Attributed Strings Programming Guide

Cocoa > Data Management

2007-06-04

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

Simultaneously published in the United States and Canada.

Contents

Tables

Introduction to Attributed Strings Programming Guide

Attributed Strings Programming Guide describes the attributed string objects, instantiated from the `NSAttributedString` class or the CFAttributedString Core Foundation opaque type, which manage sets of text attributes, such as font and kerning, that are associated with character strings or individual characters.

Who Should Read This Document

You should read this document if you need to work directly with attributed string objects.

Organization of This Document

This programming topic contains the following articles:

- "Attributed Strings" (page 9) describes the attributed string objects instantiated from `NSAttributedString`, `NSMutableAttributedString`, CFAttributedString and CFMutableAttributedString.
- "Creating Attributed Strings in Cocoa" (page 11) describes how to create attributed strings with data that you provide.
- "Accessing Attributes" (page 13) describes how to access text attributes.
- "Changing an Attributed String" (page 17) describes how to change characters and attributes in an attributed string.
- "Drawing Attributed Strings" (page 19) describes how to draw an attributed string in a view.
- "RTF Files and Attributed Strings" (page 21) explains how to read and write attributed strings to and from files of RTF data, and it describes Apple's extensions to the RTF language.
- "Word and Line Calculations in Attributed Strings" (page 29) describes how to work with attributed strings in editors.
- "Standard Attributes" (page 31) describes global `NSString` constants containing the attribute names.

See Also

For more information, refer to the following documents:

- *String Programming Guide for Cocoa* describes the string objects that hold the Unicode character information in attributed strings.

- *Text Attributes* explains how the text system handles the various kinds of attributes applied to strings of text.

Attributed Strings

Attributed string objects manage character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string. The classes `NSAttributedString` and `NSMutableAttributedString` declare the programmatic interface for read-only attributed strings and modifiable attributed strings, respectively. The Foundation Kit defines the basic functionality, while additional Objective-C methods are defined in the Application Kit. The Application Kit also uses a subclass of `NSMutableAttributedString`, called `NSTextStorage`, to provide the storage for the extended text-handling system (see *Text System Storage Layer Overview*).

`NSAttributedString` and `NSMutableAttributedString` are toll-free bridged to their Core Foundation counterparts, CFAttributedString and CFMutableAttributedString respectively. This means that a Foundation attributed string is interchangeable in function or method calls with the corresponding bridged Core Foundation type. Therefore, in a method where you see an `NSMutableAttributedString *` parameter, you can pass in a variable of type `CFMutableAttributedStringRef`, and in a function where you see a `CFAttributedStringRef` parameter, you can pass in an instance of `NSAttributedString` (or `NSMutableAttributedString`).

`NSAttributedString` is not a subclass of `NSString`. It contains an `NSString` object to which it applies attributes. This protects users of attributed strings from ambiguities caused by the semantic differences between simple and attributed strings. For example, equality can't be simply defined between an `NSString` and an attributed string. The attributed string classes adopt the `NSCopying` and `NSMutableCopying` protocols, making it convenient to convert an attributed string from one type to the other.

`NSAttributedString` and `NSMutableAttributedString` add a number of features to the basic content storage of `NSString`:

- Association of arbitrary, programmer-defined attributes with ranges of characters.
- Preservation of attribute-to-character mapping after changes (`NSMutableAttributedString`).
- Support for RTF, including file attachments and graphics.
- Drawing in `NSView` objects (note that the Application Kit adds drawing methods to `NSString` as well)
- Linguistic unit (word) and line calculation.

An attributed string identifies attributes by name, storing their values as opaque `id`s in an `NSDictionary` object. For example, the text font is stored as an `NSFont` object under the name given by `NSFontAttributeName`. You can associate any object value, by any name, with a given range of characters in the attributed string.

A mutable attributed string keeps track of the attribute mapping as characters are added to and deleted from it and as attributes are changed. It allows you to group batches of edits with the `beginEditing` and `endEditing` methods, and to consolidate changes to the attribute-to-character mapping with the `fix...` methods.

