INSIDE MACINTOSH

Apple Type Services for Unicode Imaging Reference

For ATSUI 1.2

January 11, 2000
Technical Publications

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

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction

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

This document describes the ATSUI application programming interface (API) through version 1.2 and may be useful to developers who are writing new text editors or word processing applications that renders Unicode-encoded text. You can also use the ATSUI API if you wish to modify your existing application to support Unicode text rendering.

If you are a font designer or want more information about fonts, see the Apple font group site:

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

This document describes the ATSUI API in the following chapters:

- "ATSUI Reference" (page 19) describes the complete API through ATSUI 1.2, including functions, data types, constants, and result codes.
- "Document Revision History" (page 235) provides a history of revisions to this document.
- "History of API Additions and Changes in ATSUI" (page 237) provides a history of changes to the API since ATSUI 1.0.
- "Summary of Style Run and Text Layout Attribute Tag Information" (page 241) provides tabular summaries of the Apple-defined style run and text layout attribute tags and the data type, size, and default values of the attributes they identify.
- "New Constants and Data Types Used by ATSUI" (page 247) describes new constants and data types used by ATSUI from Unicode Utilities, the Script Manager, AAT Font Table Text Rendering Utilities, and AAT Font Table Text Layout Utilities.

- "Glossary" (page 315) provides an alphabetical listing of typographic and ATSUI-specific terms.

ATSUI Reference

Contents

This chapter describes the ATSUI application programming interface (API) through ATSUI version 1.2, as follows:

Gestalt Constants

Gestalt Selectors for ATSUI

Before calling any functions dependent upon ATSUI, you should pass the `gestaltATSUVersion` selector to the `Gestalt` function to determine which version of ATSUI is available. You should pass the `gestaltATSUFeatures` selector to `Gestalt` to determine which features of ATSUI are available.

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

Constant descriptions

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

ATSUI that is installed on the user's system. You can also determine version information by testing for the feature bits described in "ATSUI Attribute Constants" (page 21).

`gestaltATSUFeatures` The `Gestalt` selector you pass to determine which features of ATSUI are available. On return, `Gestalt` passes back a 32-bit mask, described in "ATSUI Attribute Constants" (page 21), which you can test to determine which features are available.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUI Version Constants

When you pass the `gestaltATSUVersion` selector to the `Gestalt` function, on return, `Gestalt` passes back one of these constants indicating which version of ATSUI is available.

```
enum {
    gestaltOriginalATSUVersion  = (1 << 16),
    gestaltATSUUpdate1          = (2 << 16),
    gestaltATSUUpdate2          = (3 << 16)
};
```

Constant descriptions

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

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

`gestaltATSUUpdate2`
A `Fixed` value that indicates that version 1.2 of ATSUI is installed on the user's system. Available beginning with ATSUI 1.2.

VERSION NOTES

This enumeration is available beginning with ATSUI 1.0. Additional constants added with ATSUI 1.1 and 1.2.

ATSUI Attribute Constants

When you pass the `gestaltATSUFeatures` selector to the `Gestalt` function, on return, `Gestalt` passes back a bit mask which you can use to test for available features in ATSUI.

You can also test the bit mask to determine which version of ATSUI is installed. If any of the bits specified by the mask constants `gestaltATSUTrackingFeature`, `gestaltATSUMemoryFeature`, `gestaltATSUFallbacksFeature`, `gestaltATSUGlyphBoundsFeature`, `gestaltATSULineControlFeature`, `gestaltATSULayoutCacheClearFeature`, or `gestaltATSULayoutCreateAndCopyFeature` are set, ATSUI 1.1 is installed. If the bit specified by the mask constant `gestaltATSUTextLocatorUsageFeature` is set, ATSUI 1.2 is installed.

```
enum {
    gestaltATSUTrackingFeature              = 0x00000001,
    gestaltATSUMemoryFeature                = 0x00000001,
    gestaltATSUFallbacksFeature             = 0x00000001,
    gestaltATSUGlyphBoundsFeature           = 0x00000001,
    gestaltATSULineControlFeature           = 0x00000001,
    gestaltATSULayoutCacheClearFeature      = 0x00000001,
    gestaltATSULayoutCreateAndCopyFeature   = 0x00000001,
    gestaltATSUTextLocatorUsageFeature      = 0x00000002
};
```

Constant descriptions

`gestaltATSUTrackingFeature`

If the bit specified by this mask constant is set, the functions `ATSUCountFontTracking` (page 68) and `ATSUGetIndFontTracking` (page 69) are available. Available beginning with ATSUI 1.1.

`gestaltATSUMemoryFeature`

If the bit specified by this mask is set, the functions `ATSUCreateMemorySetting` (page 174), `ATSUSetCurrentMemorySetting` (page 176),

ATSUGetCurrentMemorySetting (page 176), and ATSUDisposeMemorySetting (page 177) are available. Available beginning with ATSUI 1.1.

gestaltATSUFallbacksFeature
: If the bit specified by this mask is set, the functions ATSUSetFontFallbacks (page 123) and ATSUGetFontFallbacks (page 124) are available. Available beginning with ATSUI 1.1.

gestaltATSUGlyphBoundsFeature
: If the bit specified by this mask is set, the function ATSUGetGlyphBounds (page 146) is available. Available beginning with ATSUI 1.1.

gestaltATSULineControlFeature
: If the bit specified by this mask is set, the functions ATSUCopyLineControls (page 103), ATSUSetLineControls (page 104), ATSUGetLineControl (page 106), ATSUGetAllLineControls (page 108), and ATSUClearLineControls (page 109) are available. Available beginning with ATSUI 1.1.

gestaltATSULayoutCacheClearFeature
: If the bit specified by this mask is set, the function ATSUClearLayoutCache (page 93) is available. Available beginning with ATSUI 1.1.

gestaltATSULayoutCreateAndCopyFeature
: If the bit specified by this mask is set, the function ATSUCreateAndCopyTextLayout (page 91) is available. Available beginning with ATSUI 1.1.

gestaltATSUTextLocatorUsageFeature
: If the bit specified by this mask is set, the text break locator attribute is available for both style and text layout objects. Available beginning with ATSUI 1.2.

VERSION NOTES

This enumeration is available beginning with ATSUI 1.1. Additional constants added with ATSUI 1.2.

Functions

Creating, Manipulating, and Disposing of Style Objects

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

- `ATSUCreateStyle` (page 24) creates a style object.
- `ATSUCreateAndCopyStyle` (page 25) creates a copy of a style object.
- `ATSUCompareStyles` (page 26) compares the contents of two style objects.
- `ATSUSetStyleRefCon` (page 27) sets application-specific style object data.
- `ATSUGetStyleRefCon` (page 28) obtains application-specific style object data.
- `ATSUStyleIsEmpty` (page 29) indicates whether a style object is empty.
- `ATSUClearStyle` (page 29) removes all previously set style run attribute, font feature, and font variation values from a style object.
- `ATSUDisposeStyle` (page 30) disposes of a style object.

ATSUCreateStyle

Creates a style object.

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

`oStyle` A pointer to a reference of type `ATSUStyle` (page 191). On return, the newly-created style object. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUCreateStyle` function creates an "empty" style object that contains default style run attribute, font feature, and font variation values. The default font feature and variation values are defined by the font; default style attribute values are described in Table C-1 (page 241). You can set the style run attribute values in a style object by calling the function `ATSUSetAttributes` (page 37). To set font features and font variations in a style object, call the functions `ATSUSetFontFeatures` (page 44) and `ATSUSetVariations` (page 50), respectively.

To create a copy of an existing style object, call the function `ATSUCreateAndCopyStyle` (page 25). You can copy a portion of a style object into an existing style object by calling the functions `ATSUCopyAttributes` (page 31), `ATSUOverwriteAttributes` (page 32), and `ATSUUnderwriteAttributes` (page 34).

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 beginning with ATSUI 1.0.

ATSUCreateAndCopyStyle

Creates a copy of a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a valid style object to be copied. You cannot pass `NULL` for this parameter.

`oStyle` A pointer to a reference of type `ATSUStyle` (page 191). On return, `oStyle` references a style object that contains the same style run attributes, font features, and font variations set in the `iStyle` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUCreateAndCopyStyle` function creates a copy of the contents of the style object you pass in the `iStyle` parameter, including those style run attribute, font feature, and font variation values that are not set in this style object. Those style run attributes that are not set in the source style object will be set to the default values listed in Table C-1 (page 241) in the newly-created style object. Those font features and font variations that are not set in the source style object will be set to font-default values in the newly-created style object. `ATSUCreateAndCopyStyle` does not copy reference constants.

If you wish to copy the entire contents of a style object into an existing style object, call the function `ATSUCopyAttributes` (page 31). To copy style run attributes, font features, and font variations that were previously set in the source style object into the destination style object, regardless of whether or not these values are set in the destination style object, call the function `ATSUOverwriteAttributes` (page 32). To copy only those style run attributes, font features, and font variations that were previously set in the source style object but not set in the destination style object, call the function `ATSUUnderwriteAttributes` (page 34).

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 beginning with ATSUI 1.0.

ATSUCompareStyles

Compares the contents of two style objects.

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

`iFirstStyle`	A reference of type `ATSUStyle` (page 191). 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 191). 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`. On return, `oComparison` points to a comparison of the contents. See "Style Comparison Constants" (page 216) for a description of possible values. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 231).

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 one another. It 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 beginning with ATSUI 1.0.

ATSUSetStyleRefCon

Sets application-specific style object data.

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

`iStyle`	A reference of type `ATSUStyle` (page 191). 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 231).

DISCUSSION

The `ATSUSetStyleRefCon` function enables you to associate application-specific data with a style object. 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 responsible for freeing any memory allocated for the reference constant. Calling the function `ATSUDisposeStyle` (page 30) will not do so.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetStyleRefCon

Obtains application-specific style object data.

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

`iStyle` A reference of type `ATSUStyle` (page 191). 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 231).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUStyleIsEmpty

Indicates whether a style object is empty.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object. You cannot pass `NULL` for this parameter.

`oIsClear` A pointer to a value of type `Boolean`. On return, the value indicates whether the style object 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 231).

DISCUSSION

The `ATSUStyleIsEmpty` function indicates whether a style object contains any previously set style run attribute, font feature, and font variation values. If so, `ATSUStyleIsEmpty` passes back `false` in the `oIsClear` parameter, indicating that it is not an empty style object. the `ATSUStyleIsEmpty` does not consider reference constants in its evaluation.

VERSION NOTES

Available beginning 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 191). Pass a reference to a valid style object whose contents you wish to clear. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

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

If you wish to remove only the style run attributes, font features, or font variations from a style object, call the functions `ATSUClearAttributes` (page 42), `ATSUClearFontFeatures` (page 49), and `ATSUClearFontVariations` (page 54), respectively.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUDisposeStyle

Disposes of a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to the style object you want to dispose of. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

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

VERSION NOTES

Available beginning with ATSUI 1.0.

Copying Style Contents

ATSUI provides the following functions for copying style contents:

- `ATSUCopyAttributes` (page 31) copies the entire contents of a style object into an existing style object.
- `ATSUOverwriteAttributes` (page 32) copies previously set style run attributes, font features, and font variations from one style object into another.
- `ATSUUnderwriteAttributes` (page 34) copies those style run attributes, font features, and font variations that were previously set in the source style object but not set in the destination style object.

ATSUCopyAttributes

Copies the entire contents of a style object into an existing style object.

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

`iSourceStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose values you want to copy. You cannot pass `NULL` for this parameter.

`iDestinationStyle`
A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose values you want to replace. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUCopyAttributes` function copies the contents of the style object you pass in the `iSourceStyle` parameter into the destination style object, including those style run attribute, font feature, and font variation values that are not set in the source style object. Those style run attributes that are not set in the source style object will be set to the default values listed in Table C-1 (page 241) in the destination style object. Those font features and font variations that are not set in the source style object will be set to font-default values in the destination style object.

ATSUCopyAttributes does not copy the contents of memory referenced by pointers or handles within custom style run attributes or within reference constants. You are responsible for ensuring that this memory remains valid until the source style object is disposed of.

To create a copy of the entire contents of a style object, call the function ATSUCreateAndCopyStyle (page 25). To copy style run attributes, font features, and font variations that were previously set in the source style object into the destination style object, regardless of whether or not these values are set in the destination style object, call the function ATSUOverwriteAttributes (page 32). To copy only those style run attributes, font features, and font variations that were previously set in the source style object but not set in the destination style object, call the function ATSUUnderwriteAttributes (page 34).

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 beginning with ATSUI 1.0.

ATSUOverwriteAttributes

Copies previously set style run attributes, font features, and font variations from one style object into another.

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

`iSourceStyle` A reference of type ATSUStyle (page 191). Pass a reference to a valid style object whose previously set values you want to copy. You cannot pass NULL for this parameter.

`iDestinationStyle`
A reference of type ATSUStyle (page 191). Pass a reference to a valid style object whose corresponding values you want to replace. You cannot pass NULL for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUOverwriteAttributes` function copies style run attributes, font features, and font variations that were previously set in the source style object into the destination style object, regardless of whether or not these values are set in the destination style object. All other quantities in the destination style object are left unchanged.

`ATSUOverwriteAttributes` does not copy the contents of memory referenced by pointers or handles within custom style run attributes or within reference constants. You are responsible for ensuring that this memory remains valid until the source style object is disposed of.

To create a copy of the entire contents of a style object, call the function `ATSUCreateAndCopyStyle` (page 25). If you wish to copy the entire contents of a style object into an existing style object, you should call the function `ATSUCopyAttributes` (page 31). To copy only those style run attributes, font features, and font variations that were previously set in the source style object but not set in the destination style object, call the function `ATSUUnderwriteAttributes` (page 34).

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 beginning with ATSUI 1.0.

ATSUUnderwriteAttributes

Copies those style run attributes, font features, and font variations that were previously set in the source style object but not set in the destination style object.

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

`iSourceStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose previously set values you want to copy. You cannot pass `NULL` for this parameter.

`iDestinationStyle`
A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose corresponding unset values you want to replace. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUUnderwriteAttributes` function copies those style run attributes, font features, and font variations that were previously set in the source style object but not set in the destination style object. All other quantities in the destination style object are left unchanged.

`ATSUUnderwriteAttributes` does not copy the contents of memory referenced by pointers or handles within custom style run attributes or within reference constants. You are responsible for ensuring that this memory remains valid until the source style object is disposed of.

To create a copy of the entire contents of a style object, call the function `ATSUCreateAndCopyStyle` (page 25). If you wish to copy the entire contents of a style object into an existing style object, you should call the function `ATSUCopyAttributes` (page 31). To copy style run attributes, font features, and font variations that were previously set in the source style object into the destination style object, regardless of whether or not these values are set in the destination style object, call the function `ATSUOverwriteAttributes` (page 32).

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 beginning with ATSUI 1.0.

Flattening and Unflattening Style Objects

NOT RECOMMENDED

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

- `ATSUCopyToHandle` (page 35) copies styled Unicode text data from a style object to a handle.
- `ATSUPasteFromHandle` (page 36) 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);
```

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 194) when you want to provide clipboard support or copy and paste styled text between applications or within an application.

ATSUPasteFromHandle

NOT RECOMMENDED

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

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

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 194) when you want to provide clipboard support or copy and paste styled text between applications or within an application.

Manipulating Style Run Attributes

ATSUI provides the following functions for manipulating style run attributes in a style object:

- `ATSUSetAttributes` (page 37) sets style run attribute values in a style object.
- `ATSUGetAttribute` (page 38) obtains a style run attribute value from a style object.
- `ATSUGetAllAttributes` (page 40) obtains style run attribute information for a style object.
- `ATSUClearAttributes` (page 42) removes previously set style run attributes from a style object.
- `ATSUCalculateBaselineDeltas` (page 43) calculates the optimal baseline positions in a style object.

ATSUSetAttributes

Sets style run attribute values in a style object.

```
OSStatus ATSUSetAttributes (
                    ATSUStyle iStyle,
                    ItemCount iAttributeCount,
                    ATSUAttributeTag iTag[],
                    ByteCount iValueSize[],
                    ATSUAttributeValuePtr iValue[]);
```

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose style run attributes you want to set. You cannot pass `NULL` for this parameter.

`iAttributeCount`
The number of style run attributes you want to set. This value should correspond to the number of elements in the `iTag` and `iValueSize` arrays.

`iTag` An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that corresponds to the style run attribute value you wish to set. See "Style Run Attribute Tag Constants" (page 217) for a description of the Apple-defined style run attribute tag constants. If you pass a text layout attribute tag constant or an ATSUI-reserved tag constant in this parameter, `ATSUSetAttributes` returns the result code `kATSUInvalidAttributeTagErr`. You cannot pass `NULL` for this parameter.

`iValueSize` An array of values of type `ByteCount`. Each element in the array must contain the size (in bytes) of the corresponding style run attribute value being set. If you pass a size that is less than required, `ATSUSetAttributes` returns the result code `kATSUInvalidAttributeSizeErr`, and the function sets no attributes. If, after having checked all the given sizes and found them acceptable, `ATSUSetAttributes` sets style run attributes. You cannot pass `NULL` for this parameter.

`iValue` An array of pointers of type `ATSUAttributeValuePtr` (page 186). 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. 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. If you try to set the font attribute identified by the tag constant `kATSUFontTag`, 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 231).

DISCUSSION

The `ATSUSetAttributes` function sets one or more style run attribute value(s) for a style object. Unset style run attributes retain the default values listed in Table C-1 (page 241).

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 beginning with ATSUI 1.0.

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 191). Pass a reference to a valid style object whose attribute value you want to obtain. You cannot pass `NULL` for this parameter.

`iTag` An array of values of type `ATSUAttributeTag`. Pass a valid tag that corresponds to the style run attribute value you wish to determine. See "Style Run Attribute Tag Constants" (page 217) for a description of the Apple-defined style run attribute tag constants. If you pass a text layout attribute tag constant or an ATSUI-reserved tag constant in this parameter, `ATSUGetAttribute` returns the result code `kATSUInvalidAttributeTagErr`. You cannot pass `NULL` for this parameter.

`iMaximumValueSize`
The maximum size of the style run attribute value. To determine the size of an application-defined style run attribute value, see the discussion below. If you pass a size that is less than required, `ATSUGetAttribute` returns the result code `kATSUInvalidAttributeSizeErr`.

`oValue` A pointer of type `ATSUAttributeValuePtr` (page 186). Before calling `ATSUGetAttribute`, pass a pointer to memory you have allocated for the attribute value. If you are uncertain of how much memory to allocate, see the discussion below. On return, `oValue` points to the style run attribute value. If the attribute was not previously set, `ATSUGetAttribute` passes back its default value in this parameter and returns the result code `kATSUNotSetErr`.

`oActualValueSize`
A pointer to a count. On return, 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 231).

DISCUSSION

Before calling the `ATSUGetAttribute` function, call the function `ATSUGetAllAttributes` (page 40) to obtain an array of the tags and data sizes corresponding to all previously set style run attribute values in a style object. To

determine the value of a style run attribute identified by a particular style run attribute tag, you should pass the appropriate tag and data size pair passed back in the `oAttributeInfoArray` array of `ATSUGetAllAttributes` to `ATSUGetAttribute`.

If you do not know the size of the style run attribute value you wish to determine, call `ATSUGetAttribute` twice:

1. Pass a reference to the style object containing the attribute in the `iStyle` parameter, `NULL` for the `oValue` parameter, and 0 for the other parameters. `ATSUGetAttribute` returns the size of the attribute value in the `oActualValueSize` parameter.
2. Allocate enough space for a value of the returned size, then call the function again, passing a pointer in the `oValue` parameter; on return, the pointer references the attribute value.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetAllAttributes

Obtains attribute information from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose style run attribute information you want to obtain. You cannot pass `NULL` for this parameter.

`oAttributeInfoArray`
An array of structures of type `ATSUAttributeInfo` (page 185). Before calling `ATSUGetAllAttributes`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On

return, the array contains the tag and data size pairs corresponding to all previously set style run attribute values in the style object.

`iTagValuePairArraySize`
The maximum number of tags and data sizes in the style object. Typically, this is equivalent to the number of `ATSUAttributeInfo` structures in the `oAttributeInfoArray` array. To determine this value, see the discussion below.

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

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUGetAllAttributes` function obtains an array of the tags and data sizes corresponding to all previously set style run attributes in a style object. You can obtain a particular attribute value by passing the corresponding tag and data size pair to the function `ATSUGetAttribute` (page 38).

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

1. Pass a reference to the style object containing the attribute in the `iStyle` parameter, `NULL` for the `oAttributeInfoArray` parameter, and 0 for the other parameters. `ATSUGetAllAttributes` returns the size of the tag and data size arrays in the `oTagValuePairCount` parameter.
2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oAttributeInfoArray` parameter; on return, the pointer references the array of tag and data size pairs.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUClearAttributes

Removes previously set attributes from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose attributes you want to remove. You cannot pass `NULL` for this parameter.

`iTagCount` The number of attributes you want to remove. To remove all previously set attributes, pass the constant `kATSUClearAll` in this parameter. In this case, the value in the `iTag` parameter will be ignored.

`iTag` An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that identifies the style run attribute value you want to remove. See "Style Run Attribute Tag Constants" (page 217) for a description of the Apple-defined style run attribute tag constants. If you pass a text layout attribute tag constants or an ATSUI-reserved tag constant in this parameter, `ATSUClearAttributes` returns the result code `kATSUInvalidAttributeTagErr`.

function result A result code. See "Result Codes" (page 231). You can remove unset attribute values from a style object without a function error.

DISCUSSION

The `ATSUClearAttributes` function removes those previously set style run attribute values, including application-defined attributes, that are identified by tag constants in the `iTag` array. It replaces them with the default style run attribute values described in Table C-1 (page 241).

If you wish to remove all previously set style run attribute values from a style object, pass `kATSUClearAll` in the `iTagCount` parameter. To remove all previously set style run attribute, font feature, and font variation values from a style object, call the function `ATSUClearStyle` (page 29). To remove all previously set font variation values from a style object, call the function `ATSUClearFontVariations`

(page 54). To remove all previously set font feature types and selectors from a style object, call the function `ATSUClearFontFeatures` (page 49).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUCalculateBaselineDeltas

Calculates the optimal baseline positions in a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). 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 260) 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 257). On return, the array contains the distances 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 231).

DISCUSSION

You can call the `ATSUCalculateBaselineDeltas` function to calculate the distances from a specified baseline type to each of other baseline types in a specified style

object. `ATSUCalculateBaselineDeltas` takes into account font and text size when performing these calculations.

ATSUI uses these distances to determine the cross-stream shifting to apply to the glyphs in a style run. You can use the resulting array to set or obtain the optimal baseline positions of lines in a text layout object identified by the text layout attribute tag `kATSULineBaselineValuesTag`. For a description of this tag constant, see "Text Layout Attribute Tag Constants" (page 226).

VERSION NOTES

Available beginning with ATSUI 1.0.

Manipulating Font Features

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

- `ATSUSetFontFeatures` (page 44) sets font features in a style object.
- `ATSUGetFontFeature` (page 46) obtains the font feature type and selector for an indexed font feature.
- `ATSUGetAllFontFeatures` (page 47) obtains font feature information from a style object.
- `ATSUClearFontFeatures` (page 49) removes previously set font features from a style object.

ATSUSetFontFeatures

Sets font features in a style object.

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

`iStyle`	A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose font features you want to set. You cannot pass `NULL` for this parameter.
`iFeatureCount`	The number of font features you want to set. This value should correspond to the number of elements in the `iType` and `iSelector` arrays.
`iType`	An array of values of type `ATSUFontFeatureType` (page 187). Each element in the array must represent a valid feature type. You cannot pass `NULL` for this parameter.
`iSelector`	An array of values of type `ATSUFontFeatureSelector` (page 188). Each element in the array must represent a feature selector that is valid for the corresponding feature type in the `iType` parameter. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUSetFontFeatures` function sets one or more font feature(s). Unset font features retain their font-defined default values. To set a particular font feature, you must specify both the feature type and selector.

The order that `ATSUSetFontFeatures` sets font features depends on the font-defined order, not the chronological order in which they were set in a call to `ATSUSetFontFeatures`.

Prior to ATSUI 1.2, `ATSUSetFontFeature` would not remove contradictory font features. You are responsible for maintaining your own list and remove contradictory settings when they occur. Beginning with ATSUI 1.2, `ATSUSetFontFeatures` will remove contradictory font features.

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 beginning with ATSUI 1.0. Beginning with ATSUI 1.2, if you set contradictory font features, `ATSUSetFontFeatures` will remove contradictory features. Prior to ATSUI 1.2, you should maintain a list of font feature settings and removing contradictory settings when they occur.

ATSUGetFontFeature

Obtains the font feature type and selector for an indexed font feature.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose font feature value you want to obtain. You cannot pass `NULL` for this parameter.

`iFeatureIndex` A 0-based index. Pass a value between 0 and one less than the value passed back in the `oActualFeatureCount` parameter of the function `ATSUGetAllFontFeatures` (page 47).

`oFeatureType` A pointer to a value of type `ATSUFontFeatureType` (page 187) or one of the font feature types described in "Font Feature Type Constants" (page 267). On return, the font feature type corresponding to `iFeatureIndex`. If the font feature type 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 188) or one of the font feature selector constants described in "Apple Advanced Typography Constants" (page 257). On return, the font feature selector. 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 231).

DISCUSSION

Before calling the `ATSUGetFontFeature` function, call the function `ATSUGetAllFontFeatures` (page 47) to obtain an array of all the feature types and selectors that were previously set for a particular style object. To determine the font feature type and selector for a given indexed font, pass a value between 0 and one less than the value passed back in the `oActualFeatureCount` parameter of `ATSUGetAllFontFeatures` in the `iFeatureIndex` parameter of `ATSUGetFontFeature`.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetAllFontFeatures

Obtains font feature information from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose font feature information you want to obtain. You cannot pass `NULL` for this parameter.

`iMaximumFeatureCount`
The maximum number of feature types and selectors in the style object. Typically, this is equivalent to the number of feature types in the `oFeatureType` array. To determine this value, see the discussion below.

`oFeatureType` An array of values of type `ATSUFontFeatureType` (page 187) or one of the font feature types described in "Font Feature Type Constants" (page 267). Before calling `ATSUGetAllFontFeatures`, pass a pointer to memory that you have allocated for this array.

If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains the font feature types that have been set in the style object.

`oFeatureSelector`

An array of values of type `ATSUFontFeatureSelector` (page 188) or one of the font feature selector constants described in "Apple Advanced Typography Constants" (page 257). Before calling `ATSUGetAllFontFeatures`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains the font feature selectors that have been set in the style object.

`oActualFeatureCount`

A pointer to a count. On return, the actual number of font feature types and selectors 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 231).

DISCUSSION

The `ATSUGetAllFontFeatures` function obtains an array of the feature type and selector pairs corresponding to all previously set font features in a style object. You can use the value passed back in the `oActualFeatureCount` parameter to calculate the maximum font feature index to pass in the `iFeatureIndex` parameter of the function `ATSUGetFontFeature` (page 46).

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

1. Pass a reference to the style object containing the font feature in the `iStyle` parameter, `NULL` for the `oFeatureType` and `oFeatureSelector` parameters, and 0 for the other parameters. `ATSUGetAllFontFeatures` returns the sizes of the feature type and selector arrays in the `oActualFeatureCount` parameter.
2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oFeatureType` and `oFeatureSelector` parameters; on return, the pointers references arrays of feature types and selectors, respectively.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUClearFontFeatures

Removes previously set font features from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose font features you want to remove. You cannot pass `NULL` for this parameter.

`iFeatureCount` The number of font features you want to remove. To remove all previously set font features, pass the constant `kATSUClearAll` in this parameter. In this case, the values in the `iType` and `iSelector` parameters will be ignored.

`iType` An array of values of type `ATSUFontFeatureType` (page 187). Each element in the array must contain a valid feature type that identifies a font feature you want to remove.

`iSelector` An array of values of type `ATSUFontFeatureSelector` (page 188). Each element in the array must contain a valid feature selector that specifies the setting for the font feature that you want to remove.

function result A result code. See "Result Codes" (page 231). You can remove unset font feature values from a style object without a function error.

DISCUSSION

The `ATSUClearFontFeatures` function removes those previously set font features that are identified by the feature selector and type constants in the `iSelector` and `iType` arrays. It replaces them with their font-defined default values.

If you wish to remove all previously set font features from a style object, pass `kATSUClearAll` in the `iFeatureCount` parameter. To remove all previously set style run attribute, font feature, and font variation values from a style object, call the function `ATSUClearStyle` (page 29). To remove just the previously set font variation values from a style object, call the function `ATSUClearFontVariations` (page 54).

VERSION NOTES

Available beginning with ATSUI 1.0.

Manipulating Font Variations

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

- `ATSUSetVariations` (page 50) sets font variations in a style object.
- `ATSUGetFontVariationValue` (page 51) obtains the font variation value of a font variation axis.
- `ATSUGetAllFontVariations` (page 52) obtains font variation information from a style object.
- `ATSUClearFontVariations` (page 54) removes previously set font variations from a style object.

ATSUSetVariations

Sets font variations in a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose font variations you want to set. You cannot pass `NULL` for this parameter.

`iVariationCount`
: The number of font variations you want to set. This value should correspond to the number of elements in the `iAxes` and `iValue` arrays.

`iAxes`
: An array of values of type `ATSUFontVariationAxis` (page 188). Each element in the array must represent a valid variation axis. You cannot pass `NULL` for this parameter.

`iValue`
: An array of values of type `ATSUFontVariationValue` (page 189). Each element in the array must contain a value that is valid for the corresponding variation axis in the `iAxes` parameter. You cannot pass `NULL` for this parameter.

function result
: A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUSetVariations` function sets one or more font variation (s). Unset font variations retain their font-defined default values. If the font does not support the specified variation axis, 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 beginning with ATSUI 1.0.

ATSUGetFontVariationValue

Obtains the font variation value of a font variation axis.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose font variation value you want to obtain. You cannot pass `NULL` for this parameter.

`iATSUFontVariationAxis`
A value of type `ATSUFontVariationAxis` (page 188). Pass the font variation axis whose value you want to obtain.

`oATSUFontVariationValue`
A pointer to a value of type `ATSUFontVariationValue` (page 189). On return, the value corresponding to the font variation axis. If this value has not been set, `ATSUGetFontVariationValue` passes back the font-defined default value and returns the result code `kATSUNotSetErr`. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

Before calling `ATSUGetFontVariationValue`, call the function `ATSUGetAllFontVariations` (page 52) to obtain an array of variation axes corresponding to all previously set font variations. You can then call `ATSUGetFontVariationValue` with the appropriate variation axis to determine the corresponding variation value.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetAllFontVariations

Obtains font variation information from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to the style object whose font variation information you want to obtain. You cannot pass `NULL` for this parameter.

`iVariationCount`
The maximum number of font variations in the style object. Typically, this is equivalent to the number of font variation axes in the `ATSUFontVariationAxis` array. To determine this value, see the discussion below.

`oVariationAxes`
An array of values of type `ATSUFontVariationAxis` (page 188). Before calling `ATSUGetAllFontVariations`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains the previously set font variation axes in the style object.

`oATSUFontVariationValues`
An array of values of type `ATSUFontVariationValue` (page 189). Before calling `ATSUGetAllFontVariations`, pass a pointer to memory that you have allocated for this array. If you are uncertain about how much memory to allocate, see the discussion below. On return, the array contains the font variation values that correspond to the font variation axes passed back in the `oVariationAxes` array.

`oActualVariationCount`
A pointer to a count. On return, 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 231).

DISCUSSION

The `ATSUGetAllFontVariations` function obtains an array of the variation axes and values corresponding to all previously set font variations in a style object. You can obtain a particular variation value by passing its corresponding variation axis to the function `ATSUGetFontVariationValue` (page 51).

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

1. Pass a reference to the style object containing the font variation in the `iStyle` parameter, `NULL` for the `oVariationAxes` and `oATSUFontVariationValues` parameters, and 0 for the other parameters. `ATSUGetAllFontVariations` returns the size of the axes and value arrays in the `oActualVariationCount` parameter.
2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oVariationAxes` and `oATSUFontVariationValues` parameters; on return, the pointers reference arrays of variation axes and values, respectively.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUClearFontVariations

Removes previously set font variations from a style object.

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

`iStyle` A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose font variations you want to remove. You cannot pass `NULL` for this parameter.

`iAxisCount` The number of font variations you want to remove. To remove all previously set font variations, pass the constant `kATSUClearAll` in this parameter. In this case, the value in the `iAxes` parameter will be ignored.

`iAxes` An array of values of type `ATSUFontVariationAxis` (page 188). Each element in the array must contain a valid variation axis that corresponds to the variation value you want to remove.

function result A result code. See "Result Codes" (page 231). You can remove unset font variation values from a style object without a function error.

DISCUSSION

The `ATSUClearFontVariations` function removes those previously set font variations identified in the `iAxes` array. It replaces them with their font-defined default values.

If you wish to remove all previously set font variations from a style object, pass `kATSUClearAll` in the `iAxisCount` parameter. To remove all previously set style run attribute, font feature, and font variation values from a style object, call the function `ATSUClearStyle` (page 29). To remove just the previously set font features from a style object, call the function `ATSUClearFontFeatures` (page 49).

VERSION NOTES

Available beginning with ATSUI 1.0.

Finding Compatible Fonts

ATSUI provides the following functions for finding fonts that are compatible with ATSUI:

- `ATSUFontCount` (page 55) counts the number of ATSUI-compatible fonts installed on a user's system.
- `ATSUGetFontIDs` (page 56) obtains the IDs of all ATSUI-compatible fonts installed on a user's system.
- `ATSUFindFontFromName` (page 57) finds the ID of the first font in a name table with a particular font name code, language, platform, and script.

ATSUFontCount

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

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

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

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUFontCount` function only counts the number of 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 can use this count to determine the amount of memory to allocate for the `oFontIDs` array in the function `ATSUGetFontIDs` (page 56).

The number of available fonts may change while your application is running. Although 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 possible for a font to be added and another removed between two successive calls of `ATSUFontCount`, leaving the total number unchanged.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetFontIDs

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

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

`oFontIDs` An array of values of type `ATSUFontID` (page 188). Before calling `ATSUGetFontIDs`, pass a pointer to memory that you have allocated for this array. You can determine the amount of memory to allocate by using the count returned from the function `ATSUFontCount` (page 55). On return, the array contains the IDs of all ATSUI-compatible fonts installed on the system.

`iArraySize` The maximum number of fonts in the style object. Typically, this is equivalent to the number of IDs in the `oFontIDs` array. Call `ATSUFontCount` (page 55) to determine this value.

`oFontCount`	A pointer to a count. On return, the actual number of ATSUI-compatible fonts that are installed on the user's system. This may be greater than the value passed in the `iArraySize` parameter. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUGetFontIDs` function obtains the IDs of all ATSUI-compatible fonts that are installed on the user's system. For a description of fonts that are not compatible with ATSUI, see the discussion for `ATSUFontCount` (page 55). You should call `ATSUGetFontIDs` to rebuild your font menu when your application is brought to the foreground.

Before calling `ATSUGetFontIDs`, call `ATSUFontCount` to determine the number of ATSUI-compatible fonts installed on the user's system. You should then allocate enough memory to contain this number of fonts for the array passed back in the `oFontIDs` parameter.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUFindFontFromName

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

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

`iName`	A pointer to a buffer. Pass the name string of the font you want to obtain.

`iNameLength` The length (in bytes) of the font name string.

`iFontNameCode`

A value of type `FontNameCode`. Pass the type of the font name string. See "Font Name Code Constants" (page 271) for a description of possible values.

`iFontNamePlatform`

A value of type `FontPlatformCode`. Pass the encoding of the font name string. See "Font Name Platform Constants" (page 284) for a description of possible values. You can pass the `kFontNoPlatform` constant if you do not care about the encoding. In this case, `ATSUFindFontFromName` will pass back the first font in the name table matching the other font name parameters.

`iFontNameScript`

A value of type `FontScriptCode`. Pass the script ID of the font name string. Depending upon the font name platform, see "Macintosh Platform Script Code Constants" (page 295), "Microsoft Platform Script Code Constants" (page 301), or "Unicode Platform Script Code Constants" (page 311) for a description of possible values. You can pass the `kFontNoScript` constant if you do not care about the script ID. In this case, `ATSUfindFontFromName` will pass back the first font in the name table matching the other font name parameters.

`iFontNameLanguage`

A value of type `FontLanguageCode`. Pass the language of the font name string. See "Font Name Language Constants" (page 274) for a description of possible values. You can pass the `kFontNoLanguage` constant if you do not care about the language. In this case, `ATSUFindFontFromName` will pass back the first font in the name table matching the other font name parameters.

`oFontID` A pointer to a value of type `ATSUFontID` (page 188). On return, the first font that matches the specified name code, platform, script, and language. If no installed font matches these parameters, `ATSUFindFontFromName` passes back the constant `kATSUInvalidFontID` and returns the result code `kATSUInvalidFontErr`.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

Because ATSUI cannot guarantee the uniqueness of names among installed fonts, the `ATSUFindFontFromName` function finds the first (but not necessarily the only) font that matches the specified name, platform, language, and script. If you want to find, for an indexed font name, the font name and information about the name like type, platform, script ID, and language, call the function `ATSUGetIndFontName` (page 61). If you want to find the ID of the first font in a font name table with a particular font name code, language, platform, and script, call the function `ATSUFindFontName` (page 63).

`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 beginning with ATSUI 1.0.

Searching a Font Name Table

ATSUI provides the following functions for searching a font name table:

- `ATSUCountFontNames` (page 60) counts the number of font names in a font.
- `ATSUGetIndFontName` (page 61) obtains, for an indexed font name, the font name and information about the name like type, platform, script ID, and language.
- `ATSUFindFontName` (page 63) obtains the index and font name of the first font in a name table with a particular font name code, language, platform, and script.

ATSUCountFontNames

Counts the number of font names in a font.

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

`iFont` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose names you want to count.

`oFontNameCount` A pointer to a count. On return, the total number of entries in the font name table. This count includes the names of font features, variations, tracking settings, and instances, as well as font names identified by name code constants that aren't enumerated. 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 231).

DISCUSSION

The `ATSUCountFontNames` function obtains the total number of font names defined in a font 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 pass one less than this count in the `iFontNameIndex` parameter of the function `ATSUGetIndFontName` (page 61) to iterate through the entries of a font name table.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetIndFontName

Obtains, for an indexed font name, the font name and information about the name like type, platform, script ID, and language.

