ellipsis.

Mathematical Extras

Fonts that support the mathematical extras feature type allow you to specify certain math-formatting conventions, using selectors such as those shown in Table D-24.

Table D-24 Feature selectors for the `kMathematicalExtrasType` feature type

Constant	Explanation
`kHyphenToMinusOnSelector` `kHyphenToMinusOffSelector`	Allows or prevents the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen-numeral) with a minus sign glyph (–).
`kAsteriskToMultiplyOnSelector` `kAsteriskToMultiplyOffSelector`	Allows or prevents the automatic replacement of the sequence space-asterisk-space (or the asterisk in the sequence numeral-asterisk-numeral) with a multiplication sign glyph (×).
`kSlashToDivideOnSelector` `kSlashToDivideOffSelector`	Allows or prevents the automatic replacement of the sequence space-slash-space (or the slash in the sequence numeral-slash- numeral) with a division sign glyph (÷).
`kInequalityLigaturesOnSelector` `kInequalityLigaturesOffSelector`	Allows or prevents the automatic replacement of sequences such as “>=” and “<=” with equivalent ligatures “≥” and “≤”.
`kExponentsOnSelector` `kExponentsOffSelector`	Allows or prevents the automatic replacement of the sequence *exponentiation glyph*—numerals with the superior forms of the numerals. An example of an exponentiation glyph is “^”.

Note
By convention, specifying the `kHyphenToMinusOnSelector` in the mathematical extras feature type overrides specifying the `kHyphenToEnDashOnSelector` in the typographic extras feature type. ◆

Ornament Sets

Fonts may include ornamental, nonalphabetic glyph sets used for various purposes. With a font that supports the ornament set feature type, you may be able to select among those glyph sets, using selectors such as those shown in Table D-25.

Table D-25 Feature selectors for the `kOrnamentSetsType` feature type

Constant	Explanation
`kNoOrnamentsSelector`	Specifies the use of no ornamental glyph sets.
`kDingbatsSelector`	Specifies the use of dingbats: arrows, stars, bullets, and so on.
`kPiCharactersSelector`	Specifies the use of pi characters: related nonalphabetic symbols, such as musical notation glyphs.
`kFleuronsSelector`	Specifies the use of fleurons: ornaments such as flowers, vines, and leaves.
`kDecorativeBordersSelector`	Specifies the use of decorative borders: glyphs used in interlocking patterns to form text borders.
`kInternationalSymbolsSelector`	Specifies the use of international symbols, such as the barred circle representing “no”.
`kMathSymbolsSelector`	Specifies the use of mathematical symbols.

Character Alternates

This feature type gives a font a very general way to provide different sets of glyphs. Sets are numbered sequentially. For a font that supports the character alternates feature type, you can select by number any of the sets it provides.

For example, a font with 20 ampersands could place them in 20 selectors under this feature type. In general, however, named glyph sets provided through the `kCharacterAlternativesType` feature type are preferable. Table D-26 lists the only defined selector for this feature.

Table D-26 Feature selectors for the `kCharacterAlternativesType` feature type

Constant	Explanation
`kNoAlternatesSelector`	Specifies the use of no character alternatives. This is the first (default) setting for this feature type; others are specified by number only.

Design Complexity

Some fonts may have several glyph sets that represent different designs from the same font-family, such as “plain” or “fancy.” For a font that supports the design complexity feature type, design levels are numbered, and you can select any available level by number or by selectors such as those shown in Table D-27.

Table D-27 Feature selectors for the `kDesignComplexityType` feature type

Constant	Explanation
`kDesignLevel1Selector`	Specifies the basic glyph set.
`kDesignLevel2Selector`	Specifies an alternate glyph set.
`kDesignLevel3Selector`	Specifies an alternate glyph set.
`kDesignLevel4Selector`	Specifies an alternate glyph set.
`kDesignLevel5Selector`	Specifies an alternate glyph set.

Unicode Decomposition

For a font that supports the unicode decomposition type, you can select any available level by number or by selectors such as those shown in Table D-28.