Creating Attributed Strings in Cocoa

You create an `NSAttributedString` object in a number of different ways:

- You can create a new string with the `initWithString:`, `initWithString:attributes:`, or `initWithAttributedString:` method. These methods initialize an attributed string with data you provide, as illustrated in the following example:

```
NSFont *font = [NSFont fontWithName:@"Palatino-Roman" size:14.0];
NSDictionary *attrsDictionary =
        [NSDictionary dictionaryWithObject:font
                                    forKey:NSFontAttributeName];
NSAttributedString *attrString =
    [[NSAttributedString alloc] initWithString:@"strigil"
            attributes:attrsDictionary];
```

  For a list of attributes provided by the Application Kit framework see the Constants section in NSAttributedString Additions.

  The attribute values assigned to an attributed string become the property of that string, and should not be modified "behind the attributed string" by other objects. Doing so can render inconsistent the attributed string's internal state. Always use `NSMutableAttributedString`'s `setAttributes:range:` and related methods to change attribute values. See "Changing an Attributed String" (page 17) for more details.

- You can can create an attributed string from rich text (RTF) or rich text with attachments (RTFD) data using the initialization methods, `initWithRTF:documentAttributes:`, `initWithRTFD:documentAttributes:`, and `initWithRTFDFileWrapper:documentAttributes:`, as illustrated in the following example:

```
NSData *rtfData = ...;  // assume rtfData is an NSData object containing valid
 RTF data
NSDictionary *docAttributes;
NSSize paperSize;

NSAttributedString *attrString;

if ((attrString = [[NSAttributedString alloc]
        initWithRTF: data documentAttributes: &docAttributes])) {

    NSValue *value = [docAttrs objectForKey:@"PaperSize"];
    paperSize = [value sizeValue];
    // implementation continues...
```

- You can can create an attributed string from HTML data using the initialization methods `initWithHTML:documentAttributes:` and `initWithHTML:baseURL:documentAttributes:`. The methods return text attributes defined by the HTML as the attributes of the string. They return document-level attributes defined by the HTML, such as paper and margin sizes, by reference to an `NSDictionary` object, as described in "RTF Files and Attributed Strings" (page 21). The methods translate HTML as well as possible into structures of the Cocoa text system, but the Application Kit does not provide complete, true rendering of arbitrary HTML.

Accessing Attributes

An attributed string identifies attributes by name, storing a value under the attribute name in an `NSDictionary` object, which is in turn associated with an `NSRange` that indicates the characters to which the dictionary's attributes apply. You can assign any attribute name-value pair you wish to a range of characters, in addition to the standard attributes.

Retrieving Attribute Values

With an immutable attributed string, you assign all attributes when you create the string. In Java, you use the constructors. In Objective-C, you use methods such as `initWithString:attributes:`, which explicitly take an `NSDictionary` object of name-value pairs, or `initWithString:`, which assigns no attributes. And the Application Kit's extensions to `NSAttributedString` adds methods that take an RTF file or an HTML file. See "Changing an Attributed String" (page 17) for information on assigning attributes with a mutable attributed string.

To retrieve attribute values from either type of attributed string, use any of these methods:

```
attributesAtIndex:effectiveRange:
attributesAtIndex:longestEffectiveRange:inRange:
attribute:atIndex:effectiveRange:
attribute:atIndex:longestEffectiveRange:inRange:
fontAttributesInRange:
rulerAttributesInRange:
```

The first two methods return all attributes at a given index, the `attribute:...` methods return the value of a single named attribute. The Application Kit's extensions to `NSAttributedString` add `fontAttributesInRange:` and `rulerAttributesInRange:`, which return attributes defined to apply only to characters or to whole paragraphs, respectively.

The first four methods also return by reference the effective range and the longest effective range of the attributes. These ranges allow you to determine the extent of attributes. Conceptually, each character in an attributed string has its own collection of attributes; however, it's often useful to know when the attributes and values are the same over a series of characters. This allows a routine to progress through an attributed string in chunks larger than a single character. In retrieving the effective range, an attributed string simply looks up information in its attribute mapping, essentially the dictionary of attributes that apply at the index requested. In retrieving the longest effective range, the attributed string continues checking characters past this basic range as long as the attribute values are the same. This extra comparison increases the execution time for these methods but guarantees a precise maximal range for the attributes requested.