```
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 188). Pass the ID of the font whose indexed font name you want information about.

`iFontNameIndex`
A 0-based index. Pass a value between 0 and one less than the count passed back by the function `ATSUCountFontNames` (page 60).

`iMaximumNameLength`
The maximum length of the font name. Typically, this is equivalent to the size of the buffer allocated to contain the font name pointed to by the `oName` parameter. To determine this length, see the discussion below.

`oName` A pointer to a buffer. Before calling `ATSUGetIndFontName`, pass a pointer to memory that you have allocated for this buffer. If you are uncertain of how much memory to allocate, see the discussion below. On return, the buffer contains the font name string. If the buffer you allocate is not large enough, `ATSUGetIndFontName` passes back a partial string. You cannot pass `NULL` for this parameter.

`oActualNameLength`
A pointer to a count. On return, the actual length of the font name string. This may be greater than the value passed in the `iMaximumNameLength` parameter. You should check this value to make sure that you allocated enough memory for the buffer. You cannot pass `NULL` for this parameter.

`oFontNameCode` A pointer to a value of type `FontNameCode`. On return, the type of the font name string. See "Font Name Code Constants" (page 271) for a description of possible values.

`oFontNamePlatform`
A pointer to a value of type `FontPlatformCode`. On return, the encoding of the font name string. See "Font Name Platform Constants" (page 284) for a description of possible values.

`oFontNameScript`
A pointer to a value of type `FontScriptCode`. On return, the script ID of the font name string. Depending upon the font name platform, see "Macintosh Platform Script Code Constants" (page 295), "Microsoft Platform Script Code Constants" (page 301), or "Unicode Platform Script Code Constants" (page 311) for a description of possible values.

`oFontNameLanguage`
A pointer to a value of type `FontLanguageCode`. On return, the language of the font name string. See "Font Name Language Constants" (page 274) 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 231).

DISCUSSION

You should call the `ATSUGetIndFontName` to iterate through the entries of a font name table. If you want to find the index and name of the first font in a name table with a particular font name code, language, platform, and script, call the function `ATSUFindFontFromName` (page 57). If you want to find the ID of the first font in a font name table with a particular font name code, language, platform, and script, call the function `ATSUFindFontName` (page 63).

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

1. Pass the ID of the font whose name table you want to iterate in the `iFontID` parameter, `NULL` for the `oName` parameter, and 0 for the other parameters.

`ATSUGetIndFontName` returns the length of the font name string in the `oActualNameLength` parameter.

2. Allocate enough space for a font name buffer of the returned size, then call the function again, passing a pointer in the `oName` parameter; on return, the pointer references the font name string.

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 beginning with ATSUI 1.0.

ATSUFindFontName

Obtains the index and font name of the first font in a name table with a particular font name code, language, platform, and script.

```
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 188). Pass the ID of the font whose particular font name you are searching for.

`iFontNameCode`

A value of type `FontNameCode`. Pass the type of the font name string you are searching for. See "Font Name Code Constants" (page 271) for a description of possible values.

`iFontNamePlatform`

A value of type `FontPlatformCode`. Pass the encoding of the font name string you are searching for. See "Font Name Platform Constants" (page 284) for a description of possible values. You can pass the `kFontNoPlatform` constant if you do not care about the encoding of a font name. In this case, `ATSUFindFontName` will pass back the first font in the name table matching the other font name parameters.

`iFontNameScript`

A value of type `FontScriptCode`. Pass the script ID of the font name string. Depending upon the font name platform, see "Macintosh Platform Script Code Constants" (page 295), "Microsoft Platform Script Code Constants" (page 301), or "Unicode Platform Script Code Constants" (page 311) for a description of possible values. You can pass the `kFontNoScript` constant if you do not care about the script ID. In this case, `ATSUFindFontName` will pass back the first font in the name table matching the other font name parameters.

`iFontNameLanguage`

A value of type `FontLanguageCode`. Pass the language of the font name string you are searching for. See "Font Name Language Constants" (page 274) for a description of possible values. You can pass the `kFontNoLanguage` constant if you do not care about the language of the font name. In this case, `ATSUFindFontName` will pass back the first font in the name table matching the other font name parameters.

`iMaximumNameLength`

The maximum length of the font name. Typically, this is equivalent to the size of the buffer allocated to contain the font name pointed to by the `oName` parameter. To determine this length, see the discussion below.

`oName` A pointer to a buffer. Before calling `ATSUFindFontName`, pass a pointer to memory that you have allocated for this buffer. On return, the buffer contains the font name string. If the buffer you allocate is not large enough, `ATSUFindFontName` passes back a partial string. You cannot pass `NULL` for this parameter.

`oActualNameLength`
: A pointer to a count. On return, the actual length of the font name string. This may be greater than the value passed in the `iMaximumNameLength` parameter. You should check this value to make sure that you allocated enough memory for the buffer. You cannot pass `NULL` for this parameter.

`oFontNameIndex`
: A pointer to a count. On return, the 0-based index of the font name in the font name table.

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 231).

DISCUSSION

The `ATSUFindFontName` function obtains the index and font name of the first font in a name table with a particular font name code, language, platform, and script. If you want to find the index and name of the first font in a name table with a particular font name code, language, platform, and script, call the function `ATSUFindFontFromName` (page 57). If you want to find, for an indexed font name, the font name and information about the name like type, platform, script ID, and language, call the function `ATSUGetIndFontName` (page 61).

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

1. Pass the ID of the font whose name table you are searching in the `iFontID` parameter, `NULL` for the `oName` parameter, and 0 for the other parameters. `ATSUFindFontName` returns the length of the font name string in the `oActualNameLength` parameter.
2. Allocate enough space for a font name buffer of the returned size, then call the function again, passing a pointer in the `oName` parameter; on return, the pointer references the font name string.

VERSION NOTES

Available beginning with ATSUI 1.0.

Converting Font IDs and Font Family Numbers

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

- `ATSUFONDtoFontID` (page 66) finds the ATSUI font ID corresponding to a font family number.
- `ATSUFontIDtoFOND` (page 67) finds the font family number corresponding to an ATSUI font ID.

ATSUFONDtoFontID

Finds the ATSUI font ID corresponding to a font family number.

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

`iFONDNumber` The font family number of the font whose ID you want to obtain.

`iFONDStyle` A value of type `Style`. Pass 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 188). On return, the ID corresponding to the specified font family number. If there are no installed fonts with the matching the specified font family number, `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 it is incompatible 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 231).

DISCUSSION

Font family numbers were used by QuickDraw to represent fonts to the Font Manager. Some of these fonts do not have font IDs (even if they are compatible with ATSUI). For a list of fonts that are not compatible with ATSUI, see the discussion for `ATSUFontCount` (page 55).

ATSUI assigns font IDs at run-time. As a result, font IDs can change across installs.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUFontIDtoFOND

Finds the font family number corresponding to an ATSUI font ID.

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

`iFontID` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose font family number you want to obtain.

`oFONDNumber` A pointer to a value of type `short`. On return, the font family number corresponding to the specified font ID. You cannot pass `NULL` for this parameter.

`oFONDStyle` A pointer to a value of type `Style`. On return, the font family style, if it exists, corresponding to the specified font ID. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231). If the font does not correspond to an installed font, `ATSUFontIDtoFOND` passes back `kATSUInvalidFontID` in the `oFONDNumber` parameter and returns the result code `kATSUInvalidFontErr`. If you the font is incompatible with ATSUI, `ATSUFONDtoFontID` passes back `kATSUInvalidFontID` in the `oFONDNumber` parameter and returns the result code `kATSUNoCorrespondingFontErr`.

DISCUSSION

Font family numbers were used by QuickDraw to represent fonts to the Font Manager. Some fonts may correspond to a font ID, but not be compatible with ATSUI. For a list of fonts that are not compatible with ATSUI, see the discussion for `ATSUFontCount` (page 55).

ATSUI assigns font IDs at run-time. As a result, font IDs can change across installs.

VERSION NOTES

Available beginning with ATSUI 1.0.

Obtaining Font Tracking Information

ATSUI provides the following functions for obtaining font tracking information:

- `ATSUCountFontTracking` (page 68) counts the number of font trackings in a font.
- `ATSUGetIndFontTracking` (page 69) obtains information about the name code and values of a font tracking in a font.

ATSUCountFontTracking

Counts the number of font trackings in a font.

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

`iFontID` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose font trackings you want to count.

`iCharacterOrientation`
A value of type `ATSUVerticalCharacterType`. Pass the glyph orientation of the font tracking you want to count. See "Glyph Orientation Constants" (page 207) for a description of possible values. You must specify this value because there are different font trackings for different glyph orientations.

oTrackingCount A pointer to a count. On return, the number of font trackings 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 231).

DISCUSSION

The ATSUCountFontTracking function obtains the total number of font trackings defined in a font. You can pass one less than this count in the iTrackIndex parameter of the function ATSUGetIndFontTracking (page 69).

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUGetIndFontTracking

Obtains information about the name code and values of a font tracking in a font.

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

iFontID A value of type ATSUFontID (page 188). Pass the ID of the font whose font tracking you want to get name information for.

iCharacterOrientation
A value of type ATSUVerticalCharacterType. Pass the glyph orientation of the font tracking whose value you want to obtain. See "Glyph Orientation Constants" (page 207) for a description of possible values. You must specify this value because there are different font trackings for different glyph orientations.

`iTrackIndex`	A 0-based index. Pass a value between 0 and one less than the count passed back in the function `ATSUCountFontTracking` (page 68).
`oFontTrackingValue`	A pointer to a `Fixed` value. On return, the font tracking value corresponding to the specified index and character orientation.
`oNameCode`	A pointer to a value of type `FontNameCode`. On return, the type of font tracking name. See "Font Name Code Constants" (page 271) for a description of possible values. 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 231).

DISCUSSION

The `ATSUGetIndFontTracking` function obtains, for an indexed font tracking, its value and name code. You can pass this value to the function `ATSUFindFontName` (page 63) to find the localized font tracking name identified by this name code. You can call `ATSUGetIndFontTracking` to iterate through the entries of a font tracking table.

VERSION NOTES

Available beginning with ATSUI 1.1.

Obtaining Font Feature Information

ATSUI provides the following functions for obtaining font feature information:

- `ATSUCountFontFeatureTypes` (page 71) counts the number of feature types in a font.
- `ATSUGetFontFeatureTypes` (page 72) obtains a list of the available feature types in a font.
- `ATSUCountFontFeatureSelectors` (page 73) counts the number of feature selectors for a given feature type in a font.

- `ATSUGetFontFeatureSelectors` (page 74) obtains a list of the feature selectors for a given feature type in a font.
- `ATSUGetFontFeatureNameCode` (page 76) obtains information about the name code of a feature type or selector in a font feature.

ATSUCountFontFeatureTypes

Counts the number of feature types in a font.

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

`iFontID`	A value of type `ATSUFontID` (page 188). Pass the ID of the font whose feature types you want to count.
`oTypeCount`	A pointer to a count. On return, the number of feature types 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 231).

DISCUSSION

The `ATSUCountFontFeatureTypes` function counts the total number of feature types defined in a font. You can use this count to determine how much memory to allocate for the `oTypes` array in the function `ATSUGetFontFeatureTypes` (page 72).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetFontFeatureTypes

Obtains a list of the available feature types in a font.

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

`iFont` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose feature information you want to obtain.

`iMaximumTypes` The maximum number of feature types in the font. Typically, this is equivalent to the number of elements in the `oTypes` array. To determine this value, see the discussion below.

`oTypes` An array of values of type `ATSUFontFeatureType` (page 187). Before calling `ATSUGetFontFeatureTypes`, pass a pointer to memory that you have allocated for this array. On return, the array contains a list of the feature types defined in the font. If you are of how much memory to allocate, see the discussion below. You cannot pass `NULL` for this parameter.

`oActualTypeCount`
A pointer to a count. On return, the actual number of feature types defined 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 231).

DISCUSSION

The `ATSUGetFontFeatureTypes` function obtains a list of the feature types that are available in a font.

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

1. Pass the ID of the font whose feature types you want to obtain in the `iFont` parameter, `NULL` for the `oTypes` parameter, and 0 for the other parameters.

`ATSUGetFontFeatureTypes` returns the size of the feature type array in the `oActualTypeCount` parameter.

2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oTypes` parameter; on return, the pointer references the array of feature types.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUCountFontFeatureSelectors

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

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

`iFont` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose feature selectors you want to count.

`iType` A value of type `ATSUFontFeatureType` (page 187). Pass a valid feature type whose feature selectors you want to count.

`oSelectorCount` A pointer to a count. On return, the number of feature selectors defined for the feature type 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 231).

DISCUSSION

The `ATSUCountFontFeatureSelectors` function counts the number of feature selectors that are defined for a given feature type. You can use this count to determine how much memory to allocate for the `oSelectors` array in the function `ATSUGetFontFeatureSelectors` (page 74).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetFontFeatureSelectors

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

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

`iFont` A value of type `ATSUFontID` (page 188). Pass the ID of the font for whose feature type you want to count the number of feature selectors.

`iType` A value of type `ATSUFontFeatureType` (page 187). Pass a valid feature type whose font selectors you want to determine.

`iMaximumSelectors`
The maximum number of feature selectors in the font. Typically, this is equivalent to the number of elements in the `oSelectors` array. To determine this value, see the discussion below.

`oSelectors` An array of values of type `ATSUFontFeatureSelector` (page 188) or one of the feature selector constants described in "Apple Advanced Typography Constants" (page 257). Before calling `ATSUGetFontFeatureSelectors`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains a list of all the feature selectors available for a given feature type. You cannot pass `NULL` for this parameter.

`oSelectorIsOnByDefault`
An array of `Boolean` values. Before calling `ATSUGetFontFeatureSelectors`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return,

each element in the array indicates whether the corresponding feature selector is on. If `true`, the feature selector is on by default. You cannot pass `NULL` for this parameter.

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

`oIsMutuallyExclusive`
A pointer to `Boolean` value. On return, the value indicates whether more than one font feature selector can be on at once. 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 231).

DISCUSSION

The `ATSUGetFontFeatureSelectors` function obtains a list of the feature selectors that are defined for a given feature type in a font. You can use this information to set the font features and selectors in a style object.

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

1. Pass the ID of the font whose feature selectors you want to determine in the `iFont` parameter, `NULL` for the `oSelectors` parameter, and 0 for the other parameters. `ATSUGetFontFeatureSelectors` returns the size of the feature type array in the `oActualSelectorCount` parameter.
2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oSelectors` parameter; on return, the pointer references the array of feature selectors for the given feature type.
3. You can also determine how much memory to allocate for the `oSelectors` array by calling the function `ATSUCountFontFeatureSelectors` (page 73), which will return the total number of font feature selectors in a particular feature type.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetFontFeatureNameCode

Obtains information about the name code of a feature type or selector in a font feature.

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

`iFont` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose feature type or selector you want to determine the name code for.

`iType` A value of type `ATSUFontFeatureType` (page 187). Pass a valid feature type whose name code you want to obtain.

`iSelector` A value of type `ATSUFontFeatureSelector` (page 188). Pass the feature selector whose name code you want to obtain. If you pass the constant `kATSUNoSelector`, the value passed back in the `oNameCode` parameter represents the name code of the feature type itself.

`oNameCode` A pointer to a value of type `FontNameCode`. On return, the type of the feature type or selector name, depending upon the value passed in the `iSelector` parameter. See "Font Name Code Constants" (page 271) for a description of possible values. 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 231).

DISCUSSION

By default, the `ATSUGetFontFeatureNameCode` function obtains the name code of the specified feature selector. If you pass the constant `kATSUNoSelector` in the `iSelector` parameter, `ATSUGetFontFeatureNameCode` obtains the name code of the feature type. You can pass this value to the function `ATSUFindFontName` (page 63) to find the localized feature selector or feature type name identified by this name code.

VERSION NOTES

Available beginning with ATSUI 1.0.

Obtaining Font Variation Data

ATSUI provides the following functions for obtaining font variation data:

- `ATSUCountFontVariations` (page 77) determines the number of font variations in a font.
- `ATSUGetIndFontVariation` (page 78) obtains information about a font variation.
- `ATSUGetFontVariationNameCode` (page 80) obtains information about the name code of a font variation in a font.

ATSUCountFontVariations

Determines the number of font variations in a font.

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

`iFont` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose variations you want to count.

`oVariationCount` A pointer to a count. On return, the number of font variations defined 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 231).

DISCUSSION

The `ATSUCountFontVariations` function determines the number of font variations that are defined in a font.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetIndFontVariation

Obtains information about a font variation.

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

`iFontID` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose font variation you want information about.

`iVariationIndex`
A 0-based index. Pass a value between 0 and one less than the count passed back in the function `ATSUCountFontVariations` (page 77).

`oATSUFontVariationAxis`
A pointer to a value of type `ATSUFontVariationAxis` (page 188). On return, the axis of the font variation. You cannot pass `NULL` for this parameter.

`oMinimumValue`
A pointer to a value of type `ATSUFontVariationValue` (page 189). On return, the minimum value of the variation axis. You cannot pass `NULL` for this parameter.

`oMaximumValue`
A pointer to a value of type `ATSUFontVariationValue` (page 189). On return, the maximum value of the variation axis. You cannot pass `NULL` for this parameter.

`oDefaultValue` A pointer to a value of type `ATSUFontVariationValue` (page 189). On return, the default value of the variation axis. 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 231).

DISCUSSION

The `ATSUGetIndFontVariation` function obtains, for an indexed font variation, its axis and the default, minimum, and maximum values of that axis. Some fonts are capable of generating a wide range of stylistic changes. Such a font contains font variation axes, each of which describes a particular stylistic attribute and the range of values that the font can use. Each axis has a minimum, maximum, and default value. The minimum and maximum values determine the range of values that the variation axis covers. A font may also name specific values along a variation axis as font instances.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetFontVariationNameCode

Obtains information about the name code of a font variation in a font.

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

`iFont`	A value of type `ATSUFontID` (page 188). Pass the ID of the font whose variation you want to get name information for.
`iAxis`	A value of type `ATSUFontVariationAxis` (page 188). Pass a valid variation whose name code you want to obtain.
`oNameCode`	A pointer to a value of type `FontNameCode`. On return, the type of the font variation name. See "Font Name Code Constants" (page 271) for a description of possible values. 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 231).

DISCUSSION

The `ATSUGetIndFontVariation` function obtains the name code of the specified font variation. You can pass this value to the function `ATSUFindFontName` (page 63) to find the localized font variation name identified by this name code.

VERSION NOTES

Available beginning with ATSUI 1.0.

Obtaining Font Instance Data

ATSUI provides the following functions for obtaining font instance data:

- `ATSUCountFontInstances` (page 81) counts the number of font instances in a font.

- `ATSUGetFontInstance` (page 82) obtains information about a font instance.
- `ATSUGetFontInstanceNameCode` (page 83) obtains information about the name code of a font instance in a font.

ATSUCountFontInstances

Counts the number of font instances in a font.

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

`iFont`	A value of type `ATSUFontID` (page 188). Pass the ID of the font whose font instances you want to count.
`oInstances`	A pointer to a count. On return, the number of font instances defined 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 231).

DISCUSSION

The `ATSUCountFontInstances` function determines the number of font instances that are defined in a font. You can pass one less than this count in the `iFontInstanceIndex` parameter of the function `ATSUGetFontInstance` (page 82).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetFontInstance

Obtains information about a font instance.

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

`iFontID` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose font instance you want information about.

`iFontInstanceIndex`
A 0-based index. Pass a value between 0 and one less than the count passed back in the function `ATSUCountFontInstances` (page 81).

`iMaximumVariations`
The maximum number of font variations. Typically, this is equivalent to the number of elements in the `oAxes` and `oValues` arrays. To determine this value, see the discussion below.

`oAxes` An array of values of type `ATSUFontVariationAxis` (page 188). Before calling `ATSUGetFontInstance`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains a list of the font variation axes in the font. You cannot pass `NULL` for this parameter.

`oValues` An array of values of type `ATSUFontVariationValue` (page 189). Before calling `ATSUGetFontInstance`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains a list of the font variation axes in the font. You cannot pass `NULL` for this parameter.

`oActualVariationCount`
A pointer to a count. On return, the actual number of font variations. 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 231).

DISCUSSION

The `ATSUGetFontInstance` function obtains, for an indexed font instance, the corresponding axis and value.

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

1. Pass the ID of the font whose font instance you want information about in the `iFont` parameter, `NULL` for the `oAxes` and `oValues` parameters, and 0 for the other parameters. `ATSUGetFontInstance` returns the size of the `oAxes` and `oValues` arrays in the `oActualVariationCount` parameter.
2. Allocate enough space for an array of the returned size, then call the function again, passing pointers in the `oAxes` and `oValues` parameters; on return, the pointers reference the array of axes and values corresponding to the font instances in the font.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetFontInstanceNameCode

Obtains information about the name code of a font instance in a font.

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

`iFont` A value of type `ATSUFontID` (page 188). Pass the ID of the font whose font instance you want to get name information for.

`iFontInstanceIndex`
A 0-based index. Pass a value between 0 and one less than the count passed back in the function `ATSUCountFontInstances` (page 81).

`oNameCode` A pointer to a value of type `FontNameCode`. On return, the type of the font instance name. See "Font Name Code Constants" (page 271) for a description of possible values. 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 231).

DISCUSSION

The `ATSUGetFontInstanceNameCode` function obtains the name code of an indexed font instance. You can pass this value to the function `ATSUFindFontName` (page 63) to find the localized font instance name identified by this name code.

VERSION NOTES

Available beginning with ATSUI 1.0.

Creating and Disposing of Text Layout Objects

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

- `ATSUCreateTextLayout` (page 85) creates an uninitialized text layout object.
- `ATSUCreateTextLayoutWithTextPtr` (page 86) creates a text layout object with style runs, a pointer to associated text, and default text layout attribute values.
- `ATSUCreateTextLayoutWithTextHandle` (page 88) creates a text layout object with style runs, a handle to associated text, and default text layout attribute values.
- `ATSUCreateAndCopyTextLayout` (page 91) creates a copy of a text layout object.

- ATSUSetTextLayoutRefCon (page 92) sets application-specific text layout object data.
- ATSUGetTextLayoutRefCon (page 92) obtains application-specific text layout object data.
- ATSUClearLayoutCache (page 93) clears the layout cache of a line or an entire text layout object.
- ATSUDisposeTextLayout (page 94) disposes of a text layout object.

ATSUCreateTextLayout

Creates an uninitialized text layout object.

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

oTextLayout	A pointer to a reference of type ATSUTextLayout (page 191). On return, the newly-created text layout object.
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

The ATSUCreateTextLayout function creates an empty text layout object that contains default text layout attributes but no style runs, soft line breaks, or associated text. The default text layout attribute values are described in Table C-2 (page 245). You can set non-default text layout attribute values by calling the function ATSUSetLayoutControls (page 96).

To create a text layout object that contains style runs, text, and soft line breaks, call the functions ATSUCreateTextLayoutWithTextHandle (page 88) and ATSUCreateTextLayoutWithTextPtr (page 86).

Most functions that operate on text layout objects require that they have style runs, text, and soft line breaks. To assign style runs, text, and soft line breaks to an "empty" text layout object, call the functions ATSUSetRunStyle (page 118), ATSUSetTextPointerLocation (page 111) or ATSUSetTextHandleLocation (page 113), and ATSUSetSoftLineBreak (page 159) or ATSUBreakLine (page 157).

To create a copy of an existing text layout object, call the function ATSUCreateAndCopyTextLayout (page 91).

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 beginning with ATSUI 1.0.

ATSUCreateTextLayoutWithTextPtr

Creates a text layout object with style runs, a pointer to associated text, and default text layout attribute values.

```
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 192). Pass a pointer to the beginning of a text buffer. Note that ATSUI expects the buffer to contain Unicode text. Your application is responsible for allocating the memory associated with this pointer.

`iTextOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset in backing store memory that corresponds to the beginning of range of text that you want ATSUI to perform layout operations on. If you want the range of text to start at the beginning of the text buffer, you should pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If you want the range of text to cover the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iTextLength` parameter. If the specified

range of text is outside the text buffer, `ATSUCreateTextLayoutWithTextPtr` returns the result code `kATSUInvalidTextRangeErr`.

`iTextLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text that you want ATSUI to perform layout operations on. If you want the range of text to span the end of the text buffer, you should pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). If you want the range of text to cover the entire text buffer, pass `kATSUToTextEnd` in this parameter and `kATSUFromTextBeginning` in the `iTextOffset` parameter. If the specified range of text is outside the text buffer, `ATSUCreateTextLayoutWithTextPtr` returns the result code `kATSUInvalidTextRangeErr`.

`iTextTotalLength`
A value of type `UniCharCount` (page 194). Pass 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` The number of style runs to assign to the text layout object. This should be equivalent to the number of elements in the `iRunLengths` and `iStyles` arrays.

`iRunLengths` An array of values of type `UniCharCount` (page 194). Each element in the array must contain a style run length that corresponds to a style object in the `iStyles` array. You can pass `kATSUToTextEnd` for the last style run length if you want it to extend to end of the text buffer. If the sum of the lengths is less than the total length of the text buffer (the `iTextLength` parameter), the remaining characters are assigned to the last style run.

`iStyles` An array of references of type `ATSUStyle` (page 191). Each element in the array must contain a valid style object that corresponds to a style run length in the `iRunLengths` array.

`oTextLayout` A pointer to a reference of type `ATSUTextLayout` (page 191). On return, the newly-created text layout object. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUCreateTextLayoutWithTextPtr` function creates a text layout object with style runs, a pointer to associated text, and default text layout attribute values. The default text layout attribute values are described in Table C-2 (page 245). You can create a text layout object that contains a handle to associated text by calling the function `ATSUCreateTextLayoutWithTextHandle` (page 88).

Most functions that operate on text layout objects perform these operations on the range of text that you specify in the `iTextOffset` and `iTextLength` parameters. Typically, this is a subrange of the entire text buffer. If this range is shorter than the entire text buffer, the text layout object will scan the remaining text to get the full context for bidirectional processing and other information about the text.

You are responsible for updating the memory location of the text associated with a text layout object whenever the user inserts, deletes, or moves text. To determine the current text memory location, call the function `ATSUGetTextLocation` (page 115).

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 beginning with ATSUI 1.0.

ATSUCreateTextLayoutWithTextHandle

Creates a text layout object with style runs, a handle to associated text, and default text layout attribute values.

```
OSStatus ATSUCreateTextLayoutWithTextHandle (
                    UniCharArrayHandle iText,
                    UniCharArrayOffset iTextOffset,
                    UniCharCount iTextLength,
                    UniCharCount iTextTotalLength,
                    ItemCount iNumberOfRuns,
```

```
                    UniCharCount iRunLengths[],
                    ATSUStyle iStyles[],
                    ATSUTextLayout *oTextLayout);
```

`iText` A pointer of type `UniCharArrayHandle` (page 193). Pass a handle that points to the beginning of a text buffer. Note that ATSUI expects the buffer to contain Unicode text. Your application is responsible for allocating the memory associated with this handle. ATSUI functions that need to access text referenced by this handle will return the handle to its original state (locked or unlocked) upon function completion.

`iTextOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset in backing store memory that corresponds to the beginning of range of text that you want ATSUI to perform layout operations on. If you want the range of text to start at the beginning of the text buffer, you should pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If you want the range of text to cover the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iTextLength` parameter. If the specified range of text is outside the text buffer, `ATSUCreateTextLayoutWithTextHandle` returns the result code `kATSUInvalidTextRangeErr`.

`iTextLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text that you want ATSUI to perform layout operations on. If you want the range of text to span the end of the text buffer, you should pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). If you want the range of text to cover the entire text buffer, pass `kATSUToTextEnd` in this parameter and `kATSUFromTextBeginning` in the `iTextOffset` parameter. If the specified range of text is outside the text buffer, `ATSUCreateTextLayoutWithTextHandle` returns the result code `kATSUInvalidTextRangeErr`.

`iTextTotalLength`
A value of type `UniCharCount` (page 194). Pass 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`	The number of style runs to assign to the text layout object. This should be equivalent to the number of elements in the `iRunLengths` and `iStyles` arrays.
`iRunLengths`	An array of values of type `UniCharCount` (page 194). Each element in the array must contain a style run length that corresponds to a style object in the `iStyles` array. You can pass `kATSUToTextEnd` for the last style run length if you want it to extend to end of the text buffer. If the sum of the lengths is less than the total length of the text buffer (the `iTextLength` parameter), the remaining characters are assigned to the last style run.
`iStyles`	An array of references of type `ATSUStyle` (page 191). Each element in the array must contain a valid style object that corresponds to a style run length in the `iRunLengths` array.
`oTextLayout`	A pointer to a reference of type `ATSUTextLayout` (page 191). On return, the newly-created text layout object. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUCreateTextLayoutWithTextHandle` function creates a text layout object with style runs, a handle to associated text, and default text layout attribute values. The default text layout attribute values are described in Table C-2 (page 245). You can create a text layout object that contains a pointer to associated text by calling the function `ATSUCreateTextLayoutWithTextHandle` (page 88).

Most functions that operate on text layout objects perform these operations on the range of text that you specify in the `iTextOffset` and `iTextLength` parameters. Typically, this is a subrange of the entire text buffer. If this range is shorter than the entire text buffer, the text layout object will scan the remaining text to get the full context for bidirectional processing and other information about the text.

You are responsible for updating the memory location of the text associated with a text layout object whenever the user inserts, deletes, or moves text. To determine the current text memory location, call the function `ATSUGetTextLocation` (page 115).

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 beginning 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 191). 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 191). On return, the newly-created text layout object. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUCreateAndCopyTextLayout` function creates a copy 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.

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 beginning with ATSUI 1.1.

ATSUSetTextLayoutRefCon

Sets application-specific text layout object data.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). 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 231).

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 responsible for freeing any memory allocated for the reference constant. Calling `ATSUDisposeTextLayout` (page 94) will not do so.

VERSION NOTES

Available with ATSUI 1.0.

ATSUGetTextLayoutRefCon

Obtains application-specific text layout object data.

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

`iTextLayout`	A reference of type `ATSUTextLayout` (page 191). 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 231).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUClearLayoutCache

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

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

`iTextLayout`	A reference of type `ATSUTextLayout` (page 191). 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 193). Pass the edge offset of the beginning of the line whose layout cache you want to discard. To clear the entire layout cache of an text layout object, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231).
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

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. When you clear the layout cache of a line or block of text, it takes ATSUI longer to redraw a line, since glyph layout must be performed again.

The `ATSUClearLayoutCache` function flushes the layout cache but does not alter previously set text layout attributes, soft line break positions, or the text memory location. Individual lines may be redrawn as before. If you do not care about retaining these values, you should dispose of the text layout object by calling the `ATSUDisposeTextLayout` (page 94) function.

You should call `ATSUClearLayoutCache` when line breaks in a text layout object are altered (for example, if line justification is set to full justification). You can call `ATSUClearLayoutCache` to free memory associated with the layout results of a text layout object.

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 beginning with ATSUI 1.1.

ATSUDisposeTextLayout

Disposes of a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to the text layout that you want to dispose of. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUDisposeTextLayout` function only frees memory associated with the text layout object and its internal structures. This includes text layout attributes, style runs, and soft line breaks. `ATSUDisposeTextLayout` does not dispose of the memory pointed to by reference constants. You are responsible for doing so.

VERSION NOTES

Available beginning with ATSUI 1.0.

Manipulating Text Layout Attributes

ATSUI provides the following functions for manipulating text layout attributes:

- `ATSUCopyLayoutControls` (page 95) copies all the text layout attribute values in a text layout object.
- `ATSUSetLayoutControls` (page 96) sets text layout attribute values for a text layout object.
- `ATSUGetLayoutControl` (page 98) obtains a text layout attribute value from a text layout object.
- `ATSUGetAllLayoutControls` (page 99) obtains text layout attribute information for a text layout object.
- `ATSUClearLayoutControls` (page 101) removes previously set text layout attributes from a text layout object.

ATSUCopyLayoutControls

Copies all the text layout attribute values in a text layout object.

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

`iSource` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object whose attributes you want to copy. You cannot pass `NULL` for this parameter.

`iDest` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object whose attributes you want to replace. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUCopyLayoutControls` function copies all the text layout attribute values in a text layout object. This includes previously set attributes as well as unset ones, which ATSUI sets to the default values listed in Table C-2 (page 245). If you wish to copy the text layout attribute values of a single line in a text layout object, see the function `ATSUCopyLineControls` (page 103).

`ATSUCopyLayoutControls` does not copy the contents of memory referenced by pointers or handles within reference constants. You are responsible for ensuring 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 beginning with ATSUI 1.0.

ATSUSetLayoutControls

Sets text layout attribute values for a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a text layout object whose attributes you want to set. You cannot pass `NULL` for this parameter.

`iAttributeCount`
The number of text layout attributes you want to set. This value should correspond to the number of elements in the `iTag` and `iValueSize` arrays.