Table D-28 Feature selectors for the `kUnicodeDecompositionType` feature type

Constant
`kCanonicalDecompositionOnSelector`
`kCanonicalDecompositionOffSelector`
`kCompatibilityDecompositionOnSelector`
`kCompatibilityDecompositionOffSelector`
`kTranscodingDecompositionOnSelector`
`kTranscodingDecompositionOffSelector`

Glossary

advance height The distance from the top of a glyph to the bottom of the glyph, including the top-side bearing and bottom-side bearing.

advance width The full horizontal width of a glyph as measured from its origin to the origin of the next glyph on the line, including the side bearings on both sides.

alignment The process of placing text in relation to one or both margins.

alphabetic writing system The glyphs that symbolize discrete phonemic elements in a language. Compare **syllabic writing system** and **ideographic writing system.**

angled caret A caret whose angle in relation to the baseline of the display text is equivalent to the slant of the glyphs making up the text. Compare **straight caret.**

ascent line An imaginary horizontal line that corresponds approximately to the tops of the uppercase letters in the font. Uppercase letters are chosen because, among the regularly used glyphs in a font, these are generally the tallest.

automatic form substitution The process of automatically substituting one or more glyphs for one or more other glyphs.

baseline An imaginary line used to align glyphs in a line of text.

baseline delta An offset (in points) between the various baseline types and y = 0. See **baseline type.**

baseline type The classification of baseline used with a particular kind of text. See, for example, **Roman baseline.**

bidirectional script system A script system where text is generally right-aligned with most characters written from right to left, but with some left-to-right text as well. Arabic and Hebrew are bidirectional script systems.

bottom-side bearing The white space between the bottom of the glyph and the visible ending of the glyph.

bounding box The smallest rectangle that entirely encloses the pixels or outline of a glyph.

caret A vertical or slanted blinking bar, appearing at a caret position in the display text, that marks the point at which text is to be inserted or deleted. Compare **split caret.**

caret angle The angle of a caret or the edges of a highlight. The caret angle can be perpendicular to the baseline or parallel to the angle of the style run's text.

caret position A location on screen, typically between glyphs, that relates directly to the offset (in memory) of the current text insertion point in the source text. At the boundary between a right-to-left and left-to-right direction run on a line, one

character offset may correspond to two caret positions, and one caret position may correspond to two offsets.

caret type A designation of the behavior of the caret at direction boundaries in text. See **split caret.**

character A symbol standing for a sound, syllable, or notion used in writing; one of the simple elements of a written language, for example, the lowercase letter "a" or the number "1". Compare **character code, glyph.**

character cluster A collection of characters treated as individual components of a whole, including a principal character plus attachments in memory. For example, in Hebrew, a cluster may be composed of a consonant, a vowel, a dot to soften the pronunciation of the consonant, and a cantillation mark.

character code In ATSUI, a 16-bit value representing a Unicode text character. Text is stored in memory as character codes. Each script system's keyboard-layout (`'KCHR'`) resource converts the virtual key codes generated by the keyboard or keypad into character codes; each script system's fonts convert the character codes into glyphs for display or printing.

character encoding An internal conversion table for interpreting a specific character set.

character offset The indexed position of a 2-byte Unicode character in a text buffer, starting at zero for the first character. Sequential values for character offset correspond to the **storage order** of the characters. (2) The horizontal separation between a character rectangle and a font rectangle—that is, the position of a given character within the font's bit image.

contextual form An alternate form of a glyph whose use depends on the glyph's placement in a word.

counter The oval in glyphs such as "p" or "d".

cross-stream kerning The automatic movement of glyphs perpendicular to the line orientation of the text. Compare **with-stream kerning.**

cross-stream shift A type of positional shift that applies equally to all glyphs in a style run by raising or lowering the entire style run (or shifts it sideways if it's vertical text). Compare **with-stream shift.**

cursor A small icon, often an arrow or an I-beam shape, that moves with the mouse or other pointing device. Compare **caret.**

descent line An imaginary horizontal line that usually corresponds with the bottoms of the descenders in a font. The descent line is the same distance from the baseline for all glyphs in the font, whether or not they have descenders.