`iTag`	An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that corresponds to the text layout attribute value you wish to set. See "Text Layout Attribute Tag Constants" (page 226) for a description of the Apple-defined text layout attribute tag constants. If you pass a style run attribute tag constant or an ATSUI-reserved tag constant in this parameter, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeTagErr`. You cannot pass `NULL` for this parameter.
`iValueSize`	An array of values of type `ByteCount`. Each element in the array must contain the size (in bytes) of the corresponding text layout attribute value being set. If you pass a size that is less than required, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeSizeErr`, and the function sets no attributes. If, after having checked all the given sizes and found them acceptable, `ATSUSetLayoutControls` sets text layout attributes. You cannot pass `NULL` for this parameter.
`iValue`	An array of pointers of type `ATSUAttributeValuePtr` (page 186). Each pointer in the array must reference a valid value and correspond to a tag in the `iTag` array. If you pass an invalid or undefined value, `ATSUSetLayoutControls` returns the result code `kATSUInvalidAttributeValueErr`. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 231). If there is a function error, `ATSUSetAttributes` will not set any style run attributes.

DISCUSSION

The `ATSUSetLayoutControls` function sets one or more text layout attribute value(s) for an entire text layout object. If you wish to set the text layout attribute values of a single line in a text layout object, see the function `ATSUSetLineControls` (page 104). Note that when you set a text layout attribute value for a line, this value will override the value of the text layout attribute set for the text layout object containing the line. This is true even if the attributes for the line are set before those of the entire text layout object containing the line.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetLayoutControl

Obtains a text layout 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 191). Pass a reference to a text layout object whose attribute value you want to obtain. 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 Attribute Tag Constants" (page 226) for a description of the Apple-defined text layout attribute tag constants. If you pass a style run attribute tag constant or an ATSUI-reserved tag in this parameter, `ATSUGetLayoutControl` returns the result code `kATSUInvalidAttributeTagErr`.

`iMaximumValueSize`
The maximum size of the text layout attribute value. If you pass a size that is less than required, `ATSUGetLayoutControl` returns the result code `kATSUInvalidAttributeSizeErr`.

`oValue` A pointer of type `ATSUAttributeValuePtr` (page 186). Before calling `ATSUGetLayoutControl`, pass a pointer to memory you have allocated for the attribute value. On return, `oValue` points

to the text layout attribute value. If the attribute was not previously set, `ATSUGetLayoutControl` passes back its default value in this parameter and returns the result code `kATSUNotSetErr`.

`oActualValueSize`
A pointer to a count. On return, 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 application-defined text layout attributes.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

Before calling the `ATSUGetLayoutControl` function, call the function `ATSUGetAllLayoutControls` (page 99) to obtain an array of the tags and data sizes corresponding to all previously set text layout attribute values for every line in a text layout object. To determine the value of a text layout attribute identified by a particular text layout attribute tag, you should pass the appropriate tag and data size pair passed back in the `oAttributeInfoArray` array of `ATSUGetAllLayoutControls` to `ATSUGetLayoutControl`. To determine the value of a text layout attribute value in a single line of a text layout object, call the function `ATSUGetLineControl` (page 106).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetAllLayoutControls

Obtains text layout attribute information from a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a text layout object whose attribute information you want to obtain. You cannot pass `NULL` for this parameter.

`oAttributeInfoArray`
An array of structures of type `ATSUAttributeInfo` (page 185). Before calling `ATSUGetAllAttributes`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains the tag and data size pairs corresponding to all previously set text layout attribute values in the text layout object.

`iTagValuePairArraySize`
The maximum number of tag and data size pairs in the text layout object. Typically, this is equivalent to the number of `ATSUAttributeInfo` structures in the `oAttributeInfoArray` array. To determine this value, see the discussion below.

`oTagValuePairCount`
A pointer to a count. On return, the actual number of `ATSUAttributeInfo` structures in the text layout object. This may be greater than the value you specified in the `iTagValuePairArraySize` parameter. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUGetAllLayoutControls` function obtains an array of the tags and data sizes corresponding to all previously set text layout attribute values for an entire text layout object. You can obtain a particular attribute value by passing the corresponding tag and data size pair to the function `ATSUGetLayoutControl` (page 98). To obtain the tags and data sizes corresponding to all previously set text layout attribute values in a line of a text layout object, call the function `ATSUGetAllLineControls` (page 108).

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

1. Pass a reference to the text layout object containing the attribute in the `iTextLayout` parameter, `NULL` for the `oAttributeInfoArray` parameter, and 0

for the other parameters. `ATSUGetAllLayoutControls` returns the size of the `oAttributeInfoArray` array in the `oTagValuePairCount` parameter.

2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oAttributeInfoArray` parameter; on return, the pointer references the `oAttributeInfoArray` array.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUClearLayoutControls

Removes previously set text layout attributes from a text layout object.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a text layout object whose attributes you want to remove. You cannot pass `NULL` for this parameter.

`iTagCount` The number of attribute values you want to remove. To remove all previously set attributes, pass the constant `kATSUClearAll` in this parameter. In this case, the value in the `iTag` parameter will be ignored.

`iTag` An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that corresponds to the text layout attribute value you want to remove. See "Text Layout Attribute Tag Constants" (page 226) for a description of the Apple-defined text layout attribute tag constants. If you pass a style run attribute tag constant or an ATSUI-reserved tag in this parameter, `ATSUClearLayoutControls` returns the result code `kATSUInvalidAttributeTagErr`.

function result A result code. See "Result Codes" (page 231). You can remove unset attribute values from a text layout object without a function error.

DISCUSSION

The `ATSUClearLayoutControls` function removes those previously set text layout attribute values that are identified by tag constants in the `iTag` array. It replaces them with the default text layout attribute values described in Table C-2 (page 245).

If you wish to remove all previously set text layout attribute values from a text layout object, pass `kATSUClearAll` in the `iTagCount` parameter. If you wish to remove the previously set text layout attributes in a line of text, call the function `ATSUClearLineControls` (page 109).

VERSION NOTES

Available beginningwith ATSUI 1.0.

Manipulating Text Layout Attributes in a Line

ATSUI provides the following functions for manipulating text layout attributes in a line of a text layout object:

- `ATSUCopyLineControls` (page 103) copies text layout attribute values from one line to another.
- `ATSUSetLineControls` (page 104) sets text layout attribute values for a line of text.
- `ATSUGetLineControl` (page 106) obtains a text layout attribute value from a line of text.
- `ATSUGetAllLineControls` (page 108) obtains text layout attribute information for a line of text.
- `ATSUClearLineControls` (page 109) removes previously set text layout attribute values from a line of text.

ATSUCopyLineControls

Copies text layout attribute values from one line to another.

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

`iSourceTextLayout`
A reference of type `ATSUTextLayout` (page 191). 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 193). Pass the edge offset of the beginning of the line whose attribute values you want to copy.

`iDestTextLayout`
A reference of type `ATSUTextLayout` (page 191). Pass a reference to the initialized text layout object containing 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 text layout attributes within a text layout object. You cannot pass `NULL` for this parameter.

`iDestLineStart`
A value of type `UniCharArrayOffset` (page 193). 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 231).

DISCUSSION

The `ATSUCopyLineControls` function copies all the text layout attribute values from a line in a text layout object. This includes previously set attributes as well as unset ones, which ATSUI sets to the default values listed in Table C-2 (page 245). If you wish to copy the text layout attribute values of an entire text layout object to another text layout object, see the function `ATSUCopyLayoutControls` (page 95).

`ATSUCopyLineControls` does not copy the contents of memory referenced by pointers or handles within reference constants. You are responsible for ensuring 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 beginning with ATSUI 1.1.

ATSUSetLineControls

Sets text layout attribute values for a line of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a text layout object that contains the line whose attribute values you want to set. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset of the beginning of the line whose attribute values you want to set.

`iAttributeCount`
The number of attributes you want to set for the line. This value should correspond to the number of elements in the `iTag`, `iValueSize`, and `iValue` arrays.

`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 value in the line. See "Text Layout Attribute Tag Constants" (page 226) for a description of Apple-defined tag values. 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`. Each element in the array must contain the size (in bytes) of the corresponding text layout attribute value being set. If you pass a size that is less than required, `ATSUSetLineControls` returns the result code `kATSUInvalidAttributeSizeErr`, and the function sets no attributes. If, after having checked all the given sizes and found them acceptable, `ATSUSetLayoutControls` sets text layout attributes. You cannot pass `NULL` for this parameter.
`iValue`	An array of pointers of type `ATSUAttributeValuePtr` (page 186). Each pointer in the array must reference a valid value and correspond to a tag in the `iTag` array. If you pass an invalid or undefined value, `ATSUSetLineControls` returns the result code `kATSUInvalidAttributeValueErr`. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 231). If there is a function error, `ATSUSetAttributes` will not set any style run attributes.

DISCUSSION

The `ATSUSetLineControls` function sets one or more text layout attribute value(s) for a single line in a text layout object. Note that when you set a text layout attribute value for a line, this value will override the value of the text layout attribute set for the text layout object containing the line. This is true even if the attributes for the line are set before those of the entire text layout object containing the line. If you wish to set the text layout attribute values for an entire text layout object, see the function `ATSUSetLayoutControls` (page 96).

ATSUI functions that operate on a line of text like `ATSUDrawText` (page 163), `ATSUMeasureText` (page 150), `ATSUMeasureTextImage` (page 153), and `ATSUGetGlyphBounds` (page 146), use text layout attributes that have been set by calling `ATSUSetLineControls` to calculate dimensions. If none have been set for

the line, they use those set for the text layout object containing the line. If none have been set for the text layout object containing the line, ATSUI assigns them their default value. Default text layout attribute values are described in Table C-2 (page 245).

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUGetLineControl

Obtains a text layout attribute value from a line of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference points to an initialized text layout object that contains the line whose attribute value you want to obtain. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 193). 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 text layout attribute whose value you want to determine. See "Text Layout Attribute Tag Constants" (page 226) for a description of Apple-defined tag values. If you

pass a style run attribute or an ATSUI-reserved tag value in this parameter, `ATSUGetLineControl` returns the result code `kATSUInvalidAttributeTagErr`.

`iExpectedValueSize`
The maximum size of the text layout attribute value. To determine the size of an application-defined text layout attribute value, see the discussion below. If the size is less than required, `ATSUGetLineControl` returns the result code `kATSUInvalidAttributeSizeErr`.

`oValue` A pointer of type `ATSUAttributeValuePtr` (page 186). Before calling `ATSUGetLayoutControl`, pass a pointer to memory you have allocated for the attribute value. On return, `oValue` points to the text layout attribute value. If the attribute was not previously set, `ATSUGetLineControl` passes back its default value in this parameter and returns the result code `kATSUNotSetErr`.

`oActualValueSize`
A pointer to a count. On return, 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 application-defined text layout attributes.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

Before calling the `ATSUGetLineControl` function, call the function `ATSUGetAllLineControls` (page 108) to obtain an array of the tags and data sizes corresponding to all previously set text layout attribute values for a line of text. To determine the value of a text layout attribute identified by a particular text layout attribute tag, you should pass the corresponding tag and data size pair passed back from `ATSUGetAllLineControls` in one of the structures in the `oAttributeInfoArray` array to `ATSUGetLineControl`. If you wish to obtain a particular text layout attribute value for an entire text layout object, call the function `ATSUGetLayoutControl` (page 98).

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUGetAllLineControls

Obtains text layout attribute information for a line of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a text layout object that contains the line whose attribute information you want to obtain. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset in backing store memory that corresponds to the beginning of the line whose text layout attribute values you want to determine.

`oAttributeInfoArray`
An array of structures of type `ATSUAttributeInfo` (page 185). Before calling `ATSUGetAllAttributes`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains the tag and data size pairs corresponding to all previously set text layout attribute values in the text layout object.

`iTagValuePairArraySize`
The maximum number of tag and data size pairs in the text layout object. Typically, this is equivalent to the number of `ATSUAttributeInfo` structures in the `oAttributeInfoArray` array. To determine this value, see the discussion below.

`oTagValuePairCount`
A pointer to a count. On return, the actual number of `ATSUAttributeInfo` structures in a line of the text layout object. This may be greater than the value you specified in the `iTagValuePairArraySize` parameter. You cannot pass `NULL` for this parameter.

DISCUSSION

The `ATSUGetAllLineControls` function obtains, for a line of text, an array of the tags and data sizes corresponding to all previously set text layout attribute values. You can obtain a particular attribute value by passing the corresponding tag and data size pair to the function `ATSUGetLineControl` (page 106). To obtain the tags and data sizes corresponding to all previously set text layout attribute values in an entire text layout object, call the function `ATSUGetAllLayoutControls` (page 99).

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

1. Pass a reference to the text layout object containing the attribute in the `iTextLayout` parameter, `NULL` for the `oAttributeInfoArray` parameter, and 0 for the other parameters. `ATSUGetAllLineControls` returns the size of the `oAttributeInfoArray` array in the `oTagValuePairCount` parameter.
2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oAttributeInfoArray` parameter; on return, the pointer references the `oAttributeInfoArray` array.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUClearLineControls

Removes previously set text layout attribute values from a line of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a text layout object that contains the line whose attributes you want to remove. You cannot pass `NULL` for this parameter.

`iLineStart`	A value of type `UniCharArrayOffset` (page 193). Pass the edge offset in backing store memory that corresponds to the beginning of the line whose text layout attribute values you want to remove.
`iTagCount`	The number of attributes you want to remove. To remove all previously set text layout attributes, pass the constant `kATSUClearAll` in this parameter. In this case, the value in the `iTag` parameter will be ignored.
`iTag`	An array of values of type `ATSUAttributeTag`. Each element in the array must contain a valid tag that corresponds to the text layout attribute value you want to remove. See "Text Layout Attribute Tag Constants" (page 226) for a description of the Apple-defined text layout attribute tag constants. If you pass a style run attribute tag constant or an ATSUI-reserved tag in this parameter, `ATSUClearLayoutControls` returns the result code `kATSUInvalidAttributeTagErr`.
function result	A result code. See "Result Codes" (page 231). You can remove unset attribute values from a line without a function error.

DISCUSSION

The `ATSUClearLineControls` function removes, for a line of text, those previously set text layout attribute values that are identified by tag constants in the `iTag` array. It replaces them with the default text layout attribute values described in Table C-2 (page 245).

If you wish to remove all previously set text layout attribute values from a line in a text layout object, pass `kATSUClearAll` in the `iTagCount` parameter. If you wish to remove the previously set text layout attributes for an entire text layout object, call the function `ATSUClearLayoutControls` (page 101).

VERSION NOTES

Available beginning with ATSUI 1.1.

Assigning and Updating Text

ATSUI provides the following functions for assigning and updating text in a text layout object:

- `ATSUSetTextPointerLocation` (page 111) assigns or updates text accessed with a pointer.
- `ATSUSetTextHandleLocation` (page 113) assigns or updates text accessed with a handle.

ATSUSetTextPointerLocation

Assigns or updates text accessed with a pointer.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a valid text layout object.

`iText` A pointer of type `ConstUniCharArrayPtr` (page 192). Pass a pointer to the beginning of a text buffer. Note that ATSUI expects the buffer to contain Unicode text. Your application is responsible for allocating the memory associated with this pointer.

`iTextOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset in backing store memory that corresponds to the beginning of range of text that you want ATSUI to perform layout operations on. If you want the range of text to start at the beginning of the text buffer, you should pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If you want the range of text to cover the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and `kATSUToTextEnd` in the `iTextLength` parameter. If the specified range of text is outside the text buffer, `ATSUSetTextPointerLocation` returns the result code `kATSUInvalidTextRangeErr`.

`iTextLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text that you want ATSUI to perform layout operations on. If you want the range of text to span the end of the text buffer, you should pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). If you want the range of text to cover the entire text buffer, pass `kATSUToTextEnd` in this parameter and `kATSUFromTextBeginning` in the `iTextOffset` parameter. If the specified range of text is outside the text buffer, `ATSUSetTextPointerLocation` returns the result code `kATSUInvalidTextRangeErr`.

`iTextTotalLength`
A value of type `UniCharCount` (page 194). Pass 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.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUSetTextPointerLocation` function assigns new text or updates existing text that is accessed by a handle. For uninitialized text layout objects, `ATSUSetTextPointerLocation` assigns new text; for initialized text layout objects, it updates existing text that is accessed by a pointer. `ATSUSetTextPointerLocation` clears layout caches. If you want to assign or update text accessed by a handle, call the function `ATSUSetTextHandleLocation` (page 113).

If the user deletes, inserts, or moves text in a text layout object and the range of text covers the entire text buffer, you should

- call the function `ATSUSetTextPointerLocation` or `ATSUSetTextHandleLocation` (page 113) to update the text
- call `ATSUSetRunStyle` (page 118) to update the style runs

You can then call the function `ATSUDrawText` (page 163) to display the updated text.

Most functions that operate on text layout objects perform these operations on the range of text that you specify in the `iTextOffset` and `iTextLength` parameters. Typically, this is a subrange of the entire text buffer. If this range is

shorter than the entire text buffer, the text layout object will scan the remaining text to get the full context for bidirectional processing and other information about the text.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUSetTextHandleLocation

Assigns or updates text accessed with a handle.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to a valid text layout object.

`iText` A handle of type `UniCharArrayHandle` (page 193). Pass a handle that points to the beginning of a text buffer. Note that ATSUI expects the buffer to contain Unicode text. Your application is responsible for allocating the memory associated with this handle. ATSUI functions that need to access text referenced by this handle will return the handle to its original state (locked or unlocked) upon function completion.

`iTextOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset in backing store memory that corresponds to the beginning of range of text that you want ATSUI to perform layout operations on. If you want the range of text to start at the beginning of the text buffer, you should pass the constant

kATSUFromTextBeginning, described in "Text Offset Constant" (page 231). If you want the range of text to cover the entire text buffer, pass kATSUFromTextBeginning in this parameter and kATSUToTextEnd in the iTextLength parameter. If the specified range of text is outside the text buffer, ATSUSetTextHandleLocation returns the result code kATSUInvalidTextRangeErr.

`iTextLength` A value of type UniCharCount (page 194). Pass the length of the range of text that you want ATSUI to perform layout operations on. If you want the range of text to span the end of the text buffer, you should pass the constant kATSUToTextEnd, described in "Text Length Constant" (page 230). If you want the range of text to cover the entire text buffer, pass kATSUToTextEnd in this parameter and kATSUFromTextBeginning in the iTextOffset parameter. If the specified range of text is outside the text buffer, ATSUSetTextHandleLocation returns the result code kATSUInvalidTextRangeErr.

`iTextTotalLength`
A value of type UniCharCount (page 194). Pass 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.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The ATSUSetTextHandleLocation function assigns it new text or updates existing text that is accessed by a handle. For uninitialized text layout objects, ATSUSetTextHandleLocation assigns new text; for initialized text layout objects, it updates existing text that is accessed by a handle. ATSUSetTextHandleLocation clears layout caches. If you want to assign or update text accessed by a pointer, call the function ATSUSetTextPointerLocation (page 111).

If the user deletes, inserts, or moves text in a text layout object and the range of text covers the entire text buffer, you should

- call the function ATSUSetTextPointerLocation (page 111) or ATSUSetTextHandleLocation to update the text
- call ATSUSetRunStyle (page 118) to update the style runs

You can then call the function `ATSUDrawText` (page 163) to display the updated text.

Most functions that operate on text layout objects perform these operations on the range of text that you specify in the `iTextOffset` and `iTextLength` parameters. Typically, this is a subrange of the entire text buffer. If this range is shorter than the entire text buffer, the text layout object will scan the remaining text to get the full context for bidirectional processing and other information about the text.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

Obtaining and Updating Text Memory Location

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

- `ATSUGetTextLocation` (page 115) obtains information about text, including its in physical memory.
- `ATSUTextMoved` (page 117) updates the location of text in physical memory.

ATSUGetTextLocation

Obtains information about text, including its in physical memory.

```
OSStatus ATSUGetTextLocation (
                    ATSUTextLayout iTextLayout,
                    void **oText,
                    Boolean *oTextIsStoredInHandle,
```

```
                    UniCharArrayOffset *oOffset,
                    UniCharCount *oTextLength,
                    UniCharCount *oTextTotalLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object.

`oText` A pointer of type `ConstUniCharArrayPtr` (page 192) or a handle of type `UniCharArrayHandle` (page 193), depending on the value passed back in `oTextIsStoredInHandle`. If `true`, on return, `oText` contains a handle pointing to the beginning of the text buffer. If `false`, on return, `oText` contains a pointer to the beginning of the text buffer.

`oTextIsStoredInHandle`
A pointer to a `Boolean` value. On return, the value indicates whether the text buffer in `oText` is accessed by a handle or 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 193). On return, the edge offset in backing store memory that corresponds to the beginning of range of text that ATSUI performs layout operations on.

`oTextLength` A pointer to a value of type `UniCharCount` (page 194). On return, the length of the range of text that ATSUI performs layout operations on.

`oTextTotalLength`
A pointer to a value of type `UniCharCount` (page 194). On return, the length of the entire text buffer.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUGetTextLocation` function obtains the location of text in physical memory, the length of the range of text and text buffer, and whether the text is accessed by a pointer or handle.

VERSION NOTES

Available beginning 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 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iNewLocation` A pointer of type `ConstUniCharArrayPtr` (page 192). Pass the beginning of the updated location of the text buffer physical memory of the text buffer.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUTextMoved` function updates the location of text in physical memory to reflect the location that you moved the buffer to. You are responsible for moving the text. The text buffer should remain otherwise unchanged.

If the user deletes, inserts, or moves text in a text layout object and the range of text covers the entire text buffer, you should not call `ATSUTextMoved`. Instead,

- call the function `ATSUSetTextPointerLocation` (page 111) or `ATSUSetTextHandleLocation` (page 113) to update the text
- call the function `ATSUSetRunStyle` (page 118) to update the style runs

You can then call the function `ATSUDrawText` (page 163) to display the updated text.

VERSION NOTES

Available beginning with ATSUI 1.0.

Assigning and Updating Style Runs

ATSUI provides the following function for assigning and updating style runs:

- `ATSUSetRunStyle` (page 118) assigns or updates style runs.

ATSUSetRunStyle

Assigns or updates style runs.

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

`iTextLayout`	A reference of type `ATSUTextLayout` (page 191). Pass a reference to a valid text layout object.
`iStyle`	A reference of type `ATSUStyle` (page 191). Pass a reference to a valid style object whose attributes, font features, and font variations you want to set (if uninitialized) or replace (if initialized). You cannot pass `NULL` for this parameter.
`iRunStart`	A value of type `UniCharArrayOffset` (page 193). Pass the edge offset of the beginning of the style run whose attributes, font features, and font variations you want to set (if uninitialized) or replace (if initialized).
`iRunLength`	A value of type `UniCharCount` (page 194). Pass the length of the style run whose attributes, font features, and font variations you want to set (if uninitialized) or replace (if initialized).
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUSetRunStyle` function assigns new style runs or updates existing style runs. For uninitialized text layout objects, `ATSUSetRunStyle` assigns new style runs; for initialized text layout objects, it updates replaces all previously set attributes, font features, and font variations in the `iStyle` parameter. After completion, `ATSUSetRunStyle` adjusts the lengths of the style runs on either side of the new or updated style run.

If the user deletes, inserts, or moves text in a text layout object and the range of text covers the entire text buffer, you should

- call the function `ATSUSetTextPointerLocation` (page 111) or `ATSUSetTextHandleLocation` (page 113) to update the text
- call `ATSUSetRunStyle` to update the style runs

You can then call the function `ATSUDrawText` (page 163) to display the updated text.

SPECIAL CONSIDERATIONS

You are responsible for disposing of the memory allocated for the new style run when you dispose of it. Calling the function `ATSUDisposeTextLayout` (page 94) will not do so. `ATSUSetRunStyle` may allocate memory in your application heap, unless you designate a different heap by calling the function `ATSUCreateMemorySetting` (page 174).

VERSION NOTES

Available beginning with ATSUI 1.0.

Obtaining Style Run Information

ATSUI provides the following functions for obtaining style run information:

- `ATSUGetRunStyle` (page 119) finds the style run information at a given location and the range of text that shares this information.
- `ATSUGetContinuousAttributes` (page 121) finds the style run information that is continuous over a range of text.

ATSUGetRunStyle

Finds the style run information at a given location and 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 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iOffset`	A value of type `UniCharArrayOffset` (page 193). Pass the edge offset of the location whose style run information and the range of text sharing that information you want to find. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If you pass an edge offset that is outside the text buffer, `ATSUGetRunStyle` returns the result code `kATSUInvalidTextRangeErr`.
`oStyle`	A pointer to a reference of type `ATSUStyle` (page 191). On return, the style run information of the location in `iOffset` that is shared by the range of text from `oRunStart` to `oRunLength`. If the location you passed in `iOffset` is at a style run boundary, `ATSUGetRunStyle` passes back the style run information of the next style run in this parameter.
`oRunStart`	A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the edge offset of the beginning of the style run that includes the location specified in `iOffset`.
`oRunLength`	A pointer to a value of type `UniCharCount` (page 194). On return, the length of the text sharing the same style run attributes, font features, and font variations.
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUGetRunStyle` function finds the style run information at a given location and the range of text that shares this information (including style run attributes, font features, and font variations).

If there is only one style run set in the text layout object (whether or not it covers the entire range of text in the text layout object), `ATSUGetRunStyle` passes back the style run information at the specified offset and uses it to set the style run information of the remaining text.

If you want to find the style run information that is continuous over a range of text, call the function `ATSUGetContinuousAttributes` (page 121).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetContinuousAttributes

Finds the style run information that is continuous over a range of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset of the beginning of the range of text whose continuous style run information you want to determine. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If you pass an edge offset that is outside the text buffer, `ATSUGetContinuousAttributes` returns the result code `kATSUInvalidTextRangeErr`.

`iRunLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text whose continuous style run information you want to determine. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). If you pass a length that is outside the text buffer, `ATSUGetContinuousAttributes` returns the result code `kATSUInvalidTextRangeErr`.

`oStyle` A reference of type `ATSUStyle` (page 191). Pass a valid style object. On return, the contents of this object are filled with the continuous attributes. You cannot pass `NULL` for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUGetContinuousAttributes` function finds the style run information that is continuous over a range of text, including the default values of unset attributes, font features, and font variations.

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 find the style run information at a given location and the range of text that shares this information, call the function `ATSUGetRunStyle` (page 119).

VERSION NOTES

Available beginning with ATSUI 1.0.

Mapping Font Fallbacks

ATSUI provides the following functions for fallback mapping:

- `ATSUSetFontFallbacks` (page 123) indicates the fonts to search and the search order to use when fallback mapping is needed.
- `ATSUGetFontFallbacks` (page 124) obtains the fonts to employ and search order to use when fallback mapping is required.
- `ATSUMatchFontsToText` (page 125) obtains the first subrange of text whose character(s) cannot be drawn with the assigned font and recommends a substitute font.
- `ATSUSetTransientFontMatching` (page 128) provides automatic font substitution when a character(s) cannot be drawn with the assigned font.
- `ATSUGetTransientFontMatching` (page 129) indicates whether automatic font substitution is enabled.

ATSUSetFontFallbacks

Indicates the fonts to search and the search order to use when fallback mapping is needed.

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

`iFontFallbacksCount`
The number of fonts that you want searched when fallback mapping is required. This should be equivalent to the number of elements in the `iFontIDs` array.

`iFontIDs` An array of values of type `ATSUFontID` (page 188). Each element in the array should represent an ATSUI-compatible font, and the order of these elements should reflect the order that you want ATSUI to sequentially search them when fallback mapping is needed.

`iFontFallbackMethod`
A value of type `ATSUFontFallbackMethod`. Pass the order that you want fonts in the `iFontIDs` array to be searched. See "Font Fallback Constants" (page 205) for a description of possible values.

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 231).

DISCUSSION

The `ATSUSetFontFallbacks` function indicates the fonts to search and the search order to use 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 sequentially when a substitute font is needed, and the first match will be used. You can specify this default search behavior by passing `kATSUDefaultFontFallbacks` in the `iFontFallbackMethod` parameter. This is equivalent to calling the function `ATSUMatchFontsToText` (page 125). `ATSUSetFontFallbacks` will use the first valid font it finds in the font list you specify.

If you call `ATSUSetFontFallbacks` and carefully order your chosen font fallbacks, the amount of time needed by ATSUI to find a suitable fallback for specific text can be significantly reduced.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUGetFontFallbacks

Obtains the fonts to employ and search order to use when fallback mapping is required.

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

`iMaxFontFallbacksCount`
The maximum number of fonts that you want searched. Typically, this is equivalent to the number of fonts in the `oFontIDs` array. To determine this value, see the discussion below.

`oFontIDs` An array of values of type `ATSUFontID` (page 188). Before calling `ATSUGetFontFallbacks`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains the fonts that you previously set to be searched for fallback mapping.

`oFontFallbackMethod`
A pointer to a value of type `ATSUFontFallbackMethod`. On return, the previously set order to search fonts in the `iFontIDs` array. See "Font Fallback Constants" (page 205) for a description of possible values.

`oActualFallbacksCount`
: A pointer to a count. On return, the actual number of fonts that you want searched. 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 231).

DISCUSSION

The `ATSUGetFontFallbacks` function obtains the fonts to employ and search order to use that you set in a previous set to the function `ATSUSetFontFallbacks` (page 123). They can also reflect the default fallback mapping order if you did not call `ATUSetFontFallbacks`.

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

1. Pass `NULL` for the `oFontIDs` parameter and 0 for the other parameters. `ATSUGetFontFallbacks` returns the size of the font array in the `oActualFallbacksCount` parameter.
2. Allocate enough space for an array of the returned size, then call the function again, passing a pointer in the `oFontIDs` parameter; on return, the pointer references an array of fonts to be searched.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUMatchFontsToText

Obtains the first subrange of text whose character(s) cannot be drawn with the assigned font and recommends a substitute font.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iTextStart` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset of the beginning of the range of text that you want to scan for character(s) that could not be drawn with the assigned font. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If you pass an edge offset that is outside the text buffer, `ATSUMatchFontsToText` returns the result code `kATSUInvalidTextRangeErr`.

`iTextLength` A value of type `UniCharCount` (page 194). Pass the length of the of the range of text that you want to scan for character(s) that could not be drawn with the assigned font. To indicate the beginning of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). If you pass a length that is outside the text buffer, `ATSUMatchFontsToText` returns the result code `kATSUInvalidTextRangeErr`.

`oFont` A pointer to a value of type `ATSUFontID` (page 188). On return, the substitute font recommended by `ATSUMatchFontsToText`.

`oChangedOffset`
A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the edge offset of the beginning of the range of text containing character(s) whose font does not have all the glyphs necessary to draw it.

`oChangedLength`
A pointer to a value of type `UniCharCount` (page 194). On return, the length of the range of text containing character(s) whose font does not have all the glyphs necessary 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 231).

DISCUSSION

The `ATSUMatchFontsToText` function obtains the first subrange of text whose character(s) cannot be drawn with the assigned font and recommends a substitute font. It does not actually perform font substitutions. If you want ATSUI to perform font substitutions, call the function `ATSUSetTransientFontMatching` (page 128).

`ATSUMatchFontsToText` scans all valid fonts on the user's system for a substitute font. This is the default search behavior used if you do not call the function `ATSUSetFontFallbacks` (page 123), or if you pass the constant `kATSUDefaultFontFallbacks` in the `iFontFallbackMethod` parameter of the 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 could be drawn.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUSetTransientFontMatching

Provides automatic font substitution when a character(s) cannot be drawn with the assigned font.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iTransientFontMatching`
A `Boolean` value. Pass a value that indicates whether you want ATSUI to perform automatic font substitution when a character(s) could not be drawn with the assigned font. If `true`, ATSUI will perform automatic font substitution.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUSetTransientFontMatching` function provides automatic font substitution when a character(s) cannot be drawn with the assigned font. Like `ATSUMatchFontsToText` (page 125), `ATSUSetTransientFontMatching` uses the default search behavior of scanning all valid fonts on the user's system for a substitute font.

When it performs the substitution, `ATSUSetTransientFontMatching` does not change the font attribute in the style object. As a result, if you plan to redraw a text layout object, you should instead call the function `ATSUSetFontFallbacks` (page 123) or `ATSUMatchFontsToText` (page 125).

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 beginning with ATSUI 1.0.

ATSUGetTransientFontMatching

Indicates whether automatic font substitution is enabled.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`oTransientFontMatching`
A pointer to a `Boolean` value. On return, the value indicates whether automatic font substitution is enabled. If `true`, it is enabled.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUGetTransientFontMatching` function indicates whether automatic font substitution is enabled by a previous call to the function `ATSUSetTransientFontMatching` (page 128).

VERSION NOTES

Available beginning with ATSUI 1.0.

Hit-Testing

ATSUI provides the following functions for hit-testing:

- `ATSUPositionToOffset` (page 130) obtains the edge offset(s) that correspond to the glyph nearest a mouse-down event.
- `ATSUOffsetToPosition` (page 134) obtains the caret position(s) that correspond to an edge offset.

ATSUPositionToOffset

Obtains the edge offset(s) that correspond to the glyph 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 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iLocationX` A value of type `ATSUTextMeasurement` (page 192). Pass the x-coordinate of the origin of the line (in the current graphics port) where the mouse-down event occurred. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want the location of the mouse-down event relative to the current pen location in the current graphics port.

`iLocationY` A value of type `ATSUTextMeasurement` (page 192). Pass the y-coordinate of the origin of the line (in the current graphics port) where the mouse-down event occurred. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want the location of the mouse-down event relative to the current pen location in the current graphics port.

`ioPrimaryOffset`
A pointer to a value of type `UniCharArrayOffset` (page 193). Before calling `ATSUPositionToOffset`, pass a pointer to the edge offset of the beginning of the line containing the character nearest the mouse-down event. If the line direction is right-to-left, pass the lowest edge offset. On return, the edge offset that corresponds to the closest edge of the glyph beneath the hit point on the screen. You cannot pass `NULL` for this parameter.

`oIsLeading` A pointer to a `Boolean` value. On return, the value indicates whether the offset passed back in the `ioPrimaryOffset` parameter is leading or trailing. If this value is `true`, the offset is leading (more closely associated with the following character in the backing-store memory). If this value is `false`, the offset is trailing (more closely associated with the previous character in the backing-store memory). This value indicates the line direction at the location of the mouse-down event.

`oSecondaryOffset`
A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the edge offset that corresponds to the furthest edge of the glyph beneath the hit point on the screen. This value is the same as passed back in `ioPrimaryOffset` unless the mouse-down event occurs on a line direction boundary.

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 231).

DISCUSSION

The `ATSUPositionToOffset` function obtains the edge offset(s) that correspond to the glyph nearest a mouse-down event. If the mouse-down event occurs at a line direction boundary, a second offset is passed back in the `oSecondaryOffset` parameter representing the offset that corresponds to the furthest edge of the glyph beneath the hit point on the screen. 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 value passed back in the `oIsLeading` parameter indicates the line direction of the text at the location of the mouse-down event.

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 134) 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 134) 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 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 transform 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

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUOffsetToPosition

Obtains the caret position(s) that correspond to an edge offset.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iOffset` A pointer to a value of type `UniCharArrayOffset` (page 193). Pass the edge offset whose corresponding caret position(s) you want to determine. If you wish to visually respond to a mouse-down event, pass the offset passed back in the `ioPrimaryOffset` parameter of the function `ATSUPositionToOffset` (page 130).

`iIsLeading` A `Boolean` value. This value is only relevant if the edge offset is at a line direction boundary. Pass `true` if the edge offset is leading. In this case, it corresponds to the first offset of the line whose origin is specified in the `iLocationX` and `iLocationY` parameters. Pass `false` if the edge offset is trailing. In this case, it corresponds to the last offset of the line whose origin is specified in the `iLocationX` and `iLocationY` parameters. 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 186). On return, the structure contains the starting and ending pen locations of the high caret if the value passed back in `oCaretIsSplit` is `true`. 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 186). On return, the structure contains the starting and ending pen locations of the low caret if the value passed back in `oCaretIsSplit` is `true`. If the passed back value is `false`, the structure contains the starting and ending pen locations of the main caret (same as the `oMainCaret` parameter).

`oCaretIsSplit`	A pointer to a `Boolean` value. On return, the value indicates whether the edge offset specified in `iOffset` occurs at a line direction boundary. If `true`, the offset occurs at a line direction boundary; otherwise, `false`.
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 231).

DISCUSSION

The `ATSUOffsetToPosition` function determines the caret position(s) that correspond to an edge offset.

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 location of the mouse-down event to the function `ATSUPositionToOffset` (page 130). `ATSUPositionToOffset` passes back the edge offset (or offsets, if the hit occurs on a line direction boundary) from the beginning of the line where the hit occurred.

The offset passed back by `ATSUPositionToOffset` (page 130) is relative to the beginning of the line in which the hit occurred, not from the beginning of the text layout object's buffer. Because this offset is a relative position, you must transform the starting and ending pen locations of the caret(s) that are passed back by `ATSUOffsetToPosition` before drawing them. The passed back carets do not need to be transformed to reflect angled and split caret appearances.

To do so, add the starting and ending caret coordinates to the coordinates of the origin of the line (in the current 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).

To draw the caret, you can call the QuickDraw functions `MoveTo` and `LineTo`, or equivalent functions.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

Determining Cursor Offsets

ATSUI provides the following functions for determining cursor offsets:

- `ATSUNextCursorPosition` (page 136) obtains the edge offset corresponding to the next cursor position based on the type of cursor movement.
- `ATSUPreviousCursorPosition` (page 138) obtains the edge offset corresponding to the previous cursor position based on the type of cursor movement.
- `ATSURightwardCursorPosition` (page 140) obtains the edge offset to the right of the high caret position based on the type of cursor movement.
- `ATSULeftwardCursorPosition` (page 141) obtains the edge offset to the left of the high caret position based on the type of cursor movement.

ATSUNextCursorPosition

Obtains the edge offset corresponding to the next cursor position based on the type of cursor movement.

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

`iTextLayout`	A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.
`iOldOffset`	A value of type `UniCharArrayOffset` (page 193). Pass the edge offset corresponding to the initial cursor position. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If the edge offset is outside the text buffer, `ATSUNextCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.
`iMovementType`	A value of type `ATSUCursorMovementType`. Pass the unit distance that the cursor was moved. See "Cursor Movement Constants" (page 204) for a description of possible values. You must pass a value between 2 bytes and a word in length.
`oNewOffset`	A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the edge offset corresponding to the new cursor position. This offset may be outside the text buffer.
function result	A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in styled data from an invalid cache. In this case, either 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 231).

DISCUSSION

The `ATSUNextCursorPosition` function determines the edge offset in backing store memory corresponding to the previous cursor position based on the type of cursor movement specified in the `iMovementType` parameter. You should call `ATSUNextCursorPosition` and the function `ATSUPreviousCursorPosition` (page 138) if the initial edge offset is not at a line boundary. If the initial edge offset is at a line direction boundary, you should instead call the functions `ATSURightwardCursorPosition` (page 140) and `ATSULeftwardCursorPosition` (page 141) to calculate the next and previous cursor positions. Note that at a line boundary, the caret is split into a high and low caret.

Note that you may not be able to move the cursor 2-bytes, since doing so might place the cursor in the middle of a surrogate pair.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUPreviousCursorPosition

Obtains the edge offset corresponding to the previous cursor position based on the type of cursor movement.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iOldOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset corresponding to the initial cursor position. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If the edge offset is outside the text buffer, `ATSUPreviousCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.

`iMovementType` A value of type `ATSUCursorMovementType`. Pass the unit distance that the cursor was moved. See "Cursor Movement Constants" (page 204) for a description of possible values. You must pass a value between 2 bytes and a word in length.

`oNewOffset`	A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the edge offset corresponding to the new cursor position. This offset may be outside the text buffer.
function result	A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in styled data from an invalid cache. In this case, either 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 231).

DISCUSSION

The `ATSUPreviousCursorPosition` function determines the edge offset in backing store memory corresponding to the previous cursor position based on the type of cursor movement specified in the `iMovementType` parameter. You should call `ATSUPreviousCursorPosition` and the function `ATSUNextCursorPosition` (page 136) if the initial edge offset is not at a line boundary. If the initial edge offset is at a line direction boundary, you should instead call the functions `ATSURightwardCursorPosition` (page 140) and `ATSULeftwardCursorPosition` (page 141) to calculate the next and previous cursor positions. Note that at a line boundary, the caret is split into a high and low caret.

Note that you may not be able to move the cursor 2-bytes, since doing so might place the cursor in the middle of a surrogate pair.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSURightwardCursorPosition

Obtains the edge offset corresponding to the right of the high caret position based on the type of cursor movement.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iOldOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset corresponding to the initial cursor position. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If the edge offset is outside the text buffer, `ATSURightwardCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.

`iMovementType` A value of type `ATSUCursorMovementType`. Pass the unit distance that the cursor was moved. See "Cursor Movement Constants" (page 204) for a description of possible values. You must pass a value between 2 bytes and a word in length.

`oNewOffset` A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the edge offset corresponding to the new cursor position. This offset may be outside the text buffer.

function result A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in styled data from an invalid cache. In this case, either 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 231).

DISCUSSION

The `ATSURightwardCursorPosition` function determines the edge offset in backing store memory corresponding to the right of the high caret position based on the type of cursor movement specified in the `iMovementType` parameter. You should call `ATSURightwardCursorPosition` and the function `ATSULeftwardCursorPosition` (page 141) when the initial edge offset is at a line boundary. At a line boundary, the caret is split into a high and low caret. If the initial edge offset is not at a line direction boundary, you should instead call the functions `ATSUNextCursorPosition` (page 136) and `ATSUPreviousCursorPosition` (page 138) to calculate the next and previous cursor positions.

Note that you may not be able to move the cursor 2-bytes, since doing so might place the cursor in the middle of a surrogate pair.

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 136). For right-to-left text, `ATSURightwardCursorPosition` has the same effect as calling `ATSUPreviousCursorPosition` (page 138).

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSULeftwardCursorPosition

Obtains the edge offset corresponding to the left of the high caret position based on the type of cursor movement.

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

`iTextLayout`	A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.
`iOldOffset`	A value of type `UniCharArrayOffset` (page 193). Pass the edge offset corresponding to the initial cursor position. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If the edge offset is outside the text buffer, `ATSULeftwardCursorPosition` returns the result code `kATSUInvalidTextRangeErr`.
`iMovementType`	A value of type `ATSUCursorMovementType`. Pass the unit distance that the cursor was moved. See "Cursor Movement Constants" (page 204) for a description of possible values. You must pass a value between 2 bytes and a word in length.
`oNewOffset`	A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the edge offset corresponding to the new cursor position. This offset may be outside the text buffer.
function result	A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in styled data from an invalid cache. In this case, either 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 231).

DISCUSSION

The `ATSULeftwardCursorPosition` function determines the edge offset in backing store memory corresponding to the left of the high caret position based on the type of cursor movement specified in the `iMovementType` parameter. You should call `ATSULeftwardCursorPosition` and the function `ATSURightwardCursorPosition` (page 140) when the initial edge offset is at a line boundary. At a line boundary, the caret is split into a high and low caret. If the initial edge offset is not at a line direction boundary, you should instead call the functions `ATSUNextCursorPosition` (page 136) and `ATSUPreviousCursorPosition` (page 138) to calculate the next and previous cursor positions.

Note that you may not be able to move the cursor 2-bytes, since doing so might place the cursor in the middle of a surrogate pair.

Except in the case of Indic text (and other cases where the font rearranges the glyphs), for left-to-right text, `ATSULeftwardCursorPosition` has the same effect as calling `ATSUPreviousCursorPosition` (page 138). For right-to-left text, `ATSULeftwardCursorPosition` has the same effect as calling `ATSUNextCursorPosition` (page 136).

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

Handling Text Insertion and Deletion

ATSUI provides the following functions for handling text insertion and deletion:

- `ATSUTextDeleted` (page 143) indicates the location in physical memory of deleted text.
- `ATSUTextInserted` (page 145) indicates the location in physical memory of inserted text.

ATSUTextDeleted

Indicates the location in physical memory of deleted text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iDeletedRangeStart`
A value of type `UniCharArrayOffset` (page 193). Pass the edge offset corresponding to the beginning of the deleted text. You may pass a value outside the range of text being operated on if the deletion occurs outside the range. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iDeletedRangeLength` parameter. If the offset is outside the text buffer, `ATSUTextDeleted` returns the result code `kATSUInvalidTextRangeErr`.

`iDeletedRangeLength`
A value of type `UniCharCount` (page 194). Pass the length of the deleted text. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iDeletedRangeStart` parameter. If the range is outside the text buffer, `ATSUTextDeleted` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUTextDeleted` function shortens or removes the style run containing the deletion point and the total length of the text range by the amount of the text deletion. If the deletion point is between two style runs, the first style run is removed. It also removes any soft line breaks that fall within the deleted text. `ATSUTextDeleted` then updates drawing caches.

`ATSUTextDeleted` does not change the memory location or the edge offset of the text. It shifts the text after the deleted text by the appropriate offset (`iDeletedRangeStart` + `iDeletedRangeLength`).

You are responsible for making sure that the corresponding text is deleted from the text buffer. You are also responsible for disposing of the memory associated

with style runs that have been removed by calling the function `ATSUDisposeStyle` (page 30).

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUTextInserted

Indicates the location in physical memory of inserted text.