direction See **dominant direction, glyph direction, line direction, text direction.**

direction boundary A point between offsets in memory or glyphs in a display, at which the direction of stored or displayed text changes.

direction level A hierarchical ranking of dominant direction in a line. Direction levels can be nested so that complex mixed-direction formatting is preserved.

direction-level run A sequence of contiguous glyphs that share the same text direction.

direction override A means of overriding the directional behavior of glyphs, on a style-run basis, for special effects.

discontiguous highlighting Highlighting that exactly matches the selection range it corresponds to. It may consist of discontiguous areas when the selection range crosses direction boundaries. Compare **contiguous highlighting.**

display order The left-to-right order in which ATSUI displays glyphs. Display order determines the glyph index of each glyph in a line and may differ from the input order of the text. See **glyph index;** compare **input order** and **source text.**

display text The visual representation of the text of a text layout object. Display text consists of a sequence of glyphs, arranged in display order. Compare **source text.**

dominant direction The direction in which successive groups of glyphs are read. Dominant direction is independent of glyph direction. See also **glyph direction, line direction.**

drop capital A large uppercase letter that drops below the main line of text for aesthetic reasons.

dual caret See **split caret.**

dynamic highlighting The process of continually drawing and redrawing the highlighted area as the user moves the cursor through the text while holding down the mouse button.

edge offset A byte offset into the source text of a layout shape that specifies a position between byte values. Edge offsets in source text are related to caret positions in display text. Compare **caret position** and **byte offset.**

feature selectors A means of defining particular font features in a feature type. See also **feature type.**

feature type A group of font features in a style object that are applied to each style run based on font defaults. See also **feature selectors.**

font A collection of glyphs that usually have some element of design consistency such as the shapes of the counters, the design of the stem, stroke thickness, or the use of serifs.

font attributes A group of flags that modify the behavior or identity of a font.

font embedding The technique of storing a font object's binary data in a document so that the text in the document always displays the correct font.

font family A group of fonts that share certain characteristics and a common family name.

font features The set of typographic and layout capabilities that create a specific appearance for a layout shape.

font instance A setting identified by the font's designer that matches specific values along the available variation axes and gives those values a name.

font name A set of specific information in a font object about a font, such as its family name, style, copyright date, version, and manufacturer. Some font names are used to build menus in an application, whereas other names are used to identify the font uniquely.

font object An object type that hides the complexity of font data from your application.

font variation An algorithmic way to produce a range of typestyles along a particular variation axis.

font variation suite A complete listing of every axis supported in a font in the order specified by the font. Each axis is given a value in the listing.

glyph The distinct visual representation of a character in a form that a screen or printer can display. A glyph may represent one character (the lowercase *a*), more than one character (the *fi* ligature), part of a character (the dot over an *i*), or a nonprinting character (the space character). See also **character.**

glyph code A number that specifies a particular glyph in a font. Fonts map character codes to glyph codes using Unicode `'cmap'` tables, which in turn specify individual glyphs. If a font does not have a Unicode `'cmap'` table, it is generated automatically.

glyph direction The direction in which successive glyphs are read. Compare **dominant direction.**

glyph ductility The ability to stretch the actual form of a glyph during justification.

glyph index The order of a glyph in a line of display text. The leftmost glyph in a line of text has a glyph index of 1; each succeeding glyph to the right has an index one greater than the previous glyph. Compare **glyph code, edge offset.**

glyph origin The point that ATSUI uses to position a glyph when drawing.

grow limit The maximum amount by which glyphs of a given priority can be extended during justification, before processing passes to glyphs of lower priority. Compare **shrink limit.**

hanging baseline The baseline used by Devanagari and similar scripts, where most of the glyph is below the baseline.

hanging glyphs A set of glyphs, usually punctuation, that typically extend beyond the left and right margins of the text area and whose widths are not counted when line length is measured.

highlighting The display of text in inverse video or with a colored background. Highlighting in display text corresponds to a selection range in source text.