```
OSStatus ATSUTextInserted (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iInsertionLocation,
                    UniCharCount iInsertionLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iInsertionLocation`
A value of type `UniCharArrayOffset` (page 193). Pass the edge offset corresponding to the beginning of the inserted text. You may pass a value outside the range of text being operated on if the insertion occurs outside the range. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If the offset is outside the text buffer, `ATSUTextInserted` returns the result code `kATSUInvalidTextRangeErr`.

`iInsertionLength`
A value of type `UniCharCount` (page 194). Pass the length of the inserted text. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iInsertionLocation` parameter. If the range is outside the text buffer, `ATSUTextInserted` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The `ATSUTextInserted` function extends the style run containing the insertion point and the total length of the text range by the amount of the text insertion. If the insertion point is between two style runs, the first style run is extended to include the new text. `ATSUTextInserted` then updates drawing caches.

`ATSUTextInserted` does not insert style runs or line breaks; if you wish to do so, call the functions `ATSUSetRunStyle` (page 118) and `ATSUSetSoftLineBreak` (page 159), respectively.

`ATSUTextInserted` does not change the memory location or the edge offset of the text. It shifts the text after the inserted text by the appropriate offset (`iInsertionLocation` + `iInsertionLength`).

You are responsible for making sure that the corresponding text is inserted into the text buffer.

VERSION NOTES

Available beginning with ATSUI 1.0.

Measuring Typographic and Image Bounds

- `ATSUGetGlyphBounds` (page 146) obtains the typographic glyph bounds of a final laid-out line.
- `ATSUMeasureText` (page 150) obtains the typographic bounding rectangle of a line of text prior to final line layout.
- `ATSUMeasureTextImage` (page 153) obtains the standard bounding rectangle of a final laid-out line.

ATSUGetGlyphBounds

Obtains the typographic glyph bounds of a final laid-out line.

```
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 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iTextBasePointX`

A value of type `ATSUTextMeasurement` (page 192). Pass the x-coordinate of the origin of the line (in the current graphics port) whose typographic glyph bounds you want to calculate. Pass 0 if you just want the dimensions of the bounds with respect to one another, not their actual onscreen position. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want the dimensions of the bounds relative to the current pen location in the current graphics port.

`iTextBasePointY`

A value of type `ATSUTextMeasurement` (page 192). Pass the y-coordinate of the origin of the line (in the current graphics port) whose typographic glyph bounds you want to calculate. Pass 0 if you just want the dimensions of the bounds with respect to one another, not their actual onscreen position. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want the dimensions of the bounds relative to the current pen location in the current graphics port.

`iBoundsCharStart`

A value of type `UniCharArrayOffset` (page 193). Pass the edge offset corresponding to the beginning of the range of text whose typographic glyph bounds you want to calculate. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iBoundsCharLength` parameter. If the offset is outside the text buffer, `ATSUGetGlyphBounds` returns the result code `kATSUInvalidTextRangeErr`.

`iBoundsCharLength`
A value of type `UniCharCount` (page 194). Pass the length of the range of text whose onscreen typographic glyph bounds you want to obtain. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iBoundsCharStart` parameter. If the range of text is outside the text buffer, `ATSUGetGlyphBounds` returns the result code `kATSUInvalidTextRangeErr`.

`iTypeOfBounds` The type of origin used in calculating the dimensions of the typographic bounds. See "Glyph Bound Constants" (page 206) for a description of possible values.

`iMaxNumberOfBounds`
The maximum number of enclosing trapezoids you want passed back. Typically, this is equivalent to the number of bounds in the `oGlyphBounds` array. To determine this value, see the discussion below.

`oGlyphBounds` An array of structures of type `ATSTrapezoid` (page 184). Before calling `ATSUGetGlyphBounds`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains the enclosing trapezoid(s) representing the typographic bounds of a range of text. If the specified range of text encloses directional boundaries, `ATSUGetGlyphBounds` will pass back multiple trapezoids defining these regions. In ATSUI 1.1, the maximum number of enclosing trapezoids that can be returned in 31; in ATSUI 1.2, the maximum number is 127. You cannot pass `NULL` for this parameter.

`oActualNumberOfBounds`
A pointer to a count. On return, the actual number of enclosing trapezoids bounding 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 231).

DISCUSSION

The `ATSUGetGlyphBounds` function returns the rotated final typographic bounds in multiple trapezoids, if needed, taking into account line rotation, flushness, and justification. It passes back the enclosing trapezoid(s) that represent the typographic glyph bounds of a range of text.

To obtain the typographic bounding rectangle around a line of text prior to line layout, call the function `ATSUMeasureText` (page 150). To calculate the standard bounding rectangle around a final laid-out line, call the function `ATSUMeasureTextImage` (page 153).

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.

The coordinates of each trapezoid are offset by the amount specified 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 the 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.

You should call `ATSUGetGlyphBounds` to do your own text highlighting, using the fractional origin (instead of the device origin) for the width of the highlight.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUMeasureText

Obtains the typographic bounding rectangle of a line of text prior to final line layout.

```
OSStatus ATSUMeasureText (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iLineStart,
                    UniCharCount iLineLength,
                    ATSUTextMeasurement *oTextBefore,
                    ATSUTextMeasurement *oTextAfter,
                    ATSUTextMeasurement *oAscent,
                    ATSUTextMeasurement *oDescent);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset that corresponds to the beginning of the range of text whose typographic bounding rectangle you want to obtain. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iLineLength` parameter. If the offset is outside the text buffer, `ATSUMeasureText` returns the result code `kATSUInvalidTextRangeErr`.

`iLineLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text whose typographic bounding rectangle you want to obtain. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iLineStart` parameter. If the range of text is outside the text buffer, `ATSUMeasureText` returns the result code `kATSUInvalidTextRangeErr`.

`oTextBefore`	A pointer to a value of type `ATSUTextMeasurement` (page 192). On return, the startpoint of the typographic bounding rectangle relative to the origin of the line in the current graphics port, taking into account cross-stream shifting. Assuming horizontal text, this corresponds to the left side of the bounding rectangle.
`oTextAfter`	A pointer to a value of type `ATSUTextMeasurement` (page 192). On return, the endpoint of the typographic bounding rectangle relative to the origin of the line in the current graphics port, taking into account cross-stream shifting. Assuming horizontal text, this corresponds to the right side of the bounding rectangle.
`oAscent`	A pointer to a value of type `ATSUTextMeasurement` (page 192). On return, the ascent of the typographic bounding rectangle relative to the origin of the line in the current graphics port, taking into account cross-stream shifting. Assuming horizontal text, this corresponds to the top side of the bounding rectangle.
`oDescent`	A pointer to a value of type `ATSUTextMeasurement` (page 192). On return, the descent of the typographic bounding rectangle relative to the origin of the line in the current graphics port, taking into account cross-stream shifting. Assuming horizontal text, this corresponds to the bottom side of the bounding rectangle.
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 231).

DISCUSSION

The `ATSUMeasureText` function obtains the typographic bounding rectangle of a line of text, ignoring any previously set line attributes like rotation, flushness, justification, ascent, and descent. `ATSUMeasureText` obtains a single rectangle representing the typographic bounds. The coordinates of the rectangle are independent of the rendering device used to display the text.

To obtain the typographic bounds of a line after it is laid out, call the function `ATSUGetGlyphBounds` (page 146). To calculate the standard bounding rectangle of a laid-out line, call the function `ATSUMeasureTextImage` (page 153).

Before calculating the typographic bounding rectangle enclosing 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.

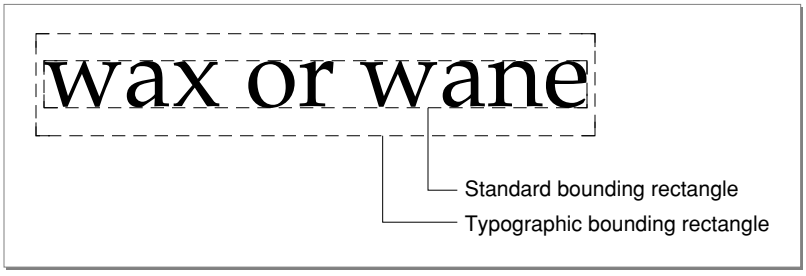
`ATSUMeasureText` will not change or invalidate existing laid-out lines. Its main purpose is to give you feedback on the typographical extrema of a range of characters so you can position lines determine line breaks.

If the range of text is less than a line, `ATSUMeasureText` treats it as a line and ignores any previously set text layout attributes. If the range of text is exactly a line in length, `ATSUMeasureText` performs post-compensation actions on it. If the range of text extends beyond a line, `ATSUMeasureText` ignores soft line breaks (that is, it treats the text as a line).

You should call `ATSUMeasureText` to determine line breaks and other text fitting inquiry like the leading line spacing to impose on a line. It can also be used to determine where the line origin should go. 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.

Figure 2-1 (page 153) illustrates the difference between the typographic bounding rectangle passed back by `ATSUMeasureText` and the standard bounding rectangle passed back by `ATSUMeasureTextImage` (page 153). The standard 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 rectangle



SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUMeasureTextImage

Obtains the standard bounding rectangle of a final laid-out line.

```
OSStatus ATSUMeasureTextImage (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iLineOffset,
                    UniCharCount iLineLength,
                    ATSUTextMeasurement iLocationX,
                    ATSUTextMeasurement iLocationY,
                    Rect *oTextImageRect);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset that corresponds to the beginning of the range of text whose standard bounding rectangle you want to obtain. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iLineLength` parameter. If the offset is outside the text buffer, `ATSUMeasureTextImage` returns the result code `kATSUInvalidTextRangeErr`.

`iLineLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text whose standard bounding rectangle you want to obtain. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iLineStart` parameter. If the range of text is outside the text buffer, `ATSUMeasureTextImage` returns the result code `kATSUInvalidTextRangeErr`.

`iLocationX` A value of type `ATSUTextMeasurement` (page 192). Pass the x-coordinate of the origin of the line (in the current graphics port) whose image bounds you want to calculate. Pass 0 if you just want the dimensions of the bounds with respect to one another, not their actual onscreen position. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want the dimensions of the bounds relative to the current pen location in the current graphics port.

`iLocationY` A value of type `ATSUTextMeasurement` (page 192). Pass the y-coordinate of the origin of the line (in the current graphics port) whose image bounds you want to calculate. Pass 0 if you just want the dimensions of the bounds with respect to one another, not their actual onscreen position. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want the dimensions of the bounds relative to the current pen location in the current graphics port.

oTextImageRect A pointer to a Rect. On return, the structure contains the dimensions of the enclosing rectangle representing the image bounds 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 231).

DISCUSSION

The ATSUMeasureTextImage function obtains the standard bounding rectangle, or image bounds, of a line of text. ATSUMeasureTextImage takes into account line rotation, alignment, and justification, as well as other characteristics that affect layout like hanging punctuation. The rectangle passed back in oTextImageRect is the same one used by the function ATSUDrawText (page 163) to draw a line 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.

To obtain the typographic bounds of a line after it is laid out, call the function ATSUGetGlyphBounds (page 146). To calculate the typographic bounds of a line before it is laid out, call the function ATSUMeasureText (page 150).

If you want to instead obtain the typographic bounds of a final laid-out line, call the function ATSUGetGlyphBounds (page 146). To calculate the standard bounding rectangle around a final laid-out line, call the function ATSUMeasureTextImage (page 153). For an illustration of the difference between a standard and typographic bounding rectangle, see Figure 2-1 (page 153).

Before calculating the standard bounding rectangle enclosing a range of text, ATSUMeasureTextImage 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,

ATSUMeasureTextImage 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, ATSUMeasureTextImage assigns these characters to the first style run it can find. If there no style run at the end of the range of text, ATSUMeasureTextImage assigns the remaining characters to the last style run it can find.

The height of the standard bounding rectangle is determined by the natural line ascent and descent calculated for the line. ATSUMeasureTextImage ignores the previously set line ascent and descent values for the line it is measuring. However, it uses other text layout attributes set for the line determine character layout. If no attributes have been set for the line, ATSUMeasureTextImage uses attributes set for the text layout object.

If the line is rotated, the sides of the passed back rectangle are parallel to the coordinate axes and encompass the rotated line. You should pass the standard bounding rectangle of a line of text to the function EraseRect to ensure erase all the text. In contrast, the typographic bounding rectangle passed back by ATSUMeasureText reflects an unrotated line.

The coordinates you specify in iLocationX and iLocationY are the same values used by ATSUDrawText (page 163) for the line of text to be measured.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

Manipulating Line Breaks

ATSUI provides the following functions for manipulating line breaks:

- ATSUBreakLine (page 157) suggests and optionally sets a soft line break in a line.
- ATSUSetSoftLineBreak (page 159) sets a soft line break.
- ATSUGetSoftLineBreaks (page 160) obtains all soft line breaks in a range of text.

- `ATSUClearSoftLineBreaks` (page 162) removes previously set soft line breaks in a range of text.

ATSUBreakLine

Suggests and optionally sets a soft line break in a line.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iLineStart` A value of type `UniCharArrayOffset` (page 193). The first time you call `ATSUBreakLine`, pass the edge offset of the beginning of the range of text whose soft line break(s) you wish to calculate. On subsequent calls, pass the offset passed back in the previous call to `ATSUBreakLine`. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). If the offset is outside the text buffer, `ATSUBreakLine` returns the result code `kATSUInvalidTextRangeErr`.

`iLineWidth` A value of type `ATSUTextMeasurement` (page 192). Pass a value greater than 0 that represents the width of the line, beginning at `iLineStart`. To indicate the line width previously set for the line, pass the constant `kATSUUseLineControlWidth`, described in "Line Layout Width Constant" (page 214).

`iUseAsSoftLineBreak`
A `Boolean` value. Pass `true` if you want `ATSUBreakLine` to set the recommended soft line break; otherwise, `false`.

`oLineBreak` A pointer to a value of type `UniCharArrayOffset` (page 193). On return, the recommended soft line break. If the line break occurs in the middle of the word, `ATSUBreakLine` passes back the location of the soft line break and returns the result code

kATSULineBreakInWord. If ATSUBreakLine passes back the same value as you passed in iLineStart, either iLineWidth is not big enough for ATSUBreakLine to perform line breaking, or there is another input error. You must check for this condition, because ATSUBreakLine will not return an error in this case.

function result A result 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 231).

DISCUSSION

The ATSUBreakLine function suggests and optionally sets a soft line break in a line based on the line width specified in iLineWidth.

Before calculating soft line breaks, ATSUBreakLine turns off any previously set line justification, rotation, width alignment, descent, and ascent values and treats the text as a single line. It 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.

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 glyph layout, it will pass back the previously set soft line breaks. You should call ATSUBreakLine repeatedly until it does not find any more soft line breaks.

If ATSUBreakLine does not encounter a hard line break, it uses the line width you specify to determine how many characters can fit on a line. If you pass true for iUseAsSoftLineBreak, it uses the soft line break it calculated 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.

To implement word break hyphenation, pass false to the iUseAsSoftLineBreak parameter and call the function ATSUSetSoftLineBreak (page 159).

ATSUBreakLine suggests a soft line break each time it encounters a hard line break character like a carriage return, line feed, form feed, line separator, or paragraph separator.

SPECIAL CONSIDERATIONS

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

ATSUSetSoftLineBreak

Sets a soft line break.

```
OSStatus ATSUSetSoftLineBreak (
                    ATSUTextLayout iTextLayout,
                    UniCharArrayOffset iLineBreak);
```

iTextLayout	A reference of type ATSUTextLayout (page 191). Pass a reference to an initialized text layout object. You cannot pass NULL for this parameter.
iLineBreak	A value of type UniCharArrayOffset (page 193). Pass the edge offset that corresponds to the soft line break you want to set. To indicate the beginning of the text buffer, pass the constant kATSUFromTextBeginning, described in "Text Offset Constant" (page 231). If the offset is outside the text buffer, ATSUSetSoftLineBreak returns the result code kATSUInvalidTextRangeErr.
function result	A result code. See "Result Codes" (page 231).

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 should call ATSUSetSoftLineBreak to implement word break hyphenation. If you do not want to set line breaks, call the function ATSUBreakLine (page 157) and pass true for the iUseAsSoftLineBreak parameter.

Before calculating soft line breaks, `ATSUSetSoftLineBreak` turns off any previously set line justification, rotation, width alignment, descent, and ascent values and treats the text as a single line. It 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, `ATSUSetSoftLineBreak` 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, `ATSUSetSoftLineBreak` assigns these characters to the first style run it can find. If there no style run at the end of the range of text, `ATSUSetSoftLineBreak` assigns the remaining characters to the last style run it can find.

After calling `ATSUSetSoftLineBreak`, you should call the function `ATSUMeasureText` (page 150) to measure the text.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetSoftLineBreaks

Obtains all soft line breaks in 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 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iRangeStart`	A value of type `UniCharArrayOffset` (page 193). Pass the edge offset of the beginning of the range of text whose soft line break positions you want to obtain. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iRangeLength` parameter. If the offset is outside the text buffer, `ATSUGetSoftLineBreaks` returns the result code `kATSUInvalidTextRangeErr`.
`iRangeLength`	A value of type `UniCharCount` (page 194). Pass the length of the range of text whose soft line break positions you want to obtain. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iRangeStart` parameter. If the range of text is outside the text buffer, `ATSUGetSoftLineBreaks` returns the result code `kATSUInvalidTextRangeErr`.
`iMaximumBreaks`	The maximum number of soft line breaks you want to obtain. Typically, this is equivalent to the number of elements in the `oBreaks` array. To determine this value, see the discussion below.
`oBreaks`	An array of values of type `UniCharArrayOffset` (page 193). Before calling `ATSUGetSoftLineBreaks`, pass a pointer to memory that you have allocated for this array. If you are uncertain of how much memory to allocate, see the discussion below. On return, the array contains all the soft line breaks in the range of text.
`oBreakCount`	A pointer to a count. On return, the actual number of soft line breaks in the 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 231).

DISCUSSION

The `ATSUGetSoftLineBreaks` function obtains all the soft line breaks that have been set in a range of text.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUClearSoftLineBreaks

Removes previously set soft line breaks in a range of text.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iRangeStart` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset of the beginning of the range of text whose soft line break positions you want to remove. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iRangeLength` parameter. If the offset is outside the text buffer, `ATSUClearSoftLineBreaks` returns the result code `kATSUInvalidTextRangeErr`.

`iRangeLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text whose soft line break positions you want to remove. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iRangeStart` parameter. If the range of text is outside the text buffer, `ATSUClearSoftLineBreaks` returns the result code `kATSUInvalidTextRangeErr`.

function result A result code. See "Result Codes" (page 231).

VERSION NOTES

Available beginning 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 screen location.

ATSUDrawText

Draws a range of text at a specified screen location.

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

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iLineOffset` A value of type `UniCharArrayOffset` (page 193). Pass the edge offset that corresponds to the beginning of the range of text that you want to render. If the range of text spans multiple lines, you should call `ATSUDrawText` for each line and pass the offset of the beginning of the new line to draw. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iLineLength` parameter. If the offset is outside the text buffer, `ATSUDrawText` returns the result code `kATSUInvalidTextRangeErr`.

`iLineLength` A value of type `UniCharCount` (page 194). Pass the length of the range of text that you want to render. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iLineOffset` parameter. If the range of text is outside the text buffer, `ATSUDrawText` returns the result code `kATSUInvalidTextRangeErr`.

`iLocationX`	A value of type `ATSUTextMeasurement` (page 192). Pass the x-coordinate of the origin of the line (in the current graphics port) containing the range of text you want to draw. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want to draw relative to the current pen location in the current graphics port.
`iLocationY`	A value of type `ATSUTextMeasurement` (page 192). Pass the y-coordinate of the origin of the line (in the current graphics port) containing the range of text you want to draw. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want to draw relative to the current pen location in the current graphics port.
function result	A result code. The result code `kATSUInvalidCacheErr` indicates that an attempt was made to read in style data from an invalid cache. This may be because 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 231).

DISCUSSION

The `ATSUDrawText` function renders a range of text at a specified screen location. It uses the same transfer mode and resolution as those set in the graphics port. Text color is taken from the style object and the value in the graphics port is ignored. If the text color was not previously set in the style object, you will get black text, regardless of what was set in the graphics port.

Before drawing the line, `ATSUDrawText` turns off any previously set line justification, rotation, width alignment, descent, and ascent values and treats the text as a single line. It 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, `ATSUDrawText` 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, `ATSUDrawText` assigns these characters to the first style run it can find. If there no style run at the end of the range of text, `ATSUDrawText` assigns the remaining characters to the last style run it can find.

If you want 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. You should adjust the `iLineOffset` parameter to reflect the beginning of each line to be drawn.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

Highlighting and Unhighlighting Text

ATSUI provides the following functions for highlighting and unhighlighting text:

- `ATSUHighlightText` (page 165) highlights a range of text.
- `ATSUUnhighlightText` (page 168) removes highlighting from a range of text.
- `ATSUGetTextHighlight` (page 170) obtains the highlight region for a range of text.

ATSUHighlightText

Highlights a range of text.

```
OSStatus ATSUHighlightText (
                    ATSUTextLayout iTextLayout,
                    ATSUTextMeasurement iTextBasePointX,
                    ATSUTextMeasurement iTextBasePointY,
                    UniCharArrayOffset iHighlightStart,
                    UniCharCount iHighlightLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

iTextBasePointX

A value of type ATSUTextMeasurement (page 192). Pass the x-coordinate of the origin of the line (in the current graphics port) containing the range of text you want to highlight. Pass the constant kATSUUseGrafPortPenLoc, described in "Current Pen Location Constant" (page 203), if you want to highlight relative to the current pen location in the current graphics port.

iTextBasePointY

A value of type ATSUTextMeasurement (page 192). Pass the y-coordinate of the origin of the line (in the current graphics port) containing the range of text you want to highlight. Pass the constant kATSUUseGrafPortPenLoc, described in "Current Pen Location Constant" (page 203), if you want to highlight relative to the current pen location in the current graphics port.

iHighlightStart

A value of type UniCharArrayOffset (page 193). Pass the edge offset that corresponds to the beginning of the range of text that you want to highlight. If the range of text spans multiple lines, you should call ATSUHighlightText for each line and pass the offset of the beginning of the new line you want to highlight. To indicate the beginning of the text buffer, pass the constant kATSUFromTextBeginning, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass kATSUFromTextBeginning in this parameter and the constant kATSUToTextEnd in the iHighlightLength parameter. If the offset is outside the text buffer, ATSUHighlightText returns the result code kATSUInvalidTextRangeErr.

iHighlightLength

A value of type UniCharCount (page 194). Pass the length of the range of text that you want to highlight. To indicate the end of the text buffer, pass the constant kATSUToTextEnd, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass kATSUToTextEnd in this parameter and the constant kATSUFromTextBeginning in the iHighlightStart parameter. If the range of text is outside the 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. This may be because 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 231).

DISCUSSION

The `ATSUHighlightText` function highlights a range of text using the highlight information in the graphics port.

Before calculating the highlighting dimensions, `ATSUHighlightText` turns off any previously set line justification, rotation, width alignment, descent, and ascent values and treats the text as a single line. It 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, `ATSUHighlightText` 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, `ATSUHighlightText` assigns these characters to the first style run it can find. If there no style run at the end of the range of text, `ATSUHighlightText` assigns the remaining characters to the last style run it can find.

If you want to highlight a range of text that spans multiple lines, you should call `ATSUHighlightText` for each line of text that is being highlighted, even if all the lines are in the same text layout object. You should adjust the `iHighlightStart` parameter to reflect the beginning of each line to be highlighted.

You can extend highlighting across tab stops by setting the bits specified by the mask constants `kATSLineFillOutToWidth` and `kATSLineImposeNoAngleForEnds`, described in "Line Layout Option Mask Constants" (page 212).

`ATSUHighlightText` uses the previously set line ascent and descent values to calculate the height of the highlighted region. If these values have not been set for the line, `ATSUHighlightText` uses the line ascent and descent values set for the text layout object containing the line. If these are not set, it uses the default values.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUUnhighlightText

Removes highlighting from a range of text.

```
OSStatus ATSUUnhighlightText (
                    ATSUTextLayout iTextLayout,
                    ATSUTextMeasurement iTextBasePointX,
                    ATSUTextMeasurement iTextBasePointY,
                    UniCharArrayOffset iHighlightStart,
                    UniCharCount iHighlightLength);
```

`iTextLayout` A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iTextBasePointX`
A value of type `ATSUTextMeasurement` (page 192). Pass the x-coordinate of the origin of the line (in the current graphics port) containing the range of text that you want to unhighlight. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want to remove highlighting relative to the current pen location in the current graphics port.

`iTextBasePointY`
A value of type `ATSUTextMeasurement` (page 192). Pass the y-coordinate of the origin of the line (in the current graphics port) containing the range of text that you want to unhighlight. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want to remove highlighting relative to the current pen location in the current graphics port.

`iHighlightStart`
A value of type `UniCharArrayOffset` (page 193). Pass the edge offset that corresponds to the beginning of the range of text that you want to unhighlight. If the range of text spans multiple lines, you should call `ATSUUnhighlightText` for each line and pass

the offset of the beginning of the new line you want to unhighlight. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iHighlightLength` parameter. If the offset is outside the text buffer, `ATSUUnhighlightText` returns the result code `kATSUInvalidTextRangeErr`.

`iHighlightLength`

A value of type `UniCharCount` (page 194). Pass the length of the range of text that you want to unhighlight. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the `iHighlightStart` parameter. If the range of text is outside the text buffer, `ATSUUnhighlightText` 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. This may be because 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 231).

DISCUSSION

The `ATSUUnhighlightText` function removes highlighting from a specified range of text using the highlight information in the graphics port.

Before calculating the dimensions of the highlighting region to remove, `ATSUUnhighlightText` turns off any previously set line justification, rotation, width alignment, descent, and ascent values and treats the text as a single line. It 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, `ATSUUnhighlightText` 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, `ATSUUnhighlightText` assigns these characters to the first style run it can find. If there no style run at the end

of the range of text, `ATSUUnhighlightText` assigns the remaining characters to the last style run it can find.

If you want to remove highlighting from a range of text that spans multiple lines, you should call `ATSUUnhighlightText` for each line of text that is being unhighlighted, even if all the lines are in the same text layout object. You should adjust the `iHighlightStart` parameter to reflect the beginning of each line to be unhighlighted.

You can remove highlighting across tab stops by setting the bits specified by the mask constants `kATSLineFillOutToWidth` and `kATSLineImposeNoAngleForEnds`, described in "Line Layout Option Mask Constants" (page 212).

`ATSUUnhighlightText` uses the previously set line ascent and descent values to calculate the height of the highlight region to be removed. If these values have not been set for the line, `ATSUUnhighlightText` uses the line ascent and descent values set for the text layout object containing the line. If these are not set, it uses the default values.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUGetTextHighlight

Obtains 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 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.

`iTextBasePointX`

A value of type `ATSUTextMeasurement` (page 192). Pass the x-coordinate of the origin of the line (in the current graphics port) containing the range of text whose highlight region you want to obtain. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want to obtain the highlighting region relative to the current pen location in the current graphics port.

`iTextBasePointY`

A value of type `ATSUTextMeasurement` (page 192). Pass the y-coordinate of the origin of the line (in the current graphics port) containing the range of text whose highlight region you want to obtain. Pass the constant `kATSUUseGrafPortPenLoc`, described in "Current Pen Location Constant" (page 203), if you want to obtain the highlighting region relative to the current pen location in the current graphics port.

`iHighlightStart`

A value of type `UniCharArrayOffset` (page 193). Pass the edge offset that corresponds to the beginning of the range of text whose highlight region you want to determine. To indicate the beginning of the text buffer, pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231). To indicate the entire text buffer, pass `kATSUFromTextBeginning` in this parameter and the constant `kATSUToTextEnd` in the `iHighlightLength` parameter. If the offset is outside the text buffer, `ATSUGetTextHighlight` returns the result code `kATSUInvalidTextRangeErr`.

`iHighlightLength`

A value of type `UniCharCount` (page 194). Pass the length of the range of text whose highlight region you want to determine. To indicate the end of the text buffer, pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). To indicate the entire text buffer, pass `kATSUToTextEnd` in this parameter and the constant `kATSUFromTextBeginning` in the

iHighlightStart parameter. If the range of text is outside the text buffer, ATSUGetTextHighlight returns the result code kATSUInvalidTextRangeErr.

oHighlightRegion
: A handle of type RgnHandle. Before calling ATSUGetTextHighlight, create a region handle by calling the NewRgn function. On return, RgnHandle points to a RgnPtr which points to a Region structure. The 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. This may be because 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 231).

DISCUSSION

The ATSUGetTextHighlight function 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 rgnBBox field of the structure represents the bounding box around the entire area of discontiguous highlighting. In ATSUI 1.1, the maximum number of components that can be passed back is 31; in version 1.2, the maximum is 127.

ATSUGetTextHighlight uses the previously set line ascent and descent values to calculate the height of the highlight region. If these values have not been set for the line, ATSUGetTextHighlight uses the line ascent and descent values set for the

text layout object containing the line. If these are not set, it uses the default values.

SPECIAL CONSIDERATIONS

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

VERSION NOTES

Available beginning with ATSUI 1.0. In ATSUI 1.2, the maximum number of components that can be passed back is 127.

Performing Background Processing

ATSUI provides the following function for performing background processing:

- `ATSUIdle` (page 173) enables ATSUI to perform background processing.

ATSUIdle

Enables ATSUI to perform background processing.

```
OSStatus ATSUIdle (ATSUTextLayout iTextLayout);
```

`iTextLayout`	A reference of type `ATSUTextLayout` (page 191). Pass a reference to an initialized text layout object. You cannot pass `NULL` for this parameter.
function result	A result code. See "Result Codes" (page 231).

DISCUSSION

Although versions 1.0 and 1.1 of ATSUI do not implement background processing for text layout objects, you should call the `ATSUIdle` function at least once in your main event loop to support the implementation of background processing in future versions of ATSUI.

VERSION NOTES

Available beginning with ATSUI 1.0.

Controlling Memory Allocation

ATSUI provides the following functions for controlling memory allocation 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) makes a memory allocation setting current.
- `ATSUGetCurrentMemorySetting` (page 176) returns the current memory allocation setting.
- `ATSUDisposeMemorySetting` (page 177) disposes of a memory allocation setting.

ATSUCreateMemorySetting

Creates a memory setting.

```
OSStatus ATSUCreateMemorySetting (
                    ATSUHeapSpec iHeapSpec,
                    ATSUMemoryCallbacks *iMemoryCallbacks,
                    ATSUMemorySetting *oMemorySetting);
```

`iHeapSpec` A value of type `ATSUHeapSpec`. Pass a value that indicates you want ATSUI or your own application to control memory allocation in ATSUI. See "Heap Specification Constants" (page 208) for a description of possible values. If you pass the `kATSUUseSpecificHeap` constant, you must pass a pointer to a union that contains the correctly-prepared heap in the `heapToUse` field in the `iMemoryCallbacks` parameter. If you pass the `kATSUUseCallbacks` constant, you must pass a pointer to a `ATSUMemoryCallbacks` union that contains pointers to your application-defined functions in the `iMemoryCallbacks` parameter. If you pass the `kATSUUseCurrentHeap` or

kATSUUseAppHeap constant, you should pass a NULL pointer in the iMemoryCallbacks parameter. You must pass a valid value for this parameter.

iMemoryCallbacks
A pointer to a union of type ATSUMemoryCallbacks (page 189). Pass a pointer to a union that contains either pointers to your application-defined memory allocation functions or the heap that you want ATSUI to use when allocating memory.

oMemorySetting
A pointer to a reference of type ATSUMemorySetting (page 190). On return, the new memory allocation setting. To make this setting current, you must pass it to the function ATSUSetCurrentMemorySetting (page 176). You cannot pass NULL for this parameter.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

The ATSUCreateMemorySetting function enables you to specify whether you wish to perform memory allocations yourself or have ATSUI do so. If you want to control memory allocation in ATSUI, pass kATSUUseCallbacks in the iHeapSpec parameter and a pointer to a ATSUMemoryCallbacks union that contains pointers to your callback functions in the iMemoryCallbacks parameter.

After creating a memory setting, you must pass it to the function ATSUSetCurrentMemorySetting (page 176) 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 with 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.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUSetCurrentMemorySetting

Makes a memory setting current.

```
OSStatus ATSUSetCurrentMemorySetting (ATSUMemorySetting iMemorySetting);
```

`iMemorySetting`
A reference of type `ATSUMemorySetting` (page 190). Pass a reference to the memory setting 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 231).

DISCUSSION

After you create a memory setting by calling the function `ATSUCreateMemorySetting` (page 174), you must pass it to the `ATSUSetCurrentMemorySetting` function to ensure that it will be used in subsequent Memory Manager calls.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUGetCurrentMemorySetting

Returns the current memory allocation setting.

```
ATSUMemorySetting ATSUGetCurrentMemorySetting (
                    void);
```

function result A reference of type `ATSUMemorySetting` (page 190) that contains the current memory setting. If there is no current memory setting, `ATSUGetCurrentMemorySetting` returns `NULL`.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUDisposeMemorySetting

Disposes of a memory setting.

```
OSStatus ATSUDisposeMemorySetting (
                    ATSUMemorySetting iMemorySetting);
```

`iMemorySetting`
A reference of type `ATSUMemorySetting` (page 190). Pass a reference to the memory setting that you want to dispose of. If you dispose of the current memory setting, ATSUI will use the current heap and its own callback functions to perform memory allocation operations.

function result A result code. See "Result Codes" (page 231).

DISCUSSION

Before you dispose of a memory setting, 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.

VERSION NOTES

Available beginning with ATSUI 1.1.

Callbacks

- `ATSUCustomAllocFunc` (page 178) defines a pointer to a memory allocation callback function. Your callback function manages memory allocation operations typically handled by ATSUI, including which heap to use and how memory allocation should occur.
- `ATSUCustomGrowFunc` (page 179) defines a pointer to a memory reallocation callback function. Your callback function manages memory reallocation operations typically handled by ATSUI.

- `ATSUCustomFreeFunc` (page 180) defines a pointer to a memory deallocation callback function. Your callback function manages memory deallocation operations typically handled by ATSUI.

ATSUCustomAllocFunc

Defines a pointer to a memory allocation callback function. Your callback function manages memory allocation operations typically handled by ATSUI, including which heap to use and how memory allocation should occur.

```
typedef void * (*ATSUCustomAllocFunc)(void *refCon, ByteCount howMuch);
```

You would declare your function like this if you were to name it `MyATSUCustomAllocFunc`:

```
void * MyATSUCustomAllocFunc (
                    void *refCon,
                    ByteCount howMuch);
```

`refCon` A pointer to arbitrary data for use in your memory allocation callback function. ATSUI passes a pointer to data that your application previously supplied in the `memoryRefCon` field of the `ATSUMemoryCallbacks` (page 189) union.

`howMuch` The amount of memory (in bytes) that you need to allocate.

function result An untyped pointer to the beginning of the block of memory allocated by your callback function.

DISCUSSION

You can register your callback function by calling the function `ATSUCreateMemorySetting` (page 174) and passing the constant `kATSUUseCallbacks` in `iHeapSpec` and a pointer to the `ATSUMemoryCallbacks` (page 189) union in `iMemoryCallbacks`. You then supply a pointer of type `ATSUCustomAllocFunc` in the `Alloc` field of the `callbacks` structure of the `ATSUMemoryCallbacks` 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 beginning with ATSUI 1.1.

ATSUCustomGrowFunc

Defines a pointer to a memory reallocation callback function. Your callback function manages memory reallocation operations typically handled by ATSUI.

```
typedef void * (*ATSUCustomGrowFunc)(void *refCon, void *oldBlock,
                    ByteCount oldSize, ByteCount newSize);
```

You would declare your function like this if you were to name it `MyATSUCustomGrowFunc`:

```
void * MyATSUCustomGrowFunc (
                     void *refCon,
                     void *oldBlock,
                     ByteCount oldSize,
                     ByteCount newSize);
```

`refCon` A pointer to arbitrary data for use in your memory reallocation callback function. ATSUI passes a pointer to data that your application previously supplied in the `memoryRefCon` field of the `ATSUMemoryCallbacks` (page 189) union.

`oldBlock` An untyped pointer to the beginning of the block of memory you wish to reallocate. ATSUI passes this value to your application.

`oldSize` The size (in bytes) of the memory you wish to reallocate. ATSUI passes this value to your application to indicate the number of bytes of memory you should copy if you need to allocate memory for the grown block.

`newSize` The size (in bytes) of the new block of memory.

function result An untyped pointer to the beginning address of the block of memory reallocated by your `MyATSUCustomGrowFunc` function.

DISCUSSION

You can register your callback function by calling the function `ATSUCreateMemorySetting` (page 174) and passing the constant `kATSUUseCallbacks` in `iHeapSpec` and a pointer to the `ATSUMemoryCallbacks` (page 189) union in `iMemoryCallbacks`. You then supply a pointer of type `ATSUCustomGrowFunc` in the `Grow` field of the `callbacks` structure of the `ATSUMemoryCallbacks` union.

Note that your `MyATSUCustomGrowFunc` function is expected to return a pointer to the start of the allocated memory, unless it terminates in an application.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUCustomFreeFunc

Defines a pointer to your memory deallocation callback function. Your callback function manages memory deallocation operations typically handled by ATSUI.

```
typedef void (*ATSUCustomFreeFunc )(void *refCon, void *doomedBlock);
```

You would declare your function like this if you were to name it `MyATSUCustomFreeFunc`:

```
void MyATSUCustomFreeFunc (
                    void *refCon,
                    void *doomedBlock);
```

`refCon` A pointer to arbitrary data for use in your memory deallocation callback function. ATSUI passes a pointer to data that your application previously supplied in the `memoryRefCon` field of the `ATSUMemoryCallbacks` (page 189) union.

`doomedBlock` A pointer to the beginning of the block of memory that your callback function will deallocate.

function result The address of the block of memory freed by your `MyATSUCustomFreeFunc` function.

DISCUSSION

You can register your callback function by calling the function `ATSUCreateMemorySetting` (page 174) and passing the constant `kATSUseCallbacks` in `iHeapSpec` and a pointer to the `ATSUMemoryCallbacks` (page 189) union in `iMemoryCallbacks`. You then supply a pointer of type `ATSUCustomFreeFunc` in the `Free` field of the `callbacks` structure of the `ATSUMemoryCallbacks` union.

VERSION NOTES

Available beginning with ATSUI 1.1.

Data Types

- `ATSJustPriorityWidthDeltaOverrides` (page 182)
- `ATSTrapezoid` (page 184)
- `ATSUAttributeInfo` (page 185)
- `ATSUAttributeValuePtr` (page 186)
- `ATSUCaret` (page 186)
- `ATSUCustomAllocFunc` (page 178)
- `ATSUCustomGrowFunc` (page 179)
- `ATSUCustomFreeFunc` (page 180)
- `ATSUFontFeatureType` (page 187)
- `ATSUFontFeatureSelector` (page 188)
- `ATSUFontID` (page 188)
- `ATSUFontVariationAxis` (page 188)
- `ATSUFontVariationValue` (page 189)
- `ATSUMemoryCallbacks` (page 189)
- `ATSUMemorySetting` (page 190)
- `ATSUStyle` (page 191)

- ATSUTextLayout (page 191)
- ATSUTextMeasurement (page 192)
- ConstUniCharArrayPtr (page 192)
- UniChar (page 192)
- UniCharArrayHandle (page 193)
- UniCharArrayOffset (page 193)
- UniCharArrayPtr (page 194)
- UniCharCount (page 194)

ATSJustPriorityWidthDeltaOverrides

The ATSJustPriorityWidthDeltaOverrides type is an array of four width delta override structures of type ATSJustWidthDeltaEntryOverride, one for each of the four justification priorities. Each structure contains, for both the grow and shrink cases, limits to the amount of space that can be added or removed from both the right and left sides of each of the glyphs of a given justification priority.

The ATSJustPriorityWidthDeltaOverrides type can be used to set and get justification behavior and priority override weighting; see "Style Run Attribute Tag Constants" (page 217).

```
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 constants that indicate whether ATSUI should apply the limits defined in the `beforeGrowLimit` and `afterGrowLimit` fields. See "Justification Override Mask Constants" (page 287) for a description of possible values. These mask constants also control whether unlimited gap absorption should be applied to the priority of glyphs specified in the given width delta override structure. You can use these mask constants to selectively override the grow case only, while retaining default behavior for other cases.
`shrinkFlags`	Mask constants that indicate whether ATSUI should apply the limits defined in the `beforeShrinkLimit` and `afterShrinkLimit` fields. See "Justification Override Mask Constants" (page 287) for a description of possible values. These mask constants also control whether unlimited gap absorption should be applied to the priority of glyphs specified in the given width delta override structure. You can use these mask constants to selectively override the grow case only, while retaining default behavior for other cases.

VERSION NOTES

Available beginning with ATSUI 1.0

ATSTrapezoid

The `ATSTrapezoid` type is a structure that contains the coordinates of typographic bounding trapezoid(s) for a final laid-out line of text. The dimensions of the resulting trapezoid are relative to the coordinates specified in the `iTextBasePointX` and `iTextBasePointY` parameters. The width of the glyph bounds will be determined based on the value passed in the `iTypeOfBounds` parameter.

The function `ATSUGetGlyphBounds` (page 146) passes back an array of structures of type `ATSTrapezoid` to specify the enclosing trapezoid(s) of a final laid-out line of text. If the range of text spans directional boundaries, `ATSUGetGlyphBounds` will pass back multiple trapezoids defining these regions.

```
struct ATSTrapezoid {
    FixedPoint      upperLeft;
    FixedPoint      upperRight;
    FixedPoint      lowerRight;
    FixedPoint      lowerLeft;
};
```

Field descriptions

`upperLeft` A structure of type `FixedPoint` that contains the upper left coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

`upperRight` A structure of type `FixedPoint` that contains the upper right coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

`lowerRight` A structure of type `FixedPoint` that identifies the lower right coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

`lowerLeft` A structure of type `FixedPoint` that identifies the lower left coordinates (assuming a horizontal line of text) of the typographic glyph bounds.

VERSION NOTES

Available beginning with ATSUI 1.1. In ATSUI 1.1, `ATSUGetGlyphBounds` can pass back a maximum of 31 bounding trapezoids; in ATSUI 1.2, `ATSUGetGlyphBounds` can pass back as many as 127 bounding trapezoids.

ATSUAttributeInfo

The `ATSUAttributeInfo` type is a structure containing an attribute tag that identifies a particular attribute value and the data size (in bytes) of that attribute value.

Several ATSUI functions pass back an array of structures of this type. The function `ATSUGetAllAttributes` (page 40) passes back an array of `ATSUAttributeInfo` structures to represent the data sizes of all previously set style run attribute values and the corresponding style run attribute tags that identify those style run attribute values. The function `ATSUGetAllLayoutControls` (page 99) passes back an array of `ATSUAttributeInfo` structures to represent the data sizes of all previously set text layout attribute values for an entire text layout object and the corresponding text layout attribute tags that identify those text layout attribute values. The function `ATSUGetAllLineControls` (page 108) passes back an array of `ATSUAttributeInfo` structures to represent the data sizes of all previously set text layout attribute values for a single line in a text layout object and the corresponding text layout attribute tags that identify those text layout attribute values.

```
typedef struct {
    ATSUAttributeTag    fTag;
    ByteCount           fValueSize
}ATSUAttributeInfo;
```

Field descriptions

`fTag`	Identifies a particular style run or text attribute value. For a description of the Apple-defined style run and text layout attribute tag constants, see "Style Run Attribute Tag Constants" (page 217) and "Text Layout Attribute Tag Constants" (page 226), respectively.
`fValueSize`	The size (in bytes) of the style run or text layout attribute value.

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUAttributeValuePtr

The `ATSUAttributeValuePtr` type is a pointer to a style run or text layout attribute value of unknown size. Each attribute value pointed to by `ATSUAttributeValuePtr` is identified by an attribute tag and the size (in bytes) of the attribute value.

You pass the `ATSUAttributeValuePtr` type to functions that set or clear attribute values in style and text layout objects. The `ATSUAttributeValuePtr` type is passed back by functions that query style and text layout objects for their attribute values. You must dereference this pointer and cast it to the appropriate data type to obtain the actual attribute value.

```
typedef void *   ATSUAttributeValuePtr;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUCaret

The `ATSUCaret` type is a structure containing the coordinates of the caret that corresponds to an edge offset. The function `ATSUOffsetToPosition` (page 134) passes back two structures of type `ATSUCaret` to represent the caret position(s), relative to the origin of the line in the current graphics port, corresponding to a specified edge offset. If the edge offset is at a line boundary, the structure passed back in `oMainCaret` contains the starting and ending pen locations of the high caret, while `oSecondCaret` contains the low caret. If the offset is not at a line boundary, both parameters contain the same structure. This structure contains the starting and ending pen locations of the main caret.

```
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 beginning with ATSUI 1.0.

ATSUFontFeatureType

The `ATSUFontFeatureType` type describes the attributes of a particular font feature. For example, `kLigaturesType` represents the presence of ligatures in a font. You pass the `ATSUFontFeatureType` type to functions that set or clear font feature types in a style run. The `ATSUFontFeatureType` type is passed back by functions that obtain font feature types in a style run.

```
typedef Uint16 ATSUFontFeatureType;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUFontFeatureSelector

The `ATSUFontFeatureSelector` type represents the state (on or off) of a particular feature type. For example, `kRequiredLigaturesOnSelector` indicates that the ligature feature type is on. You pass the `ATSUFontFeatureSelector` type to functions that set or clear font feature selectors in a style run. The `ATSUFontFeatureSelector` type is passed back by functions that obtain font feature selectors in a style run.

```
typedef Uint16   ATSUFontFeatureSelector;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUFontID

The `ATSUFontID` type represents the unique identifier of a font to the font management system in ATSUI. You pass the `ATSUFontID` type with functions that set and obtain font information. The `ATSUFontID` type is passed back by functions that count fonts installed on a users's system. The `ATSUFontID` type can be also used to set and get the font in a style run; see "Style Run Attribute Tag Constants" (page 217).

```
typedef UInt32       ATSUFontID;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUFontVariationAxis

The `ATSUFontVariationAxis` type represents a stylistic attribute and the range of values that the font can use to express this attribute. You pass the `ATSUFontVariationAxis` type to functions that set or clear the font variations in a style run. The `ATSUFontVariationAxis` type is passed back by functions that query a style run for its font variations.

```
typedef FourCharCode        ATSUFontVariationAxis;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUFontVariationValue

The `ATSUFontVariationValue` type represents the range of values that the font can use for a particular font variation. You pass the `ATSUFontVariationValue` type to functions that set and clear font variations in a style run. The `ATSUFontVariationValue` type is passed back by functions that query a style run for font variations.

```
typedef Fixed    ATSUFontVariationValue;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUMemoryCallbacks

The `ATSUMemoryCallbacks` type is a union containing either pointers to your application-defined memory allocation functions or the heap that you want ATSUI to use when allocating memory. If you want to control memory allocation in ATSUI, you should supply pointers to your application-defined functions in the `callbacks` structure field of the union. If you want ATSUI to control memory allocation, you should supply the memory heap for ATSUI to use in the `heapToUse` field.

The `ATSUMemoryCallbacks` union is passed back by the function `ATSUCreateMemorySetting` (page 174) to represent the newly-created memory setting.

```
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, if they exist, and arbitrary data for use in your callback functions.

The `Alloc` field contains a pointer of type `ATSUCustomAllocFunc` (page 178) to your memory allocation callback function.

The `Free` field contains a pointer of type `ATSUCustomFreeFunc` (page 180) to your memory deallocation callback function.

The `Grow` field contains a pointer of type `ATSUCustomGrowFunc` (page 179) to your memory reallocation callback function.

The `memoryRefCon` field contains a pointer to arbitrary data for use in your callback functions.

`heapToUse` A pointer of type `THz`. Pass a pointer to the `Zone` structure that you want ATSUI to use for simple Memory Manager calls.

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUMemorySetting

The `ATSUMemorySetting` type is a reference to a private structure containing information about the current memory setting. You pass a reference of type `ATSUMemorySetting` reference to the functions `ATSUDisposeMemorySetting` (page 177) and `ATSUSetCurrentMemorySetting` (page 176) to either dispose of a memory setting or make one current. The function `ATSUGetCurrentMemorySetting` (page 176) passes back a reference of this type to indicate the current memory setting.

```
typedef struct OpaqueATSUMemorySetting*  ATSUMemorySetting;
```

VERSION NOTES

Available beginning with ATSUI 1.1.

ATSUStyle

The `ATSUStyle` type is a reference to a private structure containing information about a style object. The structure contains the style run attributes, font features, and font variations that have been set for a style run. You pass the `ATSUStyle` type to functions that set or clear style run attributes, font features, or font variations, as well as functions that copy style objects. The `ATSUStyle` type is passed back by functions that query a style object for its style run attributes, font features, or font variations, as well as by functions that create a style object.

```
typedef struct OpaqueATSUStyle* ATSUStyle;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUTextLayout

The `ATSUTextLayout` type is a reference to a private structure containing information about a text layout object. The structure contains the text layout attributes, soft line breaks, and style runs that have been set for a block of Unicode text. You pass the `ATSUTextLayout` type with functions that set or clear text layout attributes, soft line breaks, and style runs, as well as functions that copy text layout objects. The `ATSUTextLayout` type is passed back by functions that query a text layout object for its text layout attributes, soft line breaks, and style runs, as well as by functions that create a text layout object.

```
typedef struct OpaqueATSUTextLayout* ATSUTextLayout;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ATSUTextMeasurement

The `ATSUTextMeasurement` type represents the glyph bounds (exact outline metrics) of onscreen glyphs. You pass the `ATSUTextMeasurement` type to functions that draw, measure, hit-test, and obtain the glyph bounds for text. The `ATSUTextMeasurement` type can be also used to set and get imposed glyph width, line width, line ascent, and line descent; see "Style Run Attribute Tag Constants" (page 217) and "Text Layout Attribute Tag Constants" (page 226), respectively.

```
typedef Fixed       ATSUTextMeasurement;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

ConstUniCharArrayPtr

The `ConstUniCharArrayPtr` type is a pointer to the beginning of a text buffer. Note that ATSUI expects the buffer to contain Unicode text. Your application is responsible for allocating the memory associated with this pointer. You use the `ConstUniCharArrayPtr` type with the functions `ATSUCreateTextLayoutWithTextPtr` (page 86), `ATSUSetTextPointerLocation` (page 111), and `ATSUTextMoved` (page 117).

```
typedef const UniChar *ConstUniCharArrayPtr;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

UniChar

The `UniChar` type represents a 2-byte Unicode-encoded character. You can use the `UniChar` type with functions that operate on a text layout object's range of text.

```
typedef UInt16      UniChar;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

UniCharArrayHandle

The `UniCharArrayHandle` type is a handle that points to the beginning of a text buffer. Note that ATSUI expects the buffer to contain Unicode text. Your application is responsible for allocating the memory associated with this handle. ATSUI functions that need to access text referenced by this handle will return the handle to its original state (locked or unlocked) upon function completion. You use the `UniCharArrayHandle` type with the functions `ATSUCreateTextLayoutWithTextHandle` (page 88) and `ATSUSetTextHandleLocation` (page 113).

```
typedef UniCharArrayPtr * UniCharArrayHandle;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

UniCharArrayOffset

The `UniCharArrayOffset` type is the edge offset in backing store memory that corresponds to the beginning of the range of text within a text layout object to be operated on. Functions that perform layout operations on text do so within this range of text, not the entire text buffer. If you want the range of text to start at the beginning of the text buffer, you should pass the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231).

```
typedef UInt32          UniCharArrayOffset;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

UniCharArrayPtr

The `UniCharArrayPtr` type is a pointer to the beginning of a text buffer. Note that ATSUI expects the buffer to contain Unicode text. You can use the `UniCharArrayPtr` type with functions that operate on a range of text in a text layout object.

```
typedef UniChar *       UniCharArrayPtr;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

UniCharCount

The `UniCharCount` type is the number of characters in backing store memory that correspond to the length of the range of text within a text layout object to be operated on. Functions that perform layout operations on text do so within this range of text, not the entire text buffer. If you want the range of text to span the end of the text buffer, you should pass the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230).

```
typedef UInt32              UniCharCount;
```

VERSION NOTES

Available beginning with ATSUI 1.0.

Resource

ustl

The `'ustl'` type is a clipboard data block format for copying and pasting Unicode-encoded styled text. You can use a data block format of this type to copy and paste Unicode-encoded styled text between applications or within

your application. Note that you can store styled text information in any data format of type `'ustl'`. For the purposes of this document, the `'ustl'` type will be described as a resource.

As shown in Figure 2-2, version 1 of the `'ustl'` resource is composed of four basic elements:

- a header component
- flattened text layout data
- flattened style run data
- flattened style list data

Figure 2-2 Overview of a `'ustl'` resource

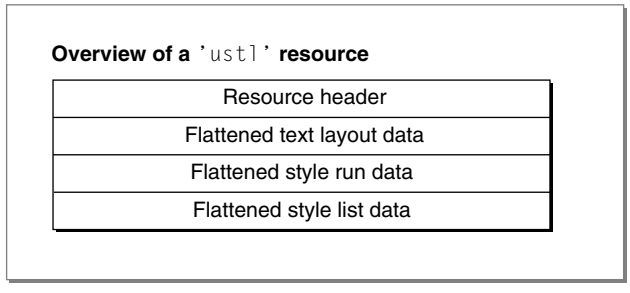


Figure 2-3 shows the header component of a `'ustl'` resource. The header contains the version and size of the `'ustl'` resource as well as offsets to flattened text layout, style run, and style list data, which you can specify in any order after the header section.

Figure 2-3 Header section of a 'ustl' resource

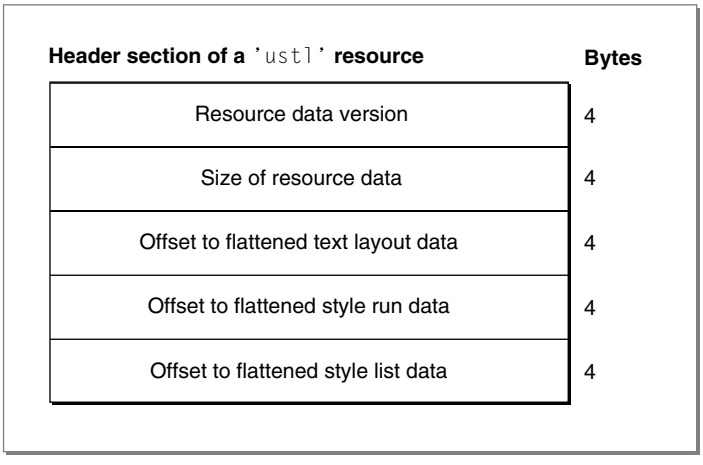
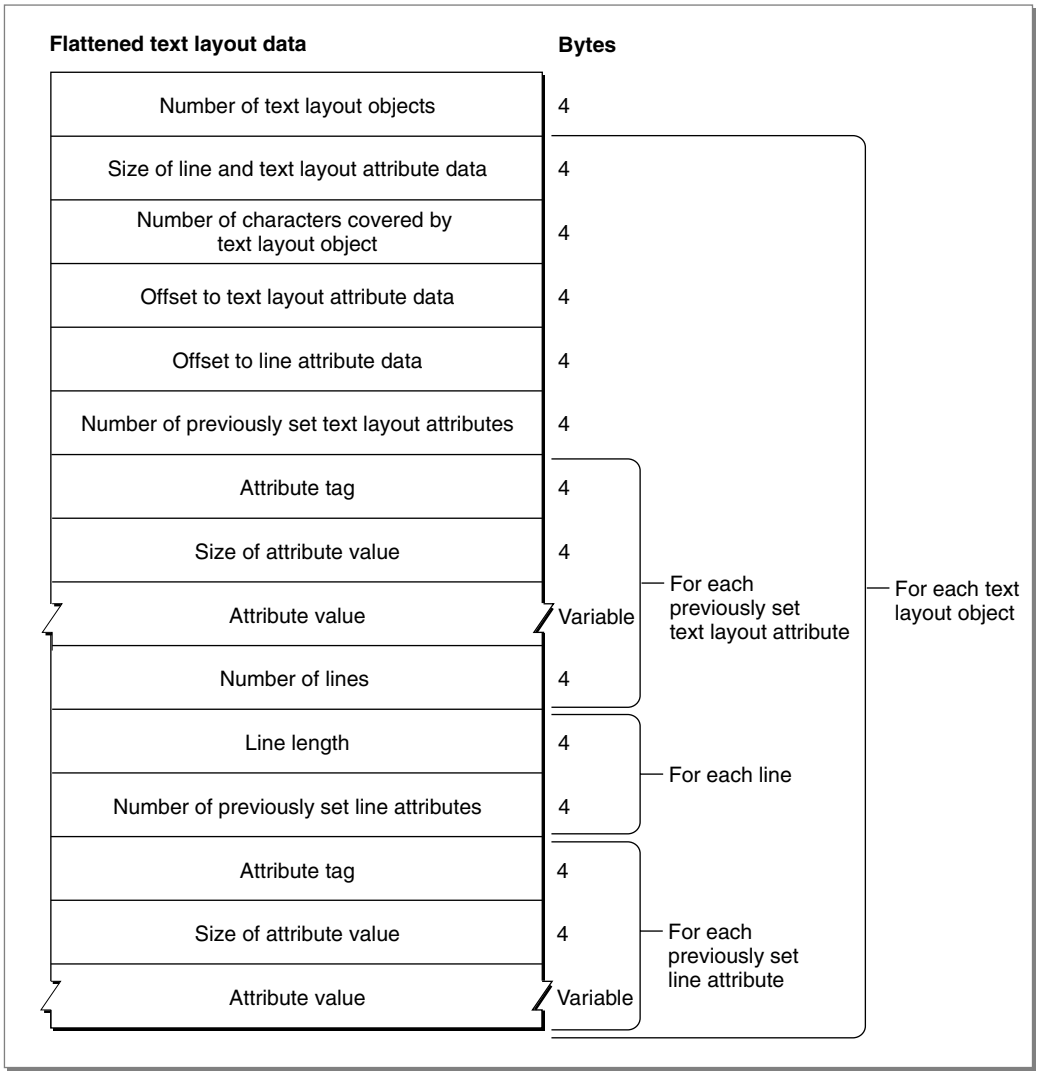


Figure 2-4 shows the format of flattened text layout data.

Figure 2-4 Flattened text layout data



As shown in Figure 2-4, the flattened text layout data is composed of the following elements:

- 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):

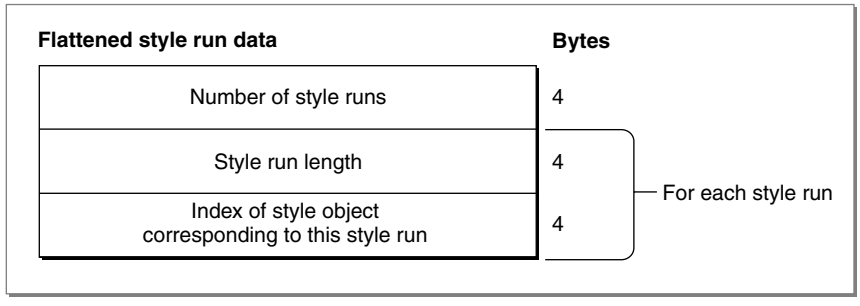
- the line length
- the number of previously set line attributes

(for each previously set line attribute):

- the attribute tag
- the size of the attribute value
- the actual attribute value (variable in length)

Figure 2-5 shows the format of flattened text layout data.

Figure 2-5 Flattened style run data



As shown in Figure 2-5, flattened style run data is composed of the following elements:

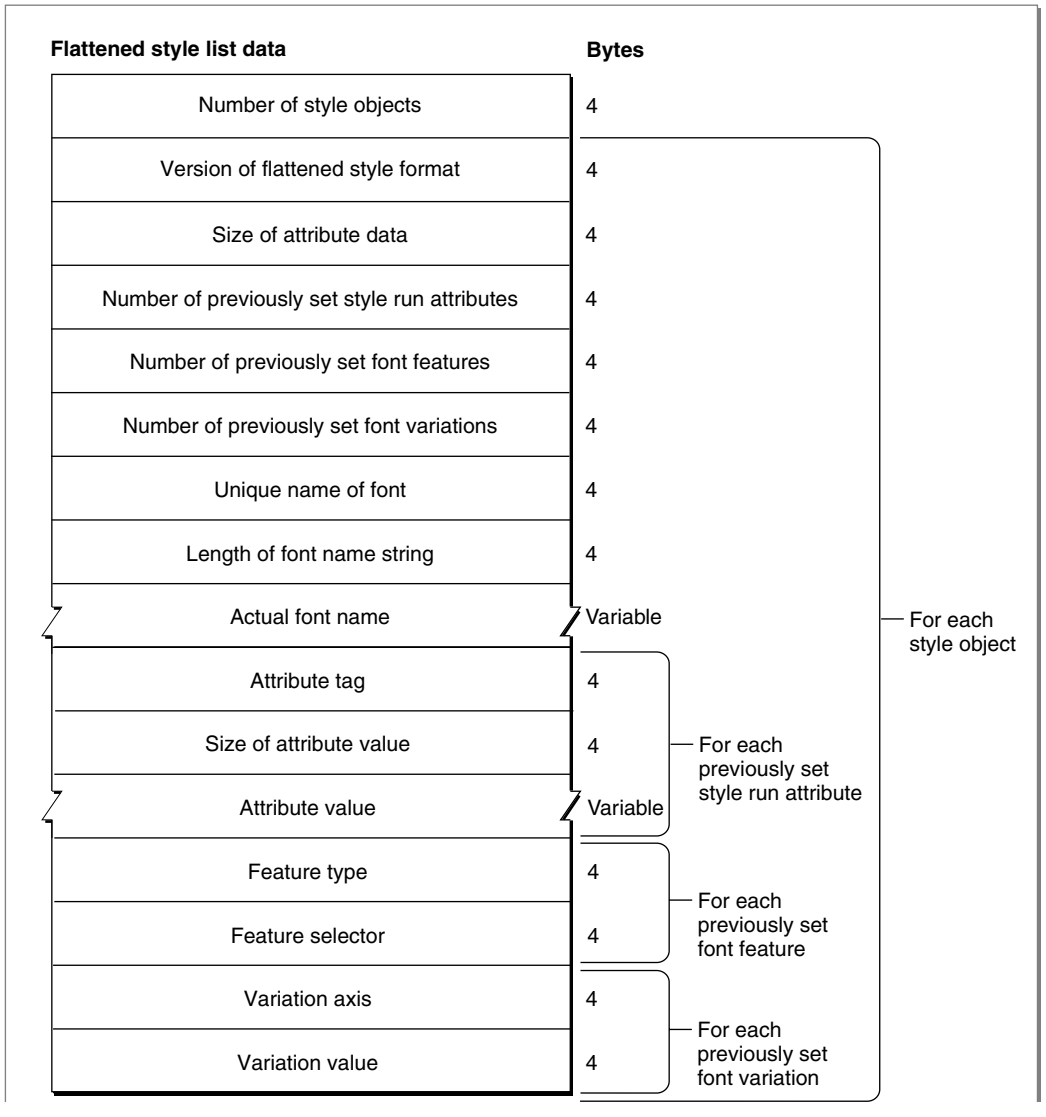
- the number of style runs
- an array of style run data

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

Figure 2-6 shows the format of flattened style list data.

Figure 2-6 Flattened style list data



As shown in Figure 2-6, flattened style list data is composed of the following elements:

- 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 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 1 of this resource, described above, is available with ATSUI 1.1. Version 0, described in *Apple Type Services for Unicode Imaging Reference* dated May 7, 1999, is available with ATSUI 1.0. You should use version 1 of the clipboard data format.

Constants

Clear All Constant

You can pass this constant to the following functions to remove previously set values from a style object: to `ATSUClearAttributes` (page 42) to remove style run attributes, to `ATSUClearFontFeatures` (page 49) to remove font features, and to `ATSUClearFontVariations` (page 54) to remove font variations.

You can also use this constant to remove previously set text layout attributes: to `ATSUClearLineControls` (page 109), to remove text layout attributes from a single line of a text layout object, and to `ATSUClearLayoutControls` (page 101) to remove text layout attributes from every line in a text layout object.