highlight type The angular character of carets and edges of highlighting areas. Highlighting and carets are either straight or angled; see **angled caret, straight caret.**

hit-testing The process of converting a location within a line of display text into a caret offset in the source text of that line.

ideographic centered baseline The baseline used by Chinese, Japanese, and Korean ideographic scripts, in which glyphs are centered halfway on the line height.

ideographic writing system The glyphs that symbolize component meanings of words in a language. Compare **syllabic writing system** and **alphabetic writing system.**

imposed width A run control feature that forces a specific width onto the glyphs of a style run, regardless of its text content or other style properties.

index See **glyph index.**

input order The order in which characters are written or entered from a keyboard. The input order of a line of text can differ from its display order. Compare **display order.**

insertion point The point in the source text at which text is to be inserted or deleted. An insertion point is specified by a single caret position. Compare **caret**; see also **caret position.**

justification The process of typographically expanding or compressing a line of text to fit a text width.

justification gap The difference in the length of a line before and after justification.

justification priority The priority order in which classes of glyphs are processed during justification.

kashida An extension-bar glyph that is added to certain Arabic glyphs during justification.

kerning An adjustment to the normal spacing that occurs between two or more specifically named glyphs, known as the *kerning pair*.

kerning adjustments array An array in the style object that overrides the normal kerning for individual pairs of glyphs by specifying a point-size factor and scaling factor.

kerning pair Two specifically named glyphs that are kerned together by a set amount. See also **kerning.**

language The written and spoken methods of combining words to create meaning used by a particular group of people.

leading edge The edge of a glyph that is encountered first when reading text of that glyph's language. For glyphs of left-to-right text, the leading edge is the left edge; for glyphs of right-to-left text, the leading edge is the right edge.

left-side bearing The white space between the glyph origin and the visible beginning of the glyph.

ligature Two or more glyphs connected to form a single new glyph.

ligature decomposition The replacement of ligatures with the glyphs for their component characters during justification.

ligature splitting The process of separating a ligature into its component glyphs.

line breaking The process of determining the proper location at which to truncate a line of text so that it fits within a given text width.

line direction The overall direction in which a line of text is read. Line direction is the lowest nested level of dominant direction on a line.

line length The distance, in points, from the origin of the first glyph on a line through the advance width of the last glyph.

line span The distance, in points, from the lowest descender on a line to the highest ascender.

margins The left, right, top, and bottom sides of the text area.

math baseline The baseline used for setting mathematical expressions; it is centered on operators such as the minus sign.

mixed-direction text The combination of text with both left-to-right and right-to-left directions within a single line of text.

neutral type A glyph directionality in which the glyph direction is always that of the surrounding glyphs. Compare **strong type, weak type.**

point size The size of a font's glyphs as measured from the baseline of one line of text to the baseline of the next line of single-spaced text. In the United States, point size is measured in typographic points.

postcompensation action The extra processing, such as addition of kashidas and ligature decomposition, that occurs after glyphs have been repositioned during justification.

priority justification override array An array that alters the standard justification behavior for all glyphs of a given justification priority.

right-side bearing The white space on the right side of the glyph; this value may or may not be equal to the value of the left-side bearing.

Roman baseline The baseline used in most Roman scripts and in Arabic and Hebrew.

run A sequence of glyphs that are contiguous in memory and share a set of common attributes.

script A method for depicting words visually.

selection range The contiguous sequence of characters in the source text that mark where the next editing operation is to occur. The glyphs corresponding to those characters are commonly highlighted on screen.

serif The fine lines stemming from and at an angle to the upper and lower ends of the main strokes of a letter—for example, the little "feet" on the bottom of the vertical strokes in the upper-case letter "M" in Times Roman typeface.

style run text attributes The set of flags that allow you to specify how ATSUI alters glyph outlines or chooses the proper metrics for horizontal or vertical text.

shrink limit The maximum amount by which glyphs of a given priority may be compressed during justification, before processing passes to glyphs of lower priority. Compare **grow limit.**

smart swash A variation of an existing glyph (often ornamental) that is contextual. Compare **swash**.