```
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 beginning with ATSUI 1.0.

Current Pen Location Constant

You can pass this constant to functions that operate on text layout objects to indicate that drawing, measuring, or hit-testing should be done relative to the current pen location in the current graphics port.

```
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 beginning with ATSUI 1.0.

Cursor Movement Constants

You can pass a constant of type `ATSUCursorMovementType` to the functions `ATSUNextCursorPosition` (page 136), `ATSUPreviousCursorPosition` (page 138), `ATSURightwardCursorPosition` (page 140), and `ATSULeftwardCursorPosition` (page 141) to represent the unit distance that the cursor has moved. These functions use this information to calculate the edge offset in backing store memory that corresponds to the resulting cursor position.

```
enum {
    kATSUByCharacter    = 0,
    kATSUByCluster      = 1,
    kATSUByWord         = 2
};
typedef int ATSUCursorMovementType;
```

Constant descriptions

`kATSUByCharacter` Indicates that the cursor has been moved 2 bytes (a Unicode character).

`kATSUByCluster` Indicates that the cursor has been moved by a 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 that the cursor has been moved by a word, as defined by Unicode. A word does not include trailing punctuation or white space.

VERSION NOTES

Available beginning with ATSUI 1.0.

Font Fallback Constants

You can pass a constant of type `ATSUFontFallbackMethod` to the function `ATSUSetFontFallbacks` (page 123) to specify the search options when a character(s) cannot be drawn with the assigned font. The function `ATSUGetFontFallbacks` (page 124) passes back one of these constants to indicate the search options you have previously specified.

```
enum {
    kATSUDefaultFontFallbacks           = 0,
    kATSULastResortOnlyFallback         = 1,
    kATSUSequentialFallbacksPreferred   = 2,
    kATSUSequentialFallbacksExclusive   = 3
};
typedef UInt16      ATSUFontFallbackMethod;
```

Constant descriptions

`kATSUDefaultFontFallbacks`
When a character cannot be drawn with the existing font, specifies that a replacement font should be identified using the following search order: (1) sequentially scanning the font list, and if no valid font is found (2) searching all valid fonts in the user's system. This is the default search order used by the functions `ATSUMatchFontsToText` (page 125) and `ATSUSetTransientFontMatching` (page 128).

`kATSULastResortOnlyFallback`
When a character cannot be drawn with the existing font, specifies that the replacement font should be the last resort font.

`kATSUSequentialFallbacksPreferred`
When a character cannot be drawn with the existing font, specifies that a replacement font should be identified using the following search order: (1) sequentially scanning the font list, and if no valid font is found (2) searching all valid fonts in the user's system, and if no valid font is found (3) using the last resort font.

`kATSUSequentialFallbacksExclusive`
When a character cannot be drawn with the existing font, specifies that a replacement font should be identified using the following search order: (1) sequentially scanning the

font list, and if no valid font is found (2) using the last resort font.

VERSION NOTES

Available beginning with ATSUI 1.0.

Glyph Bound Constants

Your application passes a glyph bounds constant in the `iTypeOfBounds` parameter of the function`ATSUGetGlyphBounds` (page 146) 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 150).

VERSION NOTES

Available beginning with ATSUI 1.1.

Glyph Direction Constants

You can use one of these constants to set or obtain glyph direction in a line of text or an entire text layout object, regardless of their font-specified direction; see the functions `ATSUSetLayoutControls` (page 96), `ATSUSetLineControls` (page 104), `ATSUGetLayoutControl` (page 98), and `ATSUGetLineControl` (page 106).

```
enum {
    kATSULeftToRightBaseDirection   = 0,
    kATSURightToLeftBaseDirection   = 1
};
```

Constant descriptions

`kATSULeftToRightBaseDirection`

Imposes left-to-right direction on glyphs in a line of horizontal text; for vertical text, imposes top-to-bottom direction.

`kATSURightToLeftBaseDirection`

Imposes right-to-left direction on glyphs in a line of horizontal text; for vertical text, imposes bottom-to-top direction.

VERSION NOTES

Available beginning with ATSUI 1.0.

Glyph Orientation Constants

You can pass a constant of type `ATSUVerticalCharacterType` to the functions `ATSUCountFontTracking` (page 68) and `ATSUGetIndFontTracking` (page 69) to specify the glyph orientation of font tracking settings, since font tracking settings differ depending upon glyph orientations.

You can also use one of these constants to set or obtain the glyph orientation of a style run; see the functions `ATSUSetAttributes` (page 37) and `ATSUGetAttribute` (page 38), respectively.

```
enum {
    kATSUStronglyHorizontal     = 0,
    kATSUStronglyVertical       = 1
};
typdef UInt16   ATSUVerticalCharacterType
```

Constant descriptions

`kATSUStronglyHorizontal`
Specifies nonrotated glyphs that are drawn with horizontal metrics.

`kATSUStronglyVertical`
Specifies that glyphs are rotated 90 degrees and are drawn with vertical metrics.

VERSION NOTES

Available beginning with ATSUI 1.0.

Heap Specification Constants

You can pass a constant of type `ATSUHeapSpec` to the function `ATSUCreateMemorySetting` (page 174) to specify whether you want ATSUI or your application to control memory allocation in ATSUI.

If you want ATSUI to perform memory allocation operations, pass the constant `kATSUUseSpecificHeap`. In this case, you must supply the correctly-prepared heap in the `heapToUse` field of the union `ATSUMemoryCallbacks` (page 189). If you want your own application-defined functions to control memory allocation, you must supply pointers to your application in the `callback` structure of the `ATSUMemoryCallbacks` 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`. 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 189) union.

`kATSUUseCallbacks`
: Indicates to use your application-defined functions to control memory allocation.

VERSION NOTES

Available beginning with ATSUI 1.1.

Invalid Font ID Constant

The functions `ATSUFONDtoFontID` (page 66), `ATSUFindFontFromName` (page 57), and `ATSUMatchFontsToText` (page 125) pass back this constant to indicate an invalid font ID. This constant is available with ATSUI 1.0.

```
enum {
    kATSUInvalidFontID      = 0
};
```

Constant description

`kATSUInvalidFontID` Indicates that the font ID is invalid.

VERSION NOTES

Available beginning with ATSUI 1.0.

Line Alignment Constants

You can use one of these constants to set or obtain the alignment of text relative to the margins in a line of text or in an entire text layout object; see the functions `ATSUSetLayoutControls` (page 96), `ATSUSetLineControls` (page 104), `ATSUGetLayoutControl` (page 98), and `ATSUGetLineControl` (page 106), 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 beginning 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 104) and ATSUSetLayoutControls (page 96), 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 104) and ATSUSetLayoutControls (page 96), 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 beginning 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 104) and ATSUSetLayoutControls (page 96), 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 beginning with ATSUI 1.0.

Line Layout Option Mask Constants

You can use a mask constant of type `ATSLineLayoutOptions` to set or obtain the line layout options in a line of text or an entire text layout object; see the functions `ATSUSetLineControls` (page 104) and `ATSUSetLayoutControls` (page 96), respectively.

```
enum {
    kATSLineNoLayoutOptions             = 0x00000000,
    kATSLineIsDisplayOnly               = 0x00000001,
    kATSLineHasNoHangers                = 0x00000002,
    kATSLineHasNoOpticalAlignment       = 0x00000004,
    kATSLineKeepSpacesOutOfMargin       = 0x00000008,
    kATSLineNoSpecialJustification      = 0x00000010,
    kATSLineLastNoJustification         = 0x00000020,
    kATSLineFractDisable                = 0x00000040,
    kATSLineImposeNoAngleForEnds        = 0x00000080,
    kATSLineFillOutToWidth              = 0x00000100,
    kATSLineTabAdjustEnabled            = 0x00000200,
    kATSLineAppleReserved               = (long)0xFFFFFC00
};
typedef UInt32      ATSLineLayoutOptions
```

Constant descriptions

`kATSLineNoLayoutOptions`
Indicates that no bits are set. Available with ATSUI 1.0.

`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. Available with ATSUI 1.0.

`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`. Available with ATSUI 1.0.

`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`. Available with ATSUI 1.0.

`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. Available with ATSUI 1.0.

`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`. Available with ATSUI 1.0.

`kATSLineLastNoJustification`

If the bit specified by this mask 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. Available with ATSUI 1.0.

`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 150). Available with ATSUI 1.1.

`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. Available with ATSUI 1.1.

`kATSLineFillOutToWidth`

If the bit specified by this mask is set, highlighting extends to both ends of a line, regardless of caret locations. It does not change caret locations. This is provided for your convenience to extend your highlighting to the full width of the line. Available with ATSUI 1.1.

`kATSLineTabAdjustEnabled`

If the bit specified by this mask is set, the tab character width will be automatically adjusted to fit the specified line width. You must set this bit to ensure that highlighting is done correctly across tab stops. To ensure this, you should also set the bit specified by the `kATSLineImposeNoAngleForEnds` mask constant. Available with ATSUI 1.2.

`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. ATSUI will return the `kATSUInvalidAttributeValueErr` result code if you set a reserved bit. Available with ATSUI 1.1.

VERSION NOTES

This enumeration is available beginning with ATSUI 1.0. Additional constants added with ATSUI 1.1 and 1.2.

Line Layout Width Constant

You can pass this constant to the function `ATSUBreakLine` (page 157) to indicate that `ATSUBreakLine` should use the line width previously set for that line to calculate the soft line break. If no line width has been set for the line, `ATSUBreakLine` will use the line width set for the text layout object.

```
enum {
    kATSUUseLineControlWidth        = 0x7FFFFFFFL
};
```

Constant description

`kATSUUseLineControlWidth`
Indicates that `ATSUBreakLine` should use the previously set line width attribute for the current line to determine how many characters can fit on the line. If no line width has been set for the line, `ATSUBreakLine` will use the line width set for the text layout object; if not set, `ATSUBreakLine` uses the default line width value.

VERSION NOTES

Available beginning 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 beginning with ATSUI 1.0.

Style Comparison Constants

The function `ATSUCompareStyles` (page 26) passes back a constant of type `ATSUStyleComparison` to indicate whether two style objects are the same, different, or a subset of one another.

```
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 beginning with ATSUI 1.0.

Style Run Attribute Tag Constants

You can pass a style run attribute tag constant of type `ATSUAttributeTag` to the functions `ATSUSetAttributes` (page 37), `ATSUGetAttribute` (page 38), `ATSUGetAllAttributes` (page 40), and `ATSUClearAttributes` (page 42) to identify the style run attribute value you wish to set, obtain, or remove from a style object. If you do not set a style run attribute, it will be set to its default value. Table C-1 (page 241) lists the Apple-defined style run attribute tags and their corresponding data type, size, and default values.

IMPORTANT

The `ATSUAttributeTag` type also defines text layout attribute tag constants; see "Text Layout Attribute Tag Constants" (page 226) for a description of these tags. Note that if you pass text layout attribute tags to functions that get, set, or remove style run attribute values, the function will return the result code `kATSUInvalidAttributeTagErr`. ▲

The following constant descriptions assume horizontal text. If you are setting or getting the style run attribute of vertical text, you should interpret these values 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,
    kATSUStyleTextLocatorTag            = 282L,
    kATSUMaxStyleTag                    = 283L,
    kATSUMaxATSUITagValue               = 65535L
};
typedef UInt32              ATSUAttributeTag;
```

Constant descriptions

`kATSUQDBoldfaceTag` Identifies the boldfaced text style attribute. You use this tag to set or get a value of type `Boolean` that indicates whether the text style is boldfaced (causes each glyph to be repeatedly drawn one bit to the right for extra thickness). A value of `true` indicates that text is boldfaced. If you do not set the attribute value corresponding to this tag, the default value is `false`, and the text style is plain. Available with ATSUI 1.0.

`kATSUQDItalicTag` Identifies the italicized text style attribute. You use this tag to set or get a value of type `Boolean` that indicates whether the text style is italicized (skews glyph bits above the baseline to the right, bits below the baseline to the left). A value of `true` indicates that text is italicized. If you do not set the attribute value corresponding to this tag, the default value is `false`, and the text style is plain. Available with ATSUI 1.0.

`kATSUQDUnderlineTag`

Identifies the underlined text style attribute. You use this tag to set or get a value of type `Boolean` that indicates whether the text style is 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 par. A value of `true` indicates that text is underlined. If you do not set the

attribute value corresponding to this tag, the default value is `false`, and the text style is plain. Available with ATSUI 1.0.

`kATSUQDCondensedTag`

Identifies the condensed text style attribute. You use this tag to set or get a value of type `Boolean` that indicates whether the text style is condensed (decreases the horizontal distance between all glyphs, including spaces, by the amount that the Font Manager determines is appropriate). A value of `true` indicates that text is condensed. If you do not set the attribute value corresponding to this tag, the default value is `false`, and the text style is plain. Available with ATSUI 1.0.

`kATSUQDExtendedTag` Identifies the extended text style attribute. You use this tag to set or get a value of type `Boolean` that indicates whether the text style is extended (increases the horizontal distance between all glyphs, including spaces, by the amount that the Font Manager determines is appropriate). A value of `true` indicates that text is extended. If you do not set the attribute value corresponding to this tag, the default value is `false`, and the text style is plain. Available with ATSUI 1.0.

`kATSUFontTag` Identifies the font ID attribute. You use this tag to set or obtain a value of type `ATSUFontID` (page 188) that uniquely identifies the font 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. Available with ATSUI 1.0.

`kATSUSizeTag` Identifies the text size attribute. You use this tag to set or obtain a value of type `Fixed` that represents text size (in typographic points, 72 per inch). 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). Available with ATSUI 1.0.

`kATSUColorTag` Identifies the text color attribute. You use this tag to set or obtain a value of type `RGBColor` that represents text color. If you do not set the attribute value corresponding to this tag, the default value is (0,0,0), and the text color will be black. Available with ATSUI 1.0.

`kATSULanguageTag` Identifies the language attribute. You use this tag to set or obtain a value of type `RegionCode` that represents the regional language and other region-dependent characteristics for glyphs. See "Region Code Constants" (page 249) for a description of possible values. 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)`. Available with ATSUI 1.0.

`kATSUVerticalCharacterTag`

Identifies the glyph orientation attribute. You use this tag to set or obtain a value of type `ATSUVerticalCharacterType` that represents glyph orientation; see "Glyph Orientation Constants" (page 207) for a description of possible values. Values can range from 0 to 2. 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`, and glyph orientation will be horizontal. Available with ATSUI 1.0.

`kATSUImposeWidthTag`

Identifies the imposed width weighting attribute. You use this tag to set or obtain a value of type `ATSUTextMeasurement` (page 192) that represents the amount to increase glyph width. Values can range from -1.0 to 1.0. A value of 0 (`kATSUNoImposedWidth`) indicates that you want to use font-defined imposed width. ATSUI ignores negative values. Positive values increase the font-defined imposed width. If you do not set the attribute value corresponding to this tag, the default value is `kATSUNoImposedWidth`, and

ATSUI uses the font-defined imposed width. Available with ATSUI 1.0.

`kATSUBeforeWithStreamShiftTag`

Identifies the with-stream shift (applied before each glyph) weighting attribute. You use this tag to set or obtain a value of type `Fixed` that represents the amount to increase or decrease space before glyphs in a style run. Values can range from -1.0 to 1.0. A value of 0 indicates that you want to use font-defined tracking. Negative values decrease space before glyphs, while positive values increase space before glyphs. If you do not set the attribute value corresponding to this tag, the bias is 0, and ATSUI uses the font-defined with-stream shift (applied before each glyph). Available with ATSUI 1.0.

`kATSUAfterWithStreamShiftTag`

Identifies the with-stream shift (applied after each glyph) weighting attribute. You use this tag to set or obtain a value of type `Fixed` that represents the amount to increase or decrease space after glyphs in a style run. Values can range from -1.0 to 1.0. A value of 0 indicates that you want to use font-defined tracking. Negative values decrease space after glyphs, while positive values increase space after glyphs. If you do not set the attribute value corresponding to this tag, the bias is 0, and ATSUI uses the font-defined with-stream shift (applied after each glyph). Available with ATSUI 1.0.

`kATSUCrossStreamShiftTag`

Identifies the cross-stream shift weighting attribute. You use this tag to set or obtain a value of type `Fixed` that represents the amount to raise or lower glyphs in a style run. Values can range from -1.0 to 1.0. A value of 0 indicates that you want to use font-defined tracking. Negative values shift glyphs downwards from the font-defined cross-stream shift, while positive values shift glyphs upwards from the font-defined cross-stream shift. If you do not set the attribute value corresponding to this tag, the bias is 0, and ATSUI uses the font-defined cross-stream shift. Available with ATSUI 1.0.

`kATSUTrackingTag` Identifies the tracking weighting attribute. You use this tag to set or obtain a value of type `Fixed` that represents the amount to bias font-defined tracking (that is, kerning

between all glyphs in the style run, not just the kerning pairs already defined by the font). Values can range from -1.0 to 1.0. A value of 0 indicates that you want to use font-defined tracking. Negative values loosen font-defined tracking, while positive values tighten font-defined tracking. If you do not set the attribute value corresponding to this tag, the default value is `kATSUNoTracking`, and ATSUI uses the font-defined tracking. Available with ATSUI 1.0.

`kATSUHangingInhibitFactorTag`

Identifies the hanging glyph weighting attribute. You use this tag to set or obtain a value of type `Fract` that represents the amount to bias font-defined hanging glyph adjustments (that is, adjustments to glyphs that typically extend beyond the text margins and are not counted when line length is measured). Values can range from 0 to 1.0. A value of 0 indicates that you want to use font-defined hanging glyph adjustments; 1 indicates that you want to override font-defined hanging glyph adjustments. If you do not set the attribute value corresponding to this tag, the bias is 0, and ATSUI uses the font-defined hanging glyph adjustments. Available with ATSUI 1.0.

`kATSUKerningInhibitFactorTag`

Identifies the kerning weighting attribute. You use this tag to set or obtain a value of type `Fract` that represents the amount to bias font-defined kerning (that is, an adjustment to the normal spacing that occurs between two or more specifically-named glyphs, also called a kerning pair). Values can range from 0 to 1.0. A value of 0 indicates that you want to use font-defined kerning; 1 indicates that you want to override font-defined kerning. If you do not set the attribute value corresponding to this tag, the bias is 0, and ATSUI uses the font-defined kerning. Available with ATSUI 1.0.

`kATSUDecompositionInhibitFactorTag`

Identifies the ligature decomposition weighting attribute. You use this tag to set or obtain a value of type `Fixed` that represents the amount to bias the font-defined ligature decomposition threshold (that is, the threshold beyond which a glyph is broken into its component characters

during justification). Values can range from -1.0 to 1.0. 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 bias is 0, and ATSUI uses the font-defined threshold. Available with ATSUI 1.0.

`kATSUBaselineClassTag`

Identifies the baseline type attribute. You use this tag to set or obtain a value of type `BslnBaselineClass` that represents the type of baseline you want `ATSUCalculateBaselineDeltas` (page 43) to use in calculating optimal baseline positions. Values can range from 0 to 31. See "Baseline Type Constants" (page 260) for a description of possible values. Typically, values of 0 through 4 are used; the remaining represent application-defined baselines. To specify the standard baseline value from the current font, pass the constant `kBSLNNoBaselineOverride`. If you do not set the attribute value corresponding to this tag, default value is `kBSLNRomanBaseline`, and ATSUI uses the Roman baseline. Available with ATSUI 1.0.

`kATSUPriorityJustOverrideTag`

Identifies the justification behavior and priority override weighting attribute. You use this tag to set or obtain an array of type `ATSJustPriorityWidthDeltaOverrides` (page 182) that represents the amount to bias font-defined justification behavior and priority. There are four width delta override structures in the array, one for each justification priority. If you do not set the attribute value corresponding to this tag, the default value is an empty array, and ATSUI uses the font-defined justification behavior and priority. Available with ATSUI 1.0.

`kATSUNoLigatureSplitTag`

Identifies the indivisible ligature attribute. You use this tag to set or obtain a value 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`, and ATSUI treats the ligature as divisible (unless the characters are a surrogate pair or a pre-coded Unicode ligature). Available with ATSUI 1.0.

`kATSUNoCaretAngleTag`

Identifies the straight caret attribute. You use this tag to set or obtain a value 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`, and the caret and highlight angles will reflect the inherent angle of the text. Available with ATSUI 1.0.

`kATSUSuppressCrossKerningTag`

Identifies the cross-kerning inhibition attribute. You use this tag to set or obtain a value of type `Boolean` that indicates whether to inhibit font-defined cross-stream kerning. Setting this value has no impact on manual cross-stream kerning. A value of `true` inhibits font-defined cross-stream kerning. If you do not set the attribute value corresponding to this tag, the default value is `false`, and ATSUI uses font-defined cross-stream kerning. Available with ATSUI 1.0.

`kATSUNoOpticalAlignmentTag`

Identifies the optical alignment inhibition attribute. You use this tag to set or obtain a value of type `Boolean` that indicates whether to inhibit font-specified optical alignment (the automatic adjustment of glyph position at the ends of lines to give a more even visual appearance to margins). A value of `true` inhibits font-defined optical alignment. If you do not set the attribute value corresponding to this tag, the default value is `false`, and ATSUI uses font-defined optical alignment. Available with ATSUI 1.0.

`kATSUForceHangingTag`

Identifies the force hanging glyphs attribute. You use this tag to set or obtain a value of type `Boolean` that indicates

whether hanging glyphs should extend into the margins. A value of `false` indicates that hanging glyphs will not extend into the text margins, even if they 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`, and hanging glyphs will not extend into the text margins. Available with ATSUI 1.0.

`kATSUNoSpecialJustificationTag`

Identifies the inhibit postcompensation actions attribute. You use this tag to set or obtain a value of type `Boolean` that indicates whether postcompensation actions should occur after glyph positions have been calculated. A value of `true` indicates that 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`, and postcompensation actions will occur if they are needed. Available with ATSUI 1.0.

`kATSUStyleTextLocatorTag`

Identifies the text locator attribute. You use this tag to set or obtain a value of type `TextBreakLocatorRef` (page 247) that contains data needed by ATSUI to form various kind of text breaks. If you do not set the attribute value corresponding to this tag, the default value is `NULL`, and ATSUI uses the region-derived locator or the default Text Utilities locator. Available with ATSUI 1.2.

`kATSUMaxStyleTag` Identifies the maximum value for Apple-defined style run attribute tags. Available with ATSUI 1.1.

`kATSUMaxATSUITagValue`

Identifies the maximum value for all Apple-defined tags. Values between `kATSUMaxStyleTag` and `kATSUMaxATSUITagValue` are reserved. You must select a value greater than `kATSUMaxATSUITagValue` if you create your own application-defined style run attribute tag. Available with ATSUI 1.1.

VERSION NOTES

This enumeration is available beginning with ATSUI 1.0. Additional constants added with ATSUI 1.1 and 1.2.

Text Layout Attribute Tag Constants

You can pass a text layout attribute tag constant of type `ATSUAttributeTag` to the functions `ATSUSetLayoutControls` (page 96), `ATSUGetLayoutControl` (page 98), `ATSUGetAllLayoutControls` (page 99), and `ATSUClearLayoutControls` (page 101) to identify the text layout attribute value you wish to set, obtain, or remove from a text layout object. If you do not set a text layout attribute, it will be set to its default value. Table C-2 (page 245) lists the Apple-defined text layout attribute tags and their corresponding data type, size, and default values.

You can also pass these tag constants to the ATSUI functions `ATSUSetLineControls` (page 104), `ATSUGetLineControl` (page 106), `ATSUGetAllLineControls` (page 108), and `ATSUClearLineControls` (page 109) to identify the text layout attribute value you wish to set, obtain, or remove from a single line in a text layout object. Note that when you set a text layout attribute value for a line, this value will override the value of the text layout attribute set for the text layout object containing the line. This is true even if the attributes for the line are set before those of the entire text layout object containing the line.

IMPORTANT

The `ATSUAttributeTag` type also defines style run attribute tag constants; see "Style Run Attribute Tag Constants" (page 217) for a description of these tags. Note that if you pass style run attribute tags to functions that get, set, or remove text layout attribute values, the function will return the result code `kATSUInvalidAttributeTagErr`. ▲

The following constant descriptions assume horizontal text. If you are setting or getting the text layout attribute of vertical text, you should interpret these values accordingly.

```
enum {
    kATSULineWidthTag                   = 1L,
    kATSULineRotationTag                = 2L,
    kATSULineDirectionTag               = 3L,
    kATSULineJustificationFactorTag     = 4L,
    kATSULineFlushFactorTag             = 5L,
    kATSULineBaselineValuesTag          = 6L,
    kATSULineLayoutOptionsTag           = 7L,
    kATSULineAscentTag                  = 8L,
    kATSULineDescentTag                 = 9L,
    kATSULineLanguageTag                = 10L,
```

```
    kATSULineTextLocatorTag             = 11L,
    kATSUMaxLineTag                     = 12L
};
typedef UInt32              ATSUAttributeTag;
```

Constant descriptions

`kATSULineWidthTag` Identifies the line width attribute. You use this tag to set or obtain a value of type `ATSUTextMeasurement` (page 192) that represents the line width to impose on a line of a text layout object. 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, and ATSUI does not impose a line width. Available with ATSUI 1.0.

`kATSULineRotationTag`
Identifies the line rotation attribute. You use this tag to set or obtain a value of type `Fixed` that represents the angle of line rotation (in units of degrees) to impose on a line of a text layout object. Values can range from -1.0 to 1.0. A value of 0 indicates no line rotation. Negative values rotate the line clockwise, while positive values rotate the line counterclockwise. If you do not set the attribute value corresponding to this tag, the default value is 0, and ATSUI does not impose line rotation. Available with ATSUI 1.0.

`kATSULineDirectionTag`
Identifies the line direction attribute. You use this tag to set or obtain a value of type `Boolean` that represents the line direction to impose on every line of a text layout object. You cannot set line direction for a single line. See "Glyph Direction Constants" (page 207) for a description of possible values. If you do not set the attribute value corresponding to this tag, the default value is `false`, and text direction is derived from the system script, which you can determine by calling the `GetSysDirection` function. Available with ATSUI 1.0.

`kATSULineJustificationFactorTag`
Identifies the line justification attribute. You use this tag to set or obtain a value of type `Fract` that represents the justification to impose on a line of a text layout object. Values can range from 0 to 1. See "Line Justification

Constants" (page 211) for a description of possible values. A value of 0 indicates no justification; a value of 1 indicates full justification. If you do not set the attribute value corresponding to this tag, the default value is `kATSUNoJustification`, and ATSUI does not impose line rotation. Available with ATSUI 1.0.

`kATSULineFlushFactorTag`

Identifies the line alignment attribute. You use this tag to set or obtain a value of type `Fract` that represents the alignment to impose on a line of a text layout object. If you set the line alignment attribute, you must also set line justification attribute for the corresponding line(s). Values can range from 0 to 1. See "Line Alignment Constants" (page 210) for a description of possible values. A value of 0 indicates no alignment; a value of 1 indicates full alignment. If you specify the constants `kATSUEndAlignment` or `kATSUCenterAlignment`, you must also set the line width of the corresponding line(s). If you do not set the attribute value corresponding to this tag, the default value is `kATSUStartAlignment`, and ATSUI aligns text to the right of the left margin. Available with ATSUI 1.0.

`kATSULineBaselineValuesTag`

Identifies the baseline positions attribute. You use this tag to set or obtain an array of type `BslnBaselineRecord` (page 257) that represents the optimal baseline positions to use in controlling glyph placement in a line of a text layout object. To determine this value, call the function `ATSUCalculateBaselineDeltas` (page 43) and pass the style object corresponding to the dominant style run in the line. If you do not set the attribute value corresponding to this tag, the default value is an empty array, and ATSUI uses the font-defined cross-stream shift for each glyph in the line. Available with ATSUI 1.0.

`kATSULineLayoutOptionsTag`

Identifies the line layout options attribute. You use this tag to set or obtain a 32-bit mask value of type `ATSLineLayoutOptions` that controls line layout options for a single line or each line of a text layout object. See "Line Layout Option Mask Constants" (page 212) for a description of possible values. If you do not set the

attribute value corresponding to this tag, the default value is `kATSUNoLayoutOptions`, and ATSUI does not set any line layout options. Available with ATSUI 1.0.

`kATSULineAscentTag` Identifies the line ascent attribute. You use this tag to set or obtain a value of type `ATSUTextMeasurement` (page 192) that represents line ascent for a line of a text layout object. This attribute is only taken into account by the functions `ATSUHighlightText` (page 165) and `ATSUUnhighlightText` (page 168) to calculate the ascent of the highlight region. Values range from -1.0 to 1.0, though ATSUI ignores negative values. Nonnegative value reflect an incremental distance above the line's baseline. If you do not set the attribute value corresponding to this tag, the default value is `kATSUseLineHeight`, and ATSUI uses the calculated line ascent from the maximum ascent along the line of all the style runs. Available with ATSUI 1.1.

`kATSULineDescentTag`

Identifies the line descent attribute. You use this tag to set or obtain a value of type `ATSUTextMeasurement` (page 192) that represents the line descent for a single line or each line of a text layout object. This attribute is only taken into account by the functions `ATSUHighlightText` (page 165) and `ATSUUnhighlightText` (page 168) to calculate the descent of the highlight region. Values range from -1.0 to 1.0, though ATSUI ignores negative values. Nonnegative value reflect an incremental distance below the line's baseline. If you do not set the attribute value corresponding to this tag, the default value is `kATSUseLineHeight`, and ATSUI uses the calculated line descent from the maximum descent along the line of all the style runs. Available with ATSUI 1.1.

`kATSULineLanguageTag`

Identifies the language attribute for a line. You use this tag to set or obtain a value of type `RegionCode` that represents the regional language and other region-dependent characteristics for glyphs in a line of a text layout object. See "Region Code Constants" (page 249) for a description of possible values. If you do not set the attribute value corresponding to this tag, the default value is `kTextRegionDontCare`, and ATSUI uses the region code of the system script. To determine this value, evaluate the

result of `GetScriptVariable (smSystemScript,smScriptLang)`. Available with ATSUI 1.2.

`kATSULineTextLocatorTag`
Identifies the text locator attribute. You use this tag to set or obtain a value of type `TextBreakLocatorRef` (page 247) that contains data needed by ATSUI to form various kind of text breaks. If you do not set the attribute value corresponding to this tag, the default value is `NULL`, and ATSUI uses the region-derived locator or the default Text Utilities locator. Available with ATSUI 1.2.

`kATSUMaxLineTag` Identifies the maximum value for Apple-defined text layout attribute tags. Available with ATSUI 1.2.

VERSION NOTES

This enumeration is available beginning with ATSUI 1.0. Additional constants added with ATSUI 1.1 and 1.2.

Text Length Constant

ATSUI functions that draw, highlight, measure, or otherwise operate on text do so to a range of text, not the entire text buffer (unless you specify the entire buffer - see next paragraph). You specify the beginning of this range with an edge offset of type `UniCharArrayOffset` (page 193), and demarcate the end of the range by indicating a length of type `UniCharCount` (page 194).

If you want the range to span the end of the text buffer, you should pass the constant `kATSUToTextEnd`. If you want the range to span the entire text buffer (from the beginning), pass `kATSUToTextEnd` in conjunction with the constant `kATSUFromTextBeginning`, described in "Text Offset Constant" (page 231).

```
enum {
    kATSUToTextEnd      = (long)0xFFFFFFFF
};
```

Constant descriptions

`kATSUToTextEnd` Indicates that the range of text to be operated on should span to the end of the text layout object's text buffer.

VERSION NOTES

Available beginning with ATSUI 1.0.

Text Offset Constant

ATSUI functions that draw, highlight, measure, or otherwise operate on text do so to a range of text, not the entire text buffer (unless you specify the entire buffer - see next paragraph). You specify the beginning of this range with an edge offset of type `UniCharArrayOffset` (page 193), and demarcate the end of the range by indicating a length of type `UniCharCount` (page 194).

If you want the range to start at the beginning of the text buffer, you should pass the constant `kATSUFromTextBeginning`. If you want the range to span the entire text buffer, pass `kATSUFromTextBeginning` in conjunction with the constant `kATSUToTextEnd`, described in "Text Length Constant" (page 230). This constant is available with ATSUI 1.0 and later.

```
enum {
    kATSUFromTextBeginning      = (long)0xFFFFFFFF,
};
```

Constant descriptions

`kATSUFromTextBeginning`
Indicates that the range of text to be operated on should start at the beginning of the text layout object's text buffer.

VERSION NOTES

Available beginning with ATSUI 1.0.

Result Codes

All ATSUI functions return result codes of type `OSStatus`. This includes general result codes such as `noErr`, indicating that the function completed successfully, and `paramErr`, indicating that you either passed the function an invalid input parameter value or passed `NULL` for all output parameters. ATSUI functions that

allocate memory may return `memFullErr` if there is not enough memory in the designated heap.

The result codes specific to ATSUI are listed in Table 2-1 (page 232). 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 2-1 ATSUI-specific result codes

Result code constant	Value	Description
`kATSUInvalidTextLayoutErr`	-8790	Text layout object not previously initialized or in an otherwise invalid state. Available beginning with ATSUI 1.0.
`kATSUInvalidStyleErr`	-8791	Style object not previously initialized or in an otherwise invalid state. Available beginning with ATSUI 1.0.
`kATSUInvalidTextRangeErr`	-8792	Text range extends beyond the limits of the text layout object's text range. Available beginning with ATSUI 1.0.
`kATSUFontsMatched`	-8793	Character could not be rendered with its assigned font. Available beginning with ATSUI 1.0.
`kATSUFontsNotMatched`	-8794	Character could not be rendered with its assigned font or any currently active font. Available beginning with ATSUI 1.0.
`kATSUNoCorrespondingFontErr`	-8795	Font ID corresponds to an existing font that isn't available to ATSUI. Available beginning with ATSUI 1.0.
`kATSUInvalidFontErr`	-8796	Font ID does not correspond to any installed font. Available beginning with ATSUI 1.0.
`kATSUInvalidAttributeValueErr`	-8797	Invalid or undefined attribute value. Available beginning with ATSIU 1.0.
`kATSUInvalidAttributeSizeErr`	-8798	Allocated attribute value size is less than required. Available beginning with ATSUI 1.0.

Table 2-1 ATSUI-specific result codes

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). Available beginning with ATSUI 1.0.
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). Available beginning with ATSUI 1.0.
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. Available beginning with ATSUI 1.0.
kATSUNoStyleRunsAssignedErr	-8802	No style runs assigned to text layout object. Available beginning with ATSUI 1.1.
kATSUQuickDrawTextErr	-8803	QuickDraw function DrawText encountered an error rendering or measuring a line of text. Available beginning with ATSUI 1.1.
kATSULowLevelErr	-8804	Error encountered in Apple Type Solution (ATS) while performing an operation requested by ATSUI. Available beginning with ATSUI 1.1.
kATSUNoFontCmapAvailableErr	-8805	'CMAP' table cannot be accessed or synthesized for a font set by the function ATSUSetAttributes (page 37). Available beginning with ATSUI 1.1.
kATSUNoFontScalerAvailableErr	-8806	No font scaler available for a font set by the function ATSUSetAttributes (page 37). Available beginning with ATSUI 1.1.

Table 2-1 ATSUI-specific result codes

kATSUCoordinateOverflowErr	-8807	Passed in coordinate values caused coordinate overflow (greater than 32K). Available beginning with ATSUI 1.1.
kATSULineBreakInWord	-8808	Not an error code. Returned by ATSUBreakLine (page 157) to indicate that ATSUBreakLine performed a line break within a word. Available beginning with ATSUI 1.2.
kATSULastErr	-8809	No ATSUI-related result codes may exceed this value. Result code values between kATSUInvalidTextLayoutErr and kATSULastErr are reserved. Available beginning with ATSUI 1.0.

VERSION NOTES

This enumeration is available beginning with ATSUI 1.0. Additional constants added with ATSUI 1.1 and 1.2.

Document Revision History

This document has had the following releases:

Table A-1 *Apple Type Services for Unicode Imaging Reference* revision history

Version	Notes
Jan. 11, 2000	This release covers ATSUI features through version 1.2. For a list of those functions introduced with ATSUI 1.2 or whose implementation have changed since 1.0 or 1.1, see "History of API Additions and Changes in ATSUI" (page 237).
Aug. 20, 1999	This release is an update to the previous document and includes corrected descriptions.
May 7, 1999	This release covers ATSUI features through version 1.1. For a list of those functions introduced with ATSUI 1.1 or whose implementation have changed since 1.0, see "History of API Additions and Changes in ATSUI" (page 237).
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 Unicode Text Using Apple Type Services for Unicode Imaging (ATSUI).*

APPENDIX

Document Revision History

History of API Additions and Changes in ATSUI

Table B-1 alphabetically lists the functions whose implementation have changed with ATSUI 1.2.

Table B-1 Functions whose implementation has changed in ATSUI 1.2

Function name	Description of change in ATSUI 1.2
`ATSUGetGlyphBounds` (page 146)	Now the maximum number of enclosing trapezoids returned is 127 rather than 31, as was true for the implementation of this function with ATSUI 1.0 and 1.1.
`ATSUGetTextHighlight` (page 170)	The maximum number of components that can be passed back is 127 rather than 31.
`ATSUSetFontFeatures` (page 44)	If you try to set mutually exclusive font features, whether in one or multiple calls, ATSUI 1.2 removes the existing feature and replaces it with your newly set feature. Earlier versions of ATSUI will not remove a feature when you set its contradictory feature.

The following constants were introduced with ATSUI 1.2: `gestaltATSUUpdate2`, `gestaltATSUTextLocatorUsageFeature`, `kATSLineTabAdjustEnabled`, `kATSUStyleTextLocatorTag`, `kATSULineLanguageTag`, `kATSULineTextLocatorTag`, and `kATSUMaxLineTag`.

Table B-2 alphabetically lists the functions that were introduced with ATSUI 1.1.

Table B-2 Functions new to ATSUI 1.1

Function name
`ATSUClearLayoutCache` (page 93)
`ATSUClearLineControls` (page 109)
`ATSUCopyLineControls` (page 103)
`ATSUCountFontTracking` (page 68)
`ATSUCreateAndCopyTextLayout` (page 91)
`ATSUCreateMemorySetting` (page 174)
`ATSUDisposeMemorySetting` (page 177)
`ATSUGetAllLineControls` (page 108)
`ATSUGetCurrentMemorySetting` (page 176)
`ATSUGetFontFallbacks` (page 124)
`ATSUGetIndFontTracking` (page 69)
`ATSUGetLineControl` (page 106)
`ATSUSetCurrentMemorySetting` (page 176)
`ATSUSetFontFallbacks` (page 123)
`ATSUSetLineControls` (page 104)
`ATSUCustomAllocFunc` (page 178)
`ATSUCustomGrowFunc` (page 179)
`ATSUCustomFreeFunc` (page 180)

Table B-3 alphabetically lists the ATSUI 1.0 functions whose implementation changed with ATSUI 1.1.

Table B-3 Functions whose implementation has changed with ATSUI 1.1

Function name	Description of change in 1.0
`ATSUBreakLine` (page 157)	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 35)	No longer used. Instead, use the `'ustl'` resource to format styled text in the clipboard.
`ATSUHighlightText` (page 165)	Now can extend highlighting across tab stops using the mask constants `kATSLineFillOutToWidth` and `kATSLineImposeNoAngleForEnds`.
`ATSUPasteFromHandle` (page 36)	No longer used. Instead, use the `'ustl'` resource to format styled text in the clipboard.
`ATSUSetAttributes` (page 37)	New style run attribute tag constants now available.
`ATSUSetLayoutControls` (page 96)	New text layout and line attribute tag constants now available.
`ATSUUnhighlightText` (page 168)	Now can erase highlighting across tab stops using the mask constants `kATSLineFillOutToWidth` and `kATSLineImposeNoAngleForEnds`.

APPENDIX B

History of API Additions and Changes in ATSUI

Summary of Style Run and Text Layout Attribute Tag Information

This appendix provides tabular summaries of the Apple-defined style run and text layout attribute tags and the data type, size, and default values of the attributes they identify. Table C-1 summarizes this information for style run attribute tags.

Table C-1 Style run attribute tags and the data type, size, and default values of the style run attributes they identify

Style run attribute tag	Corresponding data type and size (in bytes)		Corresponding default value
`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
`kATSUFontTag`	`ATSUFontID` (page 188)	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.

Table C-1 Style run attribute tags and the data type, size, and default values of the style run attributes they identify

Style run attribute tag	Corresponding data type and size (in bytes)		Corresponding default value
`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)`
`kATSUVerticalCharacterTag`	`ATSUVerticalCharacterType`	2	`kATSUStronglyHorizontal`; horizontally-oriented glyphs
`kATSUImposeWidthTag`	`ATSUTextMeasurement` (page 192)	4	`kATSUNoImposedWidth`; use font-defined character width default value
`kATSUBeforeWithStreamShiftTag`	`Fixed`	4	0; use the font-defined with-stream shift default value before glyphs

Table C-1 Style run attribute tags and the data type, size, and default values of the style run attributes they identify

Style run attribute tag	Corresponding data type and size (in bytes)		Corresponding default value
`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 188)	4	`kBSLNRomanBaseline`; Roman baseline
`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

Table C-1 Style run attribute tags and the data type, size, and default values of the style run attributes they identify

Style run attribute tag	Corresponding data type and size (in bytes)		Corresponding default value
`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
`kATSUNoSpecialJustificationTag`	`Boolean`	1	`false`; post compensation 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.

Table C-2 summarizes the Apple-defined text layout attribute tags and their corresponding data type, size, and default values.

Table C-2 Text layout attribute tags and the data type, size, and default value of the attributes they identify

Text layout attribute tag	Corresponding data type and size (in bytes)		Corresponding default value
`kATSULineWidthTag`	`ATSUTextMeasurement` (page 192)	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 257)	4	all 0's; no baseline deltas are applied to the cross-stream shifting of glyphs

Table C-2 Text layout attribute tags and the data type, size, and default value of the attributes they identify (continued)

Text layout attribute tag	Corresponding data type and size (in bytes)		Corresponding default value
`kATSULineLayoutOptionsTag`	`UInt32`	4	`kATSUNoLayoutOptions`; no special line layout options are set
`kATSULineAscentTag`	`ATSUTextMeasurement` (page 192)	4	`kATSUseLineHeight`; use the maximum line ascent of all the style runs in a line **<<is this right?>>**
`kATSULineDescentTag`	`ATSUTextMeasurement` (page 192)	4	`kATSUseLineHeight`; use the maximum line descent of all the style runs in a line
`kATSULineLanguageTag`	`RegionCode`	2	0; no language variation specified
`kATSULineTextLocatorTag`	`TextBreakLocatorRef` (page 247)	4	`NULL`; language specified locator or the default Text Utilities locator

New Constants and Data Types Used by ATSUI

This appendix describes new constants and data types used by ATSUI from Unicode Utilities, the Script Manager, and Apple Advanced Typography.

About Unicode Utilities

Version 1.0 of Unicode Utilities provided support for Unicode input methods and was documented in the seed note *Inside Macintosh: Supporting Unicode Input*. This section describes the `TextBreakLocatorRef` data type, which was added since online publication of the seed note.

Unicode Utilities Reference

Unicode Utilities Data Type

TextBreakLocatorRef

The `TextBreakLocatorRef` type is reference to a private structure that contains data needed by ATSUI and the Unicode Utilities function `UCFindTextBreak` to form various kind of text breaks. ATSUI developers must create a text break locator reference to set the style run and text layout attributes identified by the

tag constants `kATSUStyleTextLocatorTag` and `kATSULineTextLocatorTag`. For a description of these tag constants, see "Style Run Attribute Tag Constants" (page 217) and "Text Layout Attribute Tag Constants" (page 226), respectively. You create a text break locator reference by calling the Unicode Utilities function `UCCreateTextBreakLocator`. The Unicode Utilities functions `UCFindTextBreak` and `UCCreateTextBreakLocator` are documented in *Unicode Utilities Preliminary Documentation* at the Mac OS 8 and 9 Developer Documentation web site:

<http://developer.apple.com/techpubs/macos8/pdf/UnicodeUtilities.pdf>.

```
typedef struct OpaqueTextBreakLocatorRef*  TextBreakLocatorRef
```

VERSION NOTES

Available in the header file `UnicodeUtilities.h` beginning with Mac OS 9.

About Script Manager

The Script Manager enables developers to modify the features of an individual script system, as well as to parse source code and convert text among subscripts. The Script Manager was previously documented in *Inside Macintosh: Text*. This section describes the region code constants that have been added since that publication was released.

Script Manager Reference

Script Manager Constants

Region Code Constants

These constants specify a combination of a language code and a particular region. Some of these numeric values are reserved just for extra resource IDs associated with certain regions; these are not actual region codes, and are noted in the comments. Not all of the region codes are currently supported by Apple software.

ATSUI uses region code information to direct line breaking and cursor movement functions like `ATSUBreakLine` (page 157), `ATSUNextCursorPosition` (page 136), `ATSUPreviousCursorPosition` (page 138), `ATSURightwardCursorPosition` (page 140), and `ATSULeftwardCursorPosition` (page 141). You can use region code constants to set the style run and text layout attributes identified by the tag constants `kATSULanguageTag` and `kATSULineLanguageTag`. For more information, see "Style Run Attribute Tag Constants" (page 217) and "Text Layout Attribute Tag Constants" (page 226), respectively.

Note that in the comments for the following enumeration, many of the region codes have an associated P/N, ISO code, and comments. P/N stands for the Apple part number code for software localized for the specified region; the ISO code is made up of the two-letter ISO language and country codes from ISO 639 and ISO 3166. The ISO code is described by the lowercase language code, followed by "_", then the uppercase country code.

```
enum {                          /* P/N  ISO codes   comments*/
    verUS               = 0,    /*      en_US*
    verFrance           = 1,    /* F    fr_FR*/
    verBritain          = 2,    /* B    en_GB*/
    verGermany          = 3,    /* D    de_DE*/
    verItaly            = 4,    /* T    it_IT*/
    verNetherlands      = 5,    /* N    nl_NL*/
    verFlemish          = 6,    /* FN   nl_BE   Flemish (Dutch) for
                                            Belgium*/
    verSweden           = 7,    /* S    sv_SE*/
    verSpain            = 8,    /* E    es_ES   Spanish for Spain*/
    verDenmark          = 9,    /* DK   da_DK*/
    verPortugal         = 10,   /* PO   pt_PT   Portuguese for Portugal*/
    verFrCanada         = 11,   /* C    fr_CA   French for Canada*/
    verNorway           = 12,   /* H    no_NO   Bokmål*
    verIsrael           = 13,   /* HB   iw_IL   Hebrew*/
    verJapan            = 14,   /* J    ja_JP*/
```

```
    verAustralia        = 15,   /* X    en_AU   English for Australia*/
    verArabic           = 16,   /* AB   ar      Arabic for N Africa,
                                                Arabian peninsula,
                                                Levant*/
    verFinland          = 17,   /* K    fi_FI*/
    verFrSwiss          = 18,   /* SF   fr_CH   French Swiss*/
    verGrSwiss          = 19,   /* SD   de_CH   German Swiss*/
    verGreece           = 20,   /* GR   el_GR   Monotonic Greek(modern)*/
    verIceland          = 21,   /* IS   is_IS*/
    verMalta            = 22,   /* MA   mt_MT*/
    verCyprus           = 23,   /* CY    _CY    Greek or Turkish lang?*/
    verTurkey           = 24,   /* TU   tr_TR*/
    verYugoCroatian     = 25    /* YU           Croatian for Yugoslavia;
                                                now use verCroatia (68)*/
    verNetherlandsComma = 26,   /*             ID for KCHR resource -
                                                Dutch*/
    verBelgiumLuxPoint  = 27,   /*             ID for KCHR resource -
                                                Belgium*/
    verCanadaComma      = 28,   /*             ID for KCHR resource -
                                                Canadian ISO*/
    verCanadaPoint      = 29,   /*             ID for KCHR resource -
                                                Canadian, now unused*/
    vervariantPortugal  = 30,   /*             ID for resource; now
                                                unused*/
    vervariantNorway    = 31,   /*             ID for resource; now
                                                unused*/
    vervariantDenmark   = 32    /*             ID for KCHR resource -
                                                Danish Mac Plus*/
    verIndiaHindi       = 33,   /*      hi_IN   Hindi for India*/
    verPakistanUrdu     = 34,   /* UR   ur_PK   Urdu for Pakistan   */
    verTurkishModified  = 35,
    verItalianSwiss     = 36,   /* ST   it_CH   Italian Swiss*/
    verInternational    = 37,   /* Z    en      English for international
                                                use     */
                                /*              38 is unassigned*/
    verRomania          = 39,   /* RO   ro_RO*/
    verGreecePoly       = 40,   /*              Polytonic Greek
                                                (classical)  */
    verLithuania        = 41,   /* LT   lt_LT*/
    verPoland           = 42,   /* PL   pl_PL*/
    verHungary          = 43,   /* MG   hu_HU*/
```

```
    verEstonia          = 44,   /* EE   et_EE*/
    verLatvia           = 45,   /* LV   lv_LV*/
    verSami             = 46,   /*       se  */
    verFaroeIsl         = 47,   /* FA   fo_FO   */
    verIran             = 48,   /* PS   fa_IR   Persian/Farsi*/
    verRussia           = 49,   /* RSr  u_RU    Russian*/
    verIreland          = 50,   /* GA   ga_IE   Irish Gaelic for Ireland
                                                (without dot above)*/
    verKorea            = 51,   /* KH   ko_KR*/
    verChina            = 52,   /* CH   zh_CN   Simplified Chinese*/
    verTaiwan           = 53,   /* TA   zh_TW   Traditional Chinese*/
    verThailand         = 54,   /* TH   th_TH*/
    verScriptGeneric    = 55,   /* SS           Generic script system (no
                                                language or script)*/
    verCzech            = 56,   /* CZ   cs_CZ*/
    verSlovak           = 57,   /* SL   sk_SK*/
    verFarEastGeneric   = 58,   /* FE           Generic Far East system
                                                (no language or script)*/
    verMagyar           = 59,   /*               Unused; see verHungary*/
    verBengali          = 60,   /*       bn      Bangladesh or India*/
    verByeloRussian     = 61    /* BY   be_BY*/
    verUkraine          = 62,   /* UA   uk_UA*/
                                /*               63 is unassigned*/
    verGreeceAlt        = 64,   /*               unused          */
    verSerbian          = 65,   /* SR   sr_YU, sh_YU            */
    verSlovenian        = 66,   /* SV   sl_SI   */
    verMacedonian       = 67,   /* MD   mk_MK   */
    verCroatia          = 68,   /* CR   hr_HR, sh_HR*/
                                /*               69 is unassigned*/
                                /*               70 is unassigned*/
    verBrazil           = 71,   /* BR   pt_BR   Portuguese for Brazil*/
    verBulgaria         = 72,   /* BG   bg_BG*/
    verCatalonia        = 73,   /* CA   ca_ES   Catalan for Spain*/
    verMultilingual     = 74,   /* ZM           (no language or script)*/
    verScottishGaelic   = 75,   /* GD   gd*/
    verManxGaelic       = 76,   /* GV   gv      Isle of Man*/
    verBreton           = 77,   /* BZ   br*/
    verNunavut          = 78,   /* IU   iu_CA   Inuktitut for Canada*/
    verWelsh            = 79,   /* CU   cy*/
                                /*  80 is ID for KCHR resource - Canadian
                                    CSA*/
```

```
    verIrishGaelicScript= 81,   /* GS   ga_IE   Irish Gaelic for Ireland
                                               (using dot above)*/
    verEngCanada        = 82,   /* V    en_CA   English for Canada*/
    verBhutan           = 83,   /* BH   dz_BT   Dzongkha for Bhutan*/
    verArmenian         = 84,   /* HY   hy_AM*/
    verGeorgian         = 85,   /* KR   ka_GE*/
    verSpLatinAmerica   = 86,   /* LA   es      Spanish for Latin
                                               America*/
                                /*              87 is ID for KCHR
                                               resource - Spanish ISO*/
    verTonga            = 88,   /* TS   to_TO*/
                                /*              89 is ID for KCHR
                                               resource - Polish
                                               Modified*/
                                /*              90 is ID for KCHR
                                               resource - Catalan ISO*/
    verFrenchUniversal  = 91,   /*      fr      French generic*/
    verAustria          = 92,   /* AU   de_AT   German for Austria*/
                                /* Y            93 is unused alternate
                                               for verSpLatinAmerica*/
    verGujarati         = 94,   /*      gu_IN*/
    verPunjabi          = 95,   /*      pa      Pakistan or India*/
    verIndiaUrdu        = 96,   /*      ur_IN   Urdu for India*/
    verVietnam          = 97    /*      vi_VN*/}
    verFrBelgium        = 98,   /* BF   fr_BE   French for Belgium   */
    verUzbek            = 99,   /* BD   uz_UZ */
    verSingapore        = 100,  /* SG   */
    verNynorsk          = 101,  /* NY  _NO      Norwegian Nynorsk */
    verAfrikaans        = 102,  /* AK   af_ZA   */
    verEsperanto        = 103,  /*      eo  */
    verMarathi          = 104,  /*      mr_IN    */
    verTibetan          = 105,  /*      bo      */
    verNepal            = 106,  /*      ne_NP   */
    verGreenland        = 107   /*      kl      */
};
```

Constant descriptions

`verUS`	Identifies English spoken in the United States.
`verFrance`	Identifies French spoken in France.
`verBritain`	Identifies English spoken in Britain.

`verGermany`	Identifies German spoken in Germany.
`verItaly`	Identifies Italian spoken in Italy.
`verNetherlands`	Identifies Dutch spoken in the Netherlands.
`verFlemish`	Identifies Flemish (Dutch) spoken in Belgium.
`verSweden`	Identifies Swedish spoken in Sweden.
`verSpain`	Identifies Spanish spoken in Spain.
`verDenmark`	Identifies Danish spoken in Denmark.
`verPortugal`	Identifies Portugal Portuguese.
`verFrCanada`	Identifies French spoken in Canada.
`verNorway`	Identifies Bokmal.
`verIsrael`	Identifies Hebrew.
`verJapan`	Identifies Japanese.
`verAustralia`	Identifies English spoken in Australia.
`verArabic`	Identifies Arabic spoken in North Africa, Arabian peninsula, and Levant.
`verFinland`	Identifies Finnish spoken in Finland.
`verFrSwiss`	Identifies French Swiss.
`verGrSwiss`	Identifies German Swiss.
`verGreece`	Identifies monotonic (that is, modern) Greek.
`verIceland`	Identifies Icelandic.
`verMalta`	Identifies Maltese.
`verCyprus`	Identifies Greek or Turkish spoken in Cyprus.
`verTurkey`	Identifies Turkish spoken in Turkey.
`verYugoCroatian`	Identifies Croatian spoken in Yugoslavia. No longer used. You should instead use `VerCroatia`.
`verNetherlandsComma`	Identifies the Dutch `'KCHR'` resource.
`verBelgiumLuxPoint`	Identifies the Belgium `'KCHR'` resource.
`verCanadaComma`	Identifies the Canadian ISO `'KCHR'` resource.

`verCanadaPoint`	Identifies the Canadian `'KCHR'` resource. No longer used.
`vervariantPortugal`	Identifies the Portuguese `'KCHR'` resource. No longer used.
`vervariantNorway`	Identifies the Norwegian `'KCHR'` resource. No longer used.
`vervariantDenmark`	Identifies the Danish Mac Plus `'KCHR'` resource.
`verIndiaHindi`	Identifies Hindi spoken in India.
`verPakistanUrdu`	Identifies Urdu spoken in Pakistan.
`verTurkishModified`	Identifies Turkish.
`verItalianSwiss`	Identifies Italian Swiss.
`verInternational`	Identifies English for international use.
`verRomania`	Identifies Romanian.
`verGreecePoly`	Identifies polytonic (that is, classical) Greek.
`verLithuania`	Identifies Lithuanian.
`verPoland`	Identifies Polish.
`verHungary`	Identifies Hungarian.
`verEstonia`	Identifies Estonian.
`verLatvia`	Identifies Latvian.
`verSami`	Identifies Saamisk.
`verFaroeIsl`	Identifies the north Germanic language spoken on Faeroe Island.
`verIran`	Identifies Persian (that is, Farsi).
`verRussia`	Identifies Russian.
`verIreland`	Identifies Irish Gaelic spoken in Ireland (that is, without a dot above).
`verKorea`	Identifies Korean.
`verChina`	Identifies simplified Chinese.
`verTaiwan`	Identifies traditional Chinese.
`verThailand`	Identifies Thai.

`verScriptGeneric`	Identifies the generic script system (that is, no language or script).
`verCzech`	Identifies Czech.
`verSlovak`	Identifies Slovak.
`verFarEastGeneric`	Identifies generic script system for far east system.
`verMagyar`	Unused. See `verHungary`.
`verBengali`	Identifies Bengali spoken in Bangladesh or India.
`verByeloRussian`	Identifies Russian.
`verUkraine`	Identifies Ukrainian.
`verGreeceAlt`	Identifies Greek alternative. Not used.
`verSerbian`	Identifies Serbian.
`verSlovenian`	Identifies Slovenian.
`verMacedonian`	Identifies Macedonian.
`verCroatia`	Identifies Croatian.
`verBrazil`	Identifies Brazilian Portuguese.
`verBulgaria`	Identifies Bulgarian.
`verCatalonia`	Identifies Catalan spoken in Spain.
`verMultilingual`	Identifies no language or script.
`verScottishGaelic`	Identifies Scottish Gaelic.
`verManxGaelic`	Identifies Gaelic spoken in the Isle of Man.
`verBreton`	Identifies Breton.
`verNunavut`	Identifies Nunavut spoken by the Inuktitut in Canada.
`verWelsh`	Identifies Welsh.
`verIrishGaelicScript`	Identifies Irish Gaelic for Ireland (with the dot above).
`verEngCanada`	Identifies English spoken in Canada.
`verBhutan`	Identifies Dzongkha spoken in Bhutan.
`verArmenian`	Identifies Armenian.

verGeorgian	Identifies Georgia spoken in Georgia, Russia.
verSpLatinAmerica	Identifies Spanish spoken in Latin America.
verFrenchUniversal	Identifies generic French.
verAustria	Identifies German spoken in Austria.
verGujarati	Identifies Gujarti.
verPunjabi	Identifies Punjabi spoken in Pakistan and India.
verIndiaUrdu	Identifies Urdu spoken in India.
verVietnam	Identifies Vietnamese.
verFrBelgium	Identifies French spoken in Belgium.
verUzbek	Identifies Uzbek.
verSingapore	Identifies Chinese spoken in Singapore.
verNynorsk	Identifies Norwegian Nynorsk.
verAfrikaans	Identifies Afrikaans.
verEsperanto	Identifies Esperanto.
verMarathi	Identifies Marathi.
verTibetan	Identifies Tibetan.
verNepal	Identifies Nepalese.
verGreenland	Identifies Greenlandic.

About Apple Advanced Typography

While ATSUI enables advanced high-quality typography and expanded Unicode-based multilingual line layout, Apple Advanced Typography (AAT) provides advanced layout support. There are two header files associated with Apple Advanced Typography: `SFNTTypes.h` and `SFNTLayoutTypes.h`. Both files define constants and data types necessary to access the font tables of an `'SFNT'` font. These tables pertain to text rendering as well as layout. This information can be useful for font designers, as well as for developers wishing to access font table information. You can pass some of these constants and types to ATSUI functions that search a font name table for information. In addition, there are

several ATSUI functions that pass back a subset of the font feature type and selector constants described in this section.

Apple Advanced Typography Reference

Apple Advanced Typography Data Type

BslnBaselineRecord

The `BslnBaselineRecord` type is an array of 32 `Fixed` values, each of which represents a baseline type in a style run. Examples of baseline types include mathematical, ideographic, and Roman. An array of type `BslnBaselineRecord` is passed back by the function `ATSUCalculateBaselineDeltas` (page 43) to represent the distance from the dominant baseline to each baseline type in a style run. The `ATSUTextMeasurement` type can be also used to set and get optimum baseline positions; see "Text Layout Attribute Tag Constants" (page 226).

```
typedef Fixed        BslnBaselineRecord[32];
```

Apple Advanced Typography Constants

Annotation Feature Selector Constants

The annotation feature type specifies annotations (or adornments) to basic letter shapes. 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-1 lists the selectors for this feature.

Table D-1 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`.
`kInvertedCircleAnnotationSelector`	Same as circle annotation, but with white and black reversed. For instance, see Unicode characters `U+2776` through `U+277F`.

Table D-1 Feature selectors for the `kAnnotationType` feature type (continued)

Constant	Explanation
`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.

Baseline Type Constants

You can pass a constant of type `kATSUBaselineClassTag` to the function `ATSUCalculateBaselineDeltas` (page 43) to specify the type of baseline you want used in calculating optimal baseline positions. To specify the standard baseline value from the current font, pass the constant `kBSLNNoBaselineOverride`.

You can also use a constant of this type to set or obtain the primary baseline in a style run; see the functions `ATSUSetAttributes` (page 37) and `ATSUGetAttribute` (page 38) respectively.

```
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.

CJK Roman Width Feature Selector Constants

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-2 lists the selectors for this feature.

Table D-2 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.

Character Alternates Feature Selector Constants

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-3 lists the only defined selector for this feature.

Table D-3 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.

Character Shape Feature Selector Constants

The Chinese language can be represented with both a traditional and a simplified character set, as shown in Figure D-1. Chinese fonts that support the character shape feature type allow you to select either set.

Figure D-1 Traditional and simplified versions of a Chinese character

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-4 lists the selectors for this feature.

Table D-4 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.

Table D-4 Feature selectors for the `kCharacterShapeType` feature type (continued)

Constant	Explanation
`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.

Cursive Connection Font Feature Selector Constants

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 letter forms that connect in a non contextual manner, or enable completely contextual, cursively connected letter forms (as in Arabic). Table D-5 lists the feature selectors for cursive connection.

Figure D-2 shows an example of non contextual cursive connection in a Roman font.

Table D-5 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.

Figure D-2 Non contextual cursive connection in a Roman font

Design Complexity Feature Selector Constants

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-6.

Table D-6 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.

Diacritical Mark Font Feature Selector Constants

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-7.

Table D-7 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. Figure D-3 shows an example of Hebrew text drawn with and without its diacritical marks.

Figure D-3 Hebrew text with diacritical marks shown (upper) and hidden (lower)

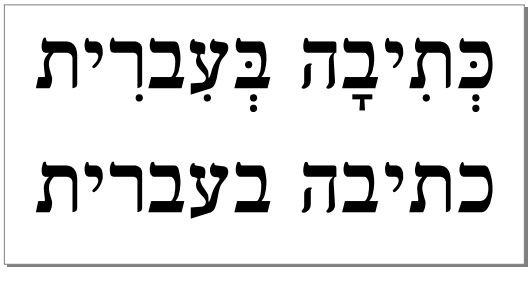
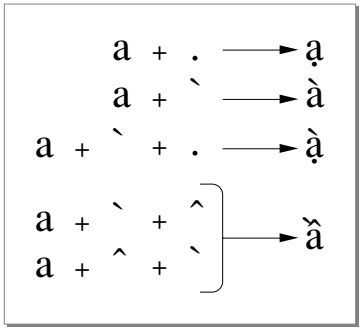


Figure D-4 shows an example of text drawn with and without its accents.

Figure D-4 Accented forms



Font Feature Type Constants

Font features are grouped into categories called feature types, within which feature selectors are used to define particular feature settings or selections. The set of feature types described in this section may not be complete. For a description of the most up-to-date set of registered typographic and layout features available to applications using Apple Advanced Typography (AAT), see the Font Feature Registry at the Apple Font Feature Registry web site:

<http://fonts.apple.com/Registry/index.html/>.

Table D-8 (page 268) contains descriptions of some of the feature types might be available in a font. You should query a font to determine which of these font features are available, then build your own list of font features.

Note that unless the feature is defaulted differently in different fonts, the zero value for the selectors represents the default value.

Table D-8 Examples of feature types

Constant	Explanation
`kAllTypographicFeaturesType`	Specifies whether or not any font features are to be applied at all. Table D-9 (page 271) lists the feature selectors related to this feature type.
`kLigaturesType`	Specifies the use of required ligatures and other categories of optional ligatures. Table D-14 (page 292) lists the feature selectors related to this feature type.
`kCursiveConnectionType`	Specifies whether or not cursive connections are to be used between glyphs. Table D-5 (page 265) 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-13 (page 291) 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-28 (page 313) lists the feature selectors related to this feature type.
`kLinguisticRearrangementType`	Either permits or inhibits linguistic (Indic-style) rearrangement of glyphs. Table D-15 (page 294) lists the feature selectors related to this feature type.
`kNumberSpacingType`	Specifies whether to use fixed-width or proportional-width glyphs for numerals. Table D-18 (page 303) 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-22 (page 307) lists the feature selectors related to this feature type.
`kDiacriticsType`	Controls whether diacritical marks are shown or hidden. Table D-7 (page 266) lists the feature selectors related to this feature type.

Table D-8 Examples of feature types (continued)

Constant	Explanation
`kVerticalPositionType`	Controls the selection of superscript and subscript glyph sets. Table D-27 (page 313) lists the feature selectors related to this feature type.
`kFractionsType`	Controls automatic substitution or formation of fractions. Table D-10 (page 286) 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-20 (page 305) 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-25 (page 310) 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-16 (page 300) 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-19 (page 304) lists the feature selectors related to this feature type.
`kCharacterAlternativesType`	Specifies, by number, any font-specific set of alternate glyph forms. Table D-3 (page 262) lists the feature selector related to this feature type.
`kDesignComplexityType`	Specifies an overall complexity of appearance, as defined by the font. Table D-6 (page 265) 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-21 (page 306) 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-4 (page 263) lists the feature selectors related to this feature type.

Table D-8 Examples of feature types (continued)

Constant	Explanation
`kNumberCaseType`	Specifies whether to use numerals that do, or do not, extend below the baseline. Table D-17 (page 302) 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-23 (page 309) lists the feature selectors related to this feature type.
`kTransliterationType`	Allows text in one format to be displayed using another format. Table D-24 (page 309) lists the feature selectors related to this feature type.
`kAnnotationType`	Specifies annotations (or adornments) to basic letter shapes. For instance, most Japanese fonts include versions of numbers that are circled, parenthesized, have periods after them, and so on. Table D-1 (page 259) lists the feature selectors related to this feature type.
`kKanaSpacingType`	Specifies widths for Japanese Hiragana and Katakana characters. Table D-12 (page 290) 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-11 (page 287) 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, Japanese, and Korean) font. Table D-2 (page 262) lists the feature selectors related to this feature type.
`kUnicodeDecompositionType`	Table D-26 (page 311) lists the feature selectors related to this feature type.
`kLastFeatureType`	Represents the last Apple-reserved font feature type value.

Font Feature Type Selector Constants

This is the feature selector indicating whether font features are on or off. Table D-9 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-9 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.

Font Name Code Constants

ATSUI identifies the type of a particular font name string by a constant of type `FontNameCode`. You can use one of these constants as part of your search criteria in the functions `ATSUFindFontFromName` (page 57) and `ATSUFindFontName` (page 63). The function `ATSUGetIndFontName` (page 61) passes back a constant of this type to represent the type of font name string.

The font name code identifies the type of 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. The actual font name string varies 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 with this font name code and a platform and script of 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 with this font name code and a platform and script of Mac Roman English is "Times".

`kFontStyleName` Identifies the font style. An example of a font name string with this font name code and a platform and script of Mac Roman English is "© Apple Computer, Inc. 1993". 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 with this font name code and a platform and script of 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 with this font name code and a platform and script of Mac Roman English is "Times Black".

`kFontVersionName` Identifies the font manufacturer's version number for the font. An example of a font name string with this font name code and a platform and script of 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 with this font name code and a platform and script of Mac Roman English is "Times-Black".

`kFontTrademarkName`
Identifies the font trademark name. An example of a font name string with this font name code and a platform and script of 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 with this font name code and a platform and script of 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`

Identifies the maximum value for Apple-defined font name codes. You can pass values between `kFontLicenseInfoURLName` and `kFontLastReservedName` to find the name of a font variation axis, font feature, font tracking setting, or font instance.

Font Name Language Constants

ATSUI identifies the language of a particular font name string by a constant of type `FontLanguageCode`. You can use one of these constants as part of your search criteria in the functions `ATSUFindFontFromName` (page 57) and `ATSUFindFontName` (page 63). The function `ATSUGetIndFontName` (page 61) passes back a constant of this type to represent the font name string language.