soft line break Line breaks within a text layout object. If you call `ATSUBreakLine` (page156), it will determine the best location for line breaks. You can set your own line breaks by calling `ATSUSetSoftLineBreak` (page159).

source text A stored sequence of character codes that represents a line of text. Characters in source text are stored in input order. Compare **display order, display text;** see also **input order.**

split caret A type of caret that, at the boundary between text of opposite directions, divides into two parts: a high caret and a low caret, each measuring half the line's height. The two separate half-carets merge into one in unidirectional text.

storage order See **input order**, **display order, source text.**

straight caret A caret that is perpendicular to the baseline of the display text, regardless of the angle of the glyphs making up the text. Compare **angled caret.**

strong type A glyph directionality that is always left to right or right to left. Compare **weak type, neutral type.**

style run A sequence of memory backing store contiguous glyphs that share the same style.

swash A variation of an existing glyph (often ornamental) that is noncontextual. Compare **smart swash**.

syllabic writing system The glyphs that symbolize syllables in a language. Compare **alphabetic writing system** and **ideographic writing system.**

text A set of specific symbols that, when displayed in a meaningful order, conveys information.

text area The space on the display device within which the text should fit.

text direction The direction in which reading proceeds. Roman text has a left-to-right direction; Hebrew and Arabic have a (predominantly) right-to-left direction; Chinese and Japanese can have a vertical direction.

text run A complete unit of text, made up of character codes or glyph codes.

text width The area between the margins; it is the length available for displaying a line of text.

tiled highlighting A highlighting mechanism whereby the highlighted area corresponding to every character in a line of text is unique, without gaps or overlaps.

top-side bearing The white space between the top of the glyph and the visible beginning of the glyph.

tracking Kerning between all glyphs in the shape, not just the kerning pairs already defined by the font. You can increase or decrease interglyph spacing by using a track number. See **kerning.**

track setting A value that specifies the relative tightness or looseness of interglyph spacing.

trailing edge The edge of a glyph that is encountered last when reading text of that glyph's language. For glyphs of left-to-right text, the trailing edge is the right edge; for glyphs of right-to-left text, the trailing edge is the left edge.

typestyle A variant version of glyphs in the same font family. Typical typestyles available on the Macintosh computer include bold, italic, underline, outline, shadow, condensed, and extended.

typographic bounding rectangle The smallest rectangle that encloses the full span of the glyphs from the ascent line to the descent line.

typographic point A unit of measurement describing the size of glyphs in a font. There are 72.27 typographic points per inch, as opposed to 72 points per inch in ATSUI.

unidirectional text A sequence of text that has a single direction. Compare **mixed-direction text.**

unlimited gap absorption The assignment of all justification gap to an individual glyph or priority of glyphs, regardless of the specified grow or shrink limits for that glyph or glyphs.

variation axis A range included in a font by the font designer that allows a font to produce different typestyles.

weak type A glyph directionality that depends on context to determine whether it is left to right or right to left. Compare **strong type, neutral type.**

with-stream kerning The automatic movement of glyphs parallel to the line orientation of the text. Compare **cross-stream kerning.**

with-stream shift A positional shift that applies equally to all glyphs in a style run by adding or removing space before or after each glyph in the run. Compare **cross-stream shift.**

WorldScript A group of Macintosh system software managers, extensions, and resources that facilitate multilanguage text processing.

x-height The position where the top of the lowercase "x" in the font lies; this measurement usually marks the height of the body of all lowercase glyphs, excluding ascenders and descenders, in the font.

Index

A

B

C, D, E

F

G, H, I

J

K, L

M–T

U–Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Lisa Karpinski

TECHNICAL CONSULTANT
Jun Suzuki

PROJECT LEAD WRITER
Donna Lee

WRITING MANAGER
Tony Francis

ILLUSTRATOR
Ruth Anderson

PRODUCTION EDITOR
Gerri Gray

Special thanks to Andy Daniels, Dan Fenwick, Dave Opstad