The font name language code identifies the language of the name string. You can pass the `kFontNoLanguage` constant if you do not care about the language of a font name string. In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first font in the name table that matches the other font name parameters that you specified.

```
enum {
    kFontNoLanguage                 = -1
    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;
```

Constant descriptions

`kFontNoLanguage` Identifies no language for the font name string.

`kFontEnglishLanguage`
Identifies English as the language of the font name string.

`kFontFrenchLanguage`
Identifies French as the language of the font name string.

`kFontGermanLanguage` Identifies German as the language of the font name string.

`kFontItalianLanguage`
Identifies German as the language of the font name string.

`kFontDutchLanguage` Identifies Dutch as the language of the font name string.

`kFontSwedishLanguage`
Identifies Swedith as the language of the font name string.

`kFontSpanishLanguage`
Identifies Spanish as the language of the font name string.

`kFontDanishLanguage`
Identifies Danish as the language of the font name string.

`kFontPortugueseLanguage`
Identifies Portuguese as the language of the font name string.

`kFontNorwegianLanguage`
Identifies Norwegian as the language of the font name string.

`kFontHebrewLanguage`
Identifies Hebrew as the language of the font name string.

`kFontJapaneseLanguage`
Identifies Japanese as the language of the font name string.

`kFontArabicLanguage`
Identifies Arabic as the language of the font name string.

`kFontFinnishLanguage`
Identifies Finnish as the language of the font name string.

`kFontGreekLanguage` Identifies Greek as the language of the font name string.

`kFontIcelandicLanguage`
Identifies Icelandic as the language of the font name string.

`kFontMalteseLanguage`
Identifies Maltese as the language of the font name string.

`kFontTurkishLanguage`
Identifies Turkish as the language of the font name string.

`kFontCroatianLanguage`
Identifies Croatian as the language of the font name string.

`kFontTradChineseLanguage`
Identifies traditional Chinese as the language of the font name string.

`kFontUrduLanguage` Identifies Urdu as the language of the font name string.

`kFontHindiLanguage` Identifies Hindi as the language of the font name string.

`kFontThaiLanguage` Identifies Thai as the language of the font name string.

`kFontKoreanLanguage` Identifies Korean as the language of the font name string.

`kFontLithuanianLanguage`
Identifies Lithuanian as the language of the font name string.

`kFontPolishLanguage`
Identifies Polish as the language of the font name string.

`kFontHungarianLanguage`
Identifies Hungarian as the language of the font name string.

`kFontEstonianLanguage`
Identifies Estonian as the language of the font name string.

`kFontLettishLanguage`
Identifies Lettish as the language of the font name string.

`kFontLatvianLanguage`
Identifies Latvian as the language of the font name string.

`kFontSaamiskLanguage`
Identifies Saamisk as the language of the font name string.

`kFontLappishLanguage`
Identifies Lappish as the language of the font name string.

`kFontFaeroeseLanguage`
Identifies the north Germanic language spoken on Faeroe Island as the language of the font name string.

`kFontFarsiLanguage` Identifies Persian Farsi as the language of the font name string.

`kFontPersianLanguage`
Identifies Persian as the language of the font name string.

`kFontRussianLanguage`
Identifies Russian as the language of the font name string.

`kFontSimpChineseLanguage`
Identifies simple Chinese as the language of the font name string.

`kFontFlemishLanguage`
Identifies Flemish (Dutch) as the language of the font name string.

`kFontIrishLanguage` Identifies Irish as the language of the font name string.

`kFontAlbanianLanguage`
Identifies Albanian as the language of the font name string.

`kFontRomanianLanguage`
Identifies Romanian as the language of the font name string.

`kFontCzechLanguage` Identifies Czech as the language of the font name string.

`kFontSlovakLanguage`
Identifies Slovak as the language of the font name string.

`kFontSlovenianLanguage`
Identifies Slovanian as the language of the font name string.

`kFontYiddishLanguage`
Identifies Yiddish as the language of the font name string.

`kFontSerbianLanguage`
Identifies Serbian as the language of the font name string.

`kFontMacedonianLanguage`
Identifies Macedonian as the language of the font name string.

`kFontBulgarianLanguage`
Identifies Bulgarian as the language of the font name string.

`kFontUkrainianLanguage`
Identifies Ukrainian as the language of the font name string.

`kFontByelorussianLanguage`
Identifies Russian as the language of the font name string.

`kFontUzbekLanguage` Identifies Uzbek as the language of the font name string.

`kFontKazakhLanguage`
Identifies Kazakh as the language of the font name string.

`kFontAzerbaijaniLanguage`
Identifies Azerbaiajani as the language of the font name string.

`kFontAzerbaijanArLanguage`
Identifies Arabic Azerbaijani as the language of the font name string.

`kFontArmenianLanguage`
Identifies Armenian as the language of the font name string.

`kFontGeorgianLanguage`
Identifies Georgian as the language of the font name string.

`kFontMoldavianLanguage`
Identifies Moldavian as the language of the font name string.

`kFontKirghizLanguage`
Identifies Kirghiz as the language of the font name string.

`kFontTajikiLanguage`
Identifies Tajiki as the language of the font name string.

`kFontTurkmenLanguage`
Identifies Turkmen as the language of the font name string.

`kFontMongolianLanguage`
Identifies Mongolian as the language of the font name string.

`kFontMongolianCyrLanguage`
Identifies Mongolian Cyrillic as the language of the font name string.

`kFontPashtoLanguage` Identifies Pashto as the language of the font name string.

`kFontKurdishLanguage`
Identifies Kurdish as the language of the font name string.

`kFontKashmiriLanguage`
Identifies Kashmiri as the language of the font name string.

`kFontSindhiLanguage` Identifies Sindhi as the language of the font name string.

`kFontTibetanLanguage`
Identifies Tibetan as the language of the font name string.

`kFontNepaliLanguage`
Identifies Nepali as the language of the font name string.

`kFontSanskritLanguage`
Identifies Sanskrit as the language of the font name string.

`kFontMarathiLanguage`
Identifies Marathi as the language of the font name string.

`kFontBengaliLanguage`
Identifies Bengali as the language of the font name string.

`kFontAssameseLanguage`
: Identifies Assamese as the language of the font name string.

`kFontGujaratiLanguage`
: Identifies Gujarati as the language of the font name string.

`kFontPunjabiLanguage`
: Identifies Punjabi as the language of the font name string.

`kFontOriyaLanguage` Identifies Oriya as the language of the font name string.

`kFontMalayalamLanguage`
: Identifies Malayalam as the language of the font name string.

`kFontKannadaLanguage`
: Identifies Kannada as the language of the font name string.

`kFontTamilLanguage` Identifies Tami as the language of the font name string.

`kFontTeluguLanguagE`
: Identifies Teluga as the language of the font name string.

`kFontSinhaleseLanguage`
: Identifies Sinhalese as the language of the font name string.

`kFontBurmeseLanguage`
: Identifies Burmese as the language of the font name string.

`kFontKhmerLanguage` Identifies Khmer as the language of the font name string.

`kFontLaoLanguage` Identifies Lao as the language of the font name string.

`kFontVietnameseLanguage`
: Identifies Vietnamese as the language of the font name string.

`kFontIndonesianLanguage`
: Identifies Indonesian as the language of the font name string.

`kFontTagalogLanguage`
: Identifies Tagalog as the language of the font name string.

`kFontMalayRomanLanguage`
: Identifies Roman Malay as the language of the font name string.

`kFontMalayArabicLanguage`
Identifies Arabic Malay as the language of the font name string.

`kFontAmharicLanguage`?
Identifies Amharic as the language of the font name string.

`kFontTigrinyaLanguage`
Identifies Tigrinya as the language of the font name string.

`kFontGallaLanguage` Identifies Galla as the language of the font name string.

`kFontOromoLanguage` Identifies Oromo as the language of the font name string.

`kFontSomaliLanguage`
Identifies Somali as the language of the font name string.

`kFontSwahiliLanguage`
Identifies Swahili as the language of the font name string.

`kFontRuandaLanguage`
Identifies Ruanda as the language of the font name string.

`kFontRundiLanguage` Identifies Rundi as the language of the font name string.

`kFontChewaLanguage`
Identifies Chewa as the language of the font name string.

`kFontMalagasyLanguage`
Identifies Malagasy as the language of the font name string.

`kFontEsperantoLanguage`
Identifies Esperanto as the language of the font name string.

`kFontWelshLanguage` Identifies Welsh as the language of the font name string.

`kFontBasqueLanguage`
Identifies Basque as the language of the font name string.

`kFontCatalanLanguage`
Identifies Catalan as the language of the font name string.

`kFontLatinLanguage` Identifies Latin as the language of the font name string.

`kFontQuechuaLanguage`
Identifies Quechua as the language of the font name string.

`kFontGuaraniLanguage`
Identifies Guarani as the language of the font name string.

`kFontAymaraLanguage`
Identifies Aymara as the language of the font name string.

`kFontTatarLanguage` Identifies Tatar as the language of the font name string.

`kFontUighurLanguage`
Identifies Uighur as the language of the font name string.

`kFontDzongkhaLanguage`
Identifies Dzongkha as the language of the font name string.

`kFontJavaneseRomLanguage`
Identifies Javanese as the language of the font name string.

`kFontSundaneseRomLanguage`
Identifies Sundanese as the language of the font name string.

Font Name Platform Constants

ATSUI identifies the encoding of a particular font name string by a constant of type `FontPlatformCode`. You can use one of these constants as part of your search criteria in the functions `ATSUFindFontFromName` (page 57) and `ATSUFindFontName` (page 63). The function `ATSUGetIndFontName` (page 61) passes back a constant of this type to represent the font name string encoding.

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.

IMPORTANT
Unicode-encoded font name entries have 8-bit instead of the expected 16-bit names. As a result, to locate a Unicode-encoded font name string, you must use the constant `kFontMacintoshPlatform` with the Unicode script code constant representing the script ID of Unicode encoding you want to find. ▲

You can pass the `kFontNoPlatform` constant if you do not care about the encoding of a font name string. In this case, `ATSUFindFontName` and

`ATSUFindFontFromName` will pass back the first font in the name table that matches the other font name parameters that you specified.

```
enum {
    kFontNoPlatform         = -1,
    kFontUnicodePlatform    = 0,
    kFontMacintoshPlatform  = 1,
    kFontReservedPlatform   = 2,
    kFontMicrosoftPlatform  = 3,
    kFontCustomPlatform     = 4
};
typedef UInt32          FontPlatformCode;
```

Constant descriptions

`kFontNoPlatform` Indicates that you do not care about the platform of the font name. In this case, you will get the first font name in the name table that matches the other parameters you specified. If you specify this constant, you should pass the constant `kFontNoScript` for the font name's script code.

`kFontUnicodePlatform`
Identifies the Unicode character code specification as the platform of the font name string.

`kFontMacintoshPlatform`
Identifies one of the Macintosh character code sets as the platform of the font name string.

`kFontReservedPlatform`
Reserved for future use.

`kFontMicrosoftPlatform`
Identifies one of the Microsoft character code sets as the platform of the font name string.

`kFontCustomPlatform`
Identifies the default platform as defined by the font for the platform of the font name string. If you specify this constant, the character encoding of a font does not correspond to a specific standard.

Fraction Font Feature Selector Constants

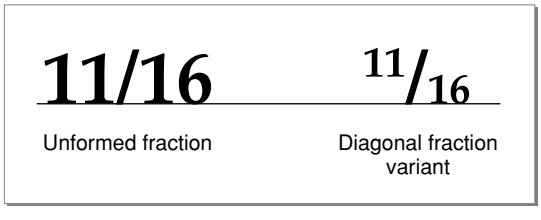
There are several ways to generate fractions with ATSUI. For a font that supports the fractions feature type, you may be able to select between two different types of automatic fraction generation, as shown in Table D-10.

Table D-10 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.

Figure D-5 shows the same fraction, drawn first with `kNoFractionsSelector` and then with `kDiagonalFractionsSelector`.

Figure D-5 Fractions



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. ◆

Ideographic Spacing Feature Selector Constants

The ideographic spacing feature type is used to select between full-width and proportional forms of ideographs (that is, Han-derived characters). Table D-11 lists the selectors for this feature.

Table D-11 Feature selectors for the `kIdeographicSpacingType` feature type

Constant	Explanation
`kFullWidthIdeographsSelector`	Selects the full width forms of ideographs.
`kProportionalIdeographsSelector`	Selects the proportional forms of ideographs.

Justification Override Mask Constants

You can use a mask constant of type `JustificationFlags` in the `growFlags` and `shrinkFlags` fields of a width delta override structure of a particular justification priority. There are four width delta override structures, one for each justification priority, in an array of structures of type `ATSJustPriorityWidthDeltaOverrides` (page 182).

In the `growFlags` field of the width delta override structure, these masks indicate whether ATSUI should apply the limits defined in the `beforeGrowLimit` and `afterGrowLimit` fields, as well as whether unlimited gap absorption should be applied to the priority of glyphs specified in the given width delta override structure.

In the `shrinkFlags` field, these masks indicate whether ATSUI should apply the limits defined in the `beforeShrinkLimit` and `afterShrinkLimit` fields, as well as

whether unlimited gap absorption should be applied to the priority of glyphs specified in the given width delta override structure.

Note that these mask constants 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`
: If the bit specified by this mask 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`
: If the bit specified by this mask is set, ATSUI uses the grow and shrink limit values set in the array of `ATSJustPriorityWidthDeltaOverrides` (page 182) structures. If this bit 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`
: If the bit specified by this mask is set, ATSUI takes into account the state of the `kJUSTUnlimited` mask constant. If this bit is cleared, the bit specified by the `kJUSTUnlimited` mask constant must also be set to 0.

`kJUSTUnlimited`
: If the bit specified by this mask is set, ATSUI distributes all remaining justification gap, even if it violates the grow or shrink limits specified in the width delta override structure specified in the array of `ATSJustPriorityWidthDeltaOverrides` (page 182) structures.

If this bit is set, you must also set the bit specified by the `kJUSTOverrideUnlimited` mask constant.

`kJUSTPriorityMask` If the bit specified by this mask is set, ATSUI identifies the new justification priority for the glyphs this width delta structure applies to. See "Justification Priority Constants" (page 289) for a description of possible values. Only a single valid justification priority value is permitted. If this bit is set, the bit specified by the mask constant `kJUSTOverrideLimit` must also be set.

Justification Priority Constants

Glyphs 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 highest to lowest. If you set the bit specified by the mask constant `kJUSTPriorityMask`, described in "Justification Override Mask Constants" (page 287), you can supply one of these constants of type `JustPCActionType` to set a new justification priority for the glyphs within the particular width delta override structure.

The types of 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 white space—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.

Kana Spacing Feature Selector Constants

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.

The Kana Spacing feature type is used to select widths specifically for Japanese Hiragana and Katakana characters. Table D-12 lists the selectors for this feature.

Table D-12 Feature selectors for the `kKanaSpacingType` feature type

Constant	Explanation
`kFullWidthKanaSelector`	Selects the full width forms of kana.
`kProportionalKanaSelector`	Selects the proportional forms of kana.

Letter Case Font Feature Selector Constants

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, you can select features that specify case changes such as those shown in Table D-13.

Table D-13 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. ◆

Ligature Font Feature Selector Constants

If the font supports the ligatures feature type, you can select features related to ligature formation, such as those shown in Table D-14.

Table D-14 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).
`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).

Table D-14 Feature selectors for the `kLigaturesType` feature type

Constant	Explanation
`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.

Figure D-6 shows several levels of ligature formation specified through ligature feature selectors.

Figure D-6 Levels of ligature formation controlled with ligature feature selectors

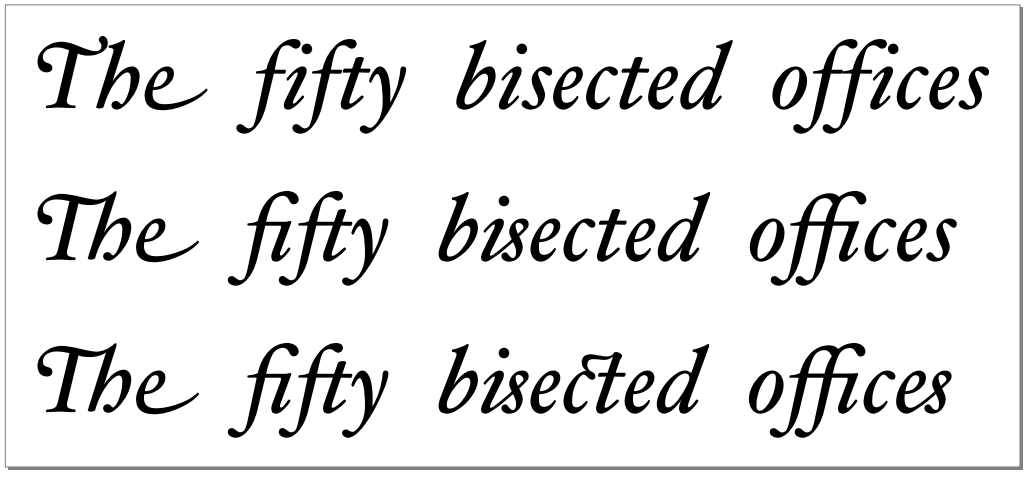
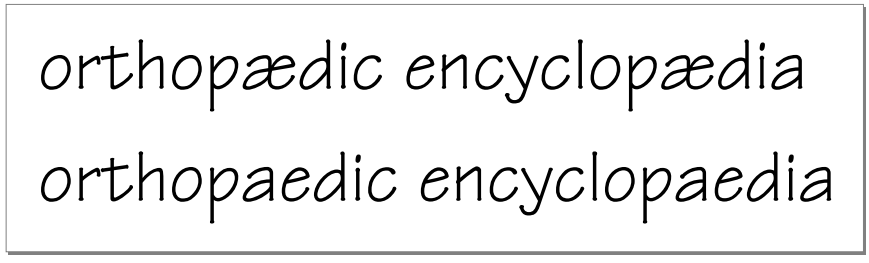


Figure D-7 shows the results of selection (upper) and deselection (lower) of diphthong ligatures.

Figure D-7 Use of diphthong ligatures



Linguistic Rearrangement Font Feature Selector Constants

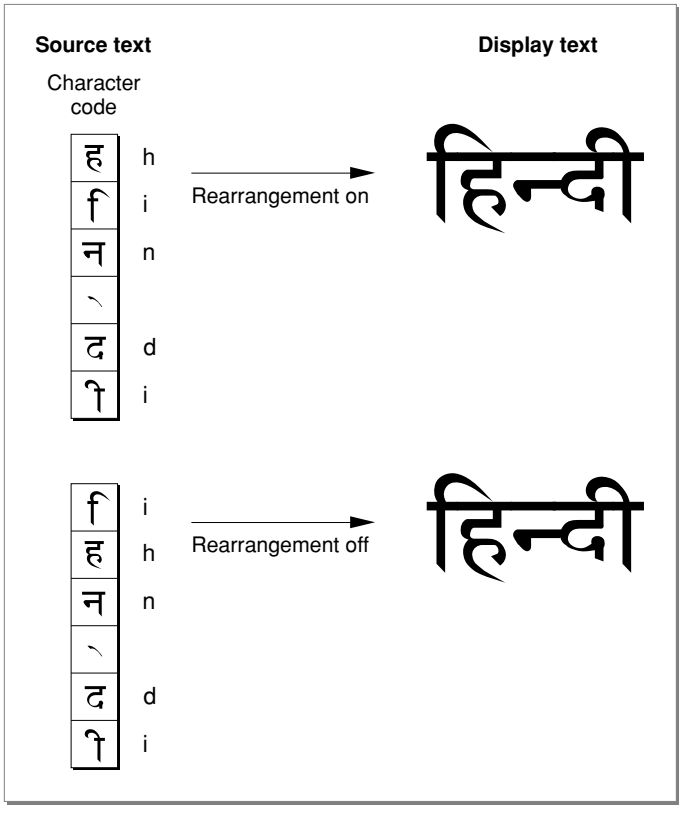
Linguistic (Indic-style) rearrangement is a standard feature of Devanagari and other South Asian scripts. However, users may not always want it to occur, preferring instead to enter characters in an "already reversed" order. If a font supports the rearrangement feature type, you can either allow the default behavior (which is to perform rearrangement) or you can prevent it. Table D-15 shows the feature selectors for rearrangement.

Table D-15 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.

Figure D-8 shows two examples of the display of the word "hindi", first with linguistic rearrangement on and then with it off. Note that when rearrangement is off, the storage order of the character codes in the source text must reflect display order, rather than normal input order.

Figure D-8 The word “hindi” drawn with rearrangement tuned on (upper) and off (lower)



Macintosh Platform Script Code Constants

ATSUI identifies the script ID of a particular font name string by a constant of type `FontScriptCode`. You can use one of these constants as part of your search criteria in the functions `ATSUFindFontFromName` (page 57) and `ATSUFindFontName` (page 63). The function `ATSUGetIndFontName` (page 61) passes back a constant of this type to represent the script ID of a font name string. Note that is the encoding of the font name string is not specified, you do not need to specify a script code.

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.

```
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` Identifies the Roman script on the Macintosh platform as the script ID of the font name string.

`kFontJapaneseScript` Identifies the Japanese script on the Macintosh platform as the script ID of the font name string.

`kFontTraditionalChineseScript` Identifies the traditional Chinese script on the Macintosh platform as the script ID of the font name string.

`kFontChineseScript` Identifies the Chinese script on the Macintosh platform as the script ID of the font name string.

`kFontKoreanScript` Identifies the Korean script on the Macintosh platform as the script ID of the font name string.

`kFontArabicScript` Identifies the Arabic script on the Macintosh platform as the script ID of the font name string.

`kFontHebrewScript` Identifies the Hebrew script on the Macintosh platform as the script ID of the font name string.

`kFontGreekScript` Identifies the Greek script on the Macintosh platform as the script ID of the font name string.

`kFontCyrillicScript` Identifies the Cyrillic script on the Macintosh platform as the script ID of the font name string.

`kFontRussian` Identifies the Russian script on the Macintosh platform as the script ID of the font name string.

`kFontRSymbolScript` Identifies the right-to-left symbol script on the Macintosh platform as the script ID of the font name string.

`kFontDevanagariScript` Identifies the Devanagari script on the Macintosh platform as the script ID of the font name string.

`kFontGurmukhiScript` Identifies the Gurmukhi script on the Macintosh platform as the script ID of the font name string.

`kFontGujaratiScript` Identifies the Gujarati script on the Macintosh platform as the script ID of the font name string.

`kFontOriyaScript` Identifies the Oriya font script on the Macintosh platform as the script ID of the font name string.

`kFontBengaliScript` Identifies the Benagli script on the Macintosh platform as the script ID of the font name string.

`kFontTamilScript` Identifies the Tamil script on the Macintosh platform as the script ID of the font name string.

`kFontTeluguScript` Identifies the Telugu script on the Macintosh platform as the script ID of the font name string.

`kFontKannadaScript` Identifies the Kannada script on the Macintosh platform as the script ID of the font name string.

`kFontMalayalamScript`
Identifies the Malayalam script on the Macintosh platform as the script ID of the font name string.

`kFontSinhaleseScript`
Identifies the Sinhalese script on the Macintosh platform as the script ID of the font name string.

`kFontBurmeseScript` Identifies the Burmese script on the Macintosh platform as the script ID of the font name string.

`kFontKhmerScript` Identifies the Khmer script on the Macintosh platform as the script ID of the font name string.

`kFontThaiScript` Identifies the Thai script on the Macintosh platform as the script ID of the font name string.

`kFontLaotianScript` Identifies the Laotian script on the Macintosh platform as the script ID of the font name string.

`kFontGeorgianScript`
Identifies the Georgian script on the Macintosh platform as the script ID of the font name string.

`kFontArmenianScript`
Identifies the Armenian script on the Macintosh platform as the script ID of the font name string.

`kFontSimpleChineseScript`
Identifies the simplified Chinese script on the Macintosh platform as the script ID of the font name string.

`kFontTibetanScript`
Identifies the Tibetan script on the Macintosh platform as the script ID of the font name string.

`kFontMongolianScript`
: Identifies the Mongolian script on the Macintosh platform as the script ID of the font name string.

`kFontGeezScript`
: Identifies the Ge'ez script on the Macintosh platform as the script ID of the font name string.

`kFontEthiopicScript`
: Identifies the Ethiopic script on the Macintosh platform as the script ID of the font name string.

`kFontAmharicScript`
: Identifies the Amharic script on the Macintosh platform as the script ID of the font name string.

`kFontSlavicScript`
: Identifies the Slavic script on the Macintosh platform as the script ID of the font name string.

`kFontEastEuropeanRomanScript`
: Identifies the East European script on the Macintosh platform as the script ID of the font name string.

`kFontVietnameseScript`
: Identifies the Vietnamese script on the Macintosh platform as the script ID of the font name string.

`kFontExtendedArabicScript`
: Identifies the extended Arabic script on the Macintosh platform as the script ID of the font name string.

`kFontSindhiScript`
: Identifies the Sindhi script on the Macintosh platform as the script ID of the font name string.

`kFontUninterpretedScript`
: Identifies an uninterpreted script on the Macintosh platform as the script ID of the font name string.

Mathematical Extras Feature Selector Constants

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-16.

Table D-16 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. ◆

Microsoft Platform Script Code Constants

```
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.

Number Case Feature Selector Constants

Some fonts support both lowercase (also called traditional or old-style) numerals, in which some glyphs extend below the baseline, and uppercase (also

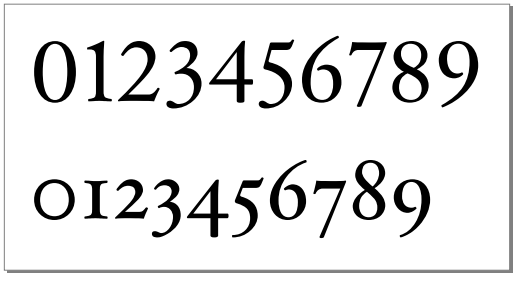
called **lining**) numerals, in which no glyphs extend below the baseline. For fonts that support the number case feature type, you can select either kind of numeral. Table D-17 lists the selectors for this feature.

Table D-17 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.

Figure D-9 shows both kinds of numerals.

Figure D-9 Uppercase and lowercase numerals



Number Width Feature Selector Constants

Many fonts support both proportional-width and fixed-width numerals, as shown in Figure D-10. 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

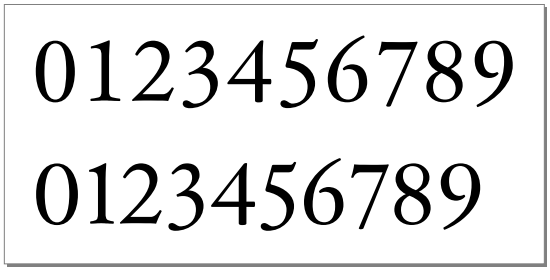
fixed-width or proportional-width numerals. Table D-18 lists the selectors for this feature.

Table D-18 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.

Figure D-10 shows both kinds of numerals.

Figure D-10 Fixed-width and proportional-width numerals



Ornament Sets Feature Selector Constants

Fonts may include ornamental, non alphabetic glyph sets used for various purposes. With a font that supports the ornament set feature type, you may be

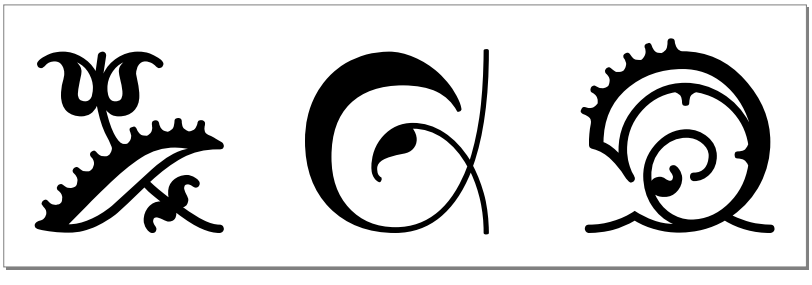
able to select among those glyph sets, using selectors such as those shown in Table D-19.

Table D-19 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.

Figure D-11 shows an example of glyphs from an ornamental set.

Figure D-11 Ornamental glyphs



Prevention of Glyph Overlap Font Feature Selector Constants

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”.

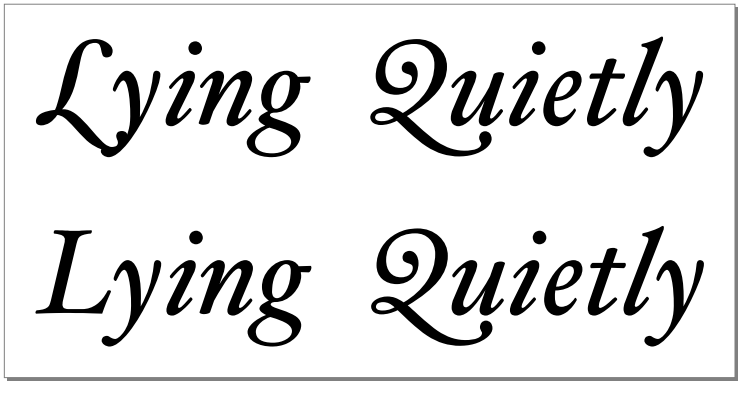
For fonts that support the glyph overlap feature type, you can specify that no glyph may overlap the outline of the following glyph. If it does, a non-overlapping form of the glyph is substituted. Table D-20 lists the selectors for this feature.

Table D-20 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.

In the case of Figure D-12, for example, preventing glyph overlap means that the script “Q” can remain because the following “u” has no descenders to collide with it, whereas the script “L” is replaced with a simpler form to avoid collision with the “y”.

Figure D-12 Allowing and preventing glyph overlap



Style Options Feature Selector Constants

An ATSUI-compatible font may offer named sets of non contextual 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-21.

Table D-21 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.

Swash Font Feature Selector Constants

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

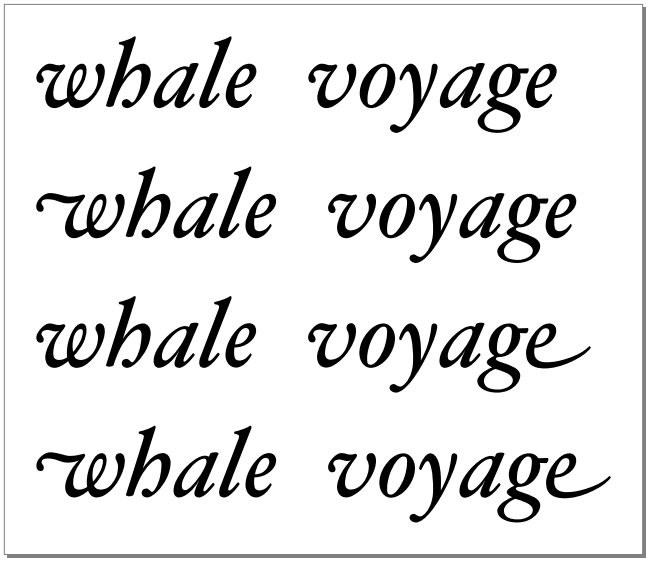
are not. If the font supports the smart swashes feature type, you can select features that allow you to specify sets of swashes, such as shown in Table D-22.

Table D-22 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.
`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

Figure D-13 shows the same phrase written four times: first without swash variants, then with line initials, then with line finals, and finally with both line initials and line finals.

Figure D-13 Specifying different swashes with feature selectors



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. ◆

Text Width Feature Selector Constants

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 Feature Selector

Constants" (page 290). This is an exclusive feature type. Table D-23 lists the selectors for this feature.

Table D-23 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.

Transliteration Feature Selector Constants

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-24 lists the selectors for this feature.

Table D-24 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.
`kKanaToRomanizationSelector`	Allows text in Kana to be displayed using Romanization.
`kRomanizationToHiraganaSelector`	Allows text in Romanization to be displayed using Hiragana.

Table D-24 Feature selectors for the `kTransliterationType` feature type (continued)

Constant	Explanation
`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.

Typographic Extras Feature Selector Constants

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-25.

Table D-25 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.

Table D-25 Feature selectors for the `kTypographicExtrasType` feature type (continued)

Constant	Explanation
`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.

Unicode Decomposition Feature Selector Constants

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-26.

Table D-26 Feature selectors for the `kUnicodeDecompositionType` feature type

Constant
`kCanonicalDecompositionOnSelector`
`kCanonicalDecompositionOffSelector`
`kCompatibilityDecompositionOnSelector`
`kCompatibilityDecompositionOffSelector`
`kTranscodingDecompositionOnSelector`
`kTranscodingDecompositionOffSelector`

Unicode Platform Script Code Constants

ATSUI identifies the script ID of a particular font name string by a constant of type `FontScriptCode`. You can use one of these constants as part of your search criteria in the functions `ATSUFindFontFromName` (page 57) and `ATSUFindFontName` (page 63). The function `ATSUGetIndFontName` (page 61) passes back a constant of this type to represent the script ID of a font name string. Note that is the encoding of the font name string is not specified, you do not need to specify a script code.

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.

You can pass the `kFontNoScript` constant if you do not care about the script ID of a font name string. In this case, `ATSUFindFontName` and `ATSUFindFontFromName` will pass back the first font in the name table that matches the other font name parameters that you specified. These constants are available with ATSUI 1.0.

```
enum {
    kFontNoScript                   = -1,
    kFontUnicodeDefaultSemantics    = 0,
    kFontUnicodeV1_1Semantics       = 1,
    kFontISO10646_1993Semantics     = 2,
    kFontUnicodeV2BasedSemantics    = 3
```

Constant descriptions

`kFontNoScript` Indicates that you do not care about the script ID of the font name. In this case, you will get the first font name in the name table that matches the other parameters you specified.

`kFontUnicodeDefaultSemantics`
Identifies the default Unicode character code specification as the platform version of the font name string.

`kFontUnicodeV1_1Semantics`
Identifies version 1.1 of the Unicode character code specification as the platform version of the font name string.

`kFontISO10646_1993Semantics`
Identifies the ISO/IEC 10646-1993 specification as the platform version of the font name string.

`kFontUnicodeV2BasedSemantics`
Identifies version 2.0 or later of the Unicode character code specification as the platform version of the font name string.

Vertical Position Font Feature Selector Constants

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-27.

Table D-27 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").

Vertical Substitution Font Feature Selector Constants

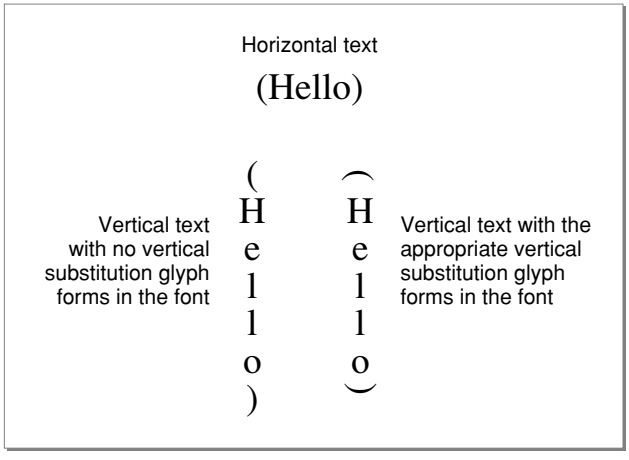
Vertical substitution is a glyph substitution in which the glyph for a given glyph code is replaced by an alternate form in a vertical line. (This is not the same as rotating the glyph.) Table D-28 shows the feature selectors for vertical substitution.

Table D-28 Feature selectors for the `kVerticalSubstitutionType` feature type

Constant	Explanation
`kSubstituteVerticalFormsOnSelector` `kSubstituteVerticalFormsOffSelector`	Allows or prevents the substitution of alternate glyph forms in vertical lines.

Figure D-14 illustrates how vertical substitution works.

Figure D-14 Vertical substitution forms in a font



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.

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

V–Z