Effective and Maximal Ranges

Methods that return an effective range by reference are not guaranteed to return the maximal range to which the attribute(s) apply; they are merely guaranteed to return some range over which they apply. In practice they will return whatever range is readily available from the attributed string's internal storage mechanisms, which may depend on the implementation and on the precise history of modifications to the attributed string.

Methods that return a longest effective range by reference, on the other hand, are guaranteed to return the longest range containing the specified index to which the attribute(s) in question apply (constrained by the value of the argument passed in for `inRange:`). For efficiency, it is important that the `inRange:` argument should be as small as appropriate for the range of interest to the client.

When you iterate over an attributed string by attribute ranges, either sort of method may be appropriate depending on the situation. If there is some processing to be done for each range, and you know that the full range for a given attribute is going to have to be handled eventually, it may be more efficient to use the longest-effective-range variant, so as not to have to handle the range in pieces. However, you should use the longest-effective-range methods with caution, because the longest effective range could be quite long—potentially the entire length of the document, if the `inRange:` argument is not constrained.

The Objective-C code fragment below progresses through an attributed string in chunks based on the effective range. The fictitious analyzer object here counts the number of characters in each font. The while loop progresses as long as the effective range retrieved does not include the end of the attributed string, retrieving the font in effect just past the latest retrieved range. For each font attribute retrieved, the analyzer tallies the number of characters in the effective range. In this example, it is possible that consecutive invocations of `attribute:atIndex:effectiveRange:` will return the same value.

```
NSAttributedString *attrStr;
unsigned int length;
NSRange effectiveRange;
id attributeValue;

length = [attrStr length];
effectiveRange = NSMakeRange(0, 0);

while (NSMaxRange(effectiveRange) < length) {
    attributeValue = [attrStr attribute:NSFontAttributeName
        atIndex:NSMaxRange(effectiveRange) effectiveRange:&effectiveRange];
    [analyzer tallyCharacterRange:effectiveRange font:attributeValue];
}
```

In contrast, the next Objective-C code fragment progresses through the attributed string according to the maximum effective range for each font. In this case, the analyzer counts font changes, which may not be represented by merely retrieving effective ranges. In this case the while loop is predicated on the length of the limiting range, which begins as the entire length of the attributed string and is whittled down as the loop progresses. After the analyzer records the font change, the limit range is adjusted to account for the longest effective range retrieved.

```
NSAttributedString *attrStr;
NSRange limitRange;
NSRange effectiveRange;
id attributeValue;

limitRange = NSMakeRange(0, [attrStr length]);
```

```
while (limitRange.length > 0) {
    attributeValue = [attrStr attribute:NSFontAttributeName
        atIndex:limitRange.location longestEffectiveRange:&effectiveRange
        inRange:limitRange];
    [analyzer recordFontChange:attributeValue];
    limitRange = NSMakeRange(NSMaxRange(effectiveRange),
        NSMaxRange(limitRange) - NSMaxRange(effectiveRange));
}
```

Note that the second code fragment is more complex. Because of this, and because `attribute:atIndex:longestEffectiveRange:inRange:` is somewhat slower than `attribute:atIndex:effectiveRange:`, you should typically use it only when absolutely necessary for the work you're performing. In most cases working by effective range is enough.

Changing an Attributed String

`NSMutableAttributedString` declares a number of methods for changing both characters and attributes. You must take care not to modify attribute values after they have been passed to an attributed string. You may also need to repair inconsistencies that can be introduced if you modify an attributed string.

Modifying Attributes

`NSMutableAttributedString` declares a number of methods for changing both characters and attributes, such as the primitive `replaceCharactersInRange:withString:` and `setAttributes:range:`, or the more convenient methods `addAttribute:value:range:`, `applyFontTraits:range:`, and so on.

The following example illustrates how to specify a link attribute for a selected range in an attributed string, underline the text, and color it blue. Note that *you can define whatever value you want for the link attribute, it is up to you to interpret the value when the link is selected*—see "Accessing Attributes" (page 13)—typically, however, you use either a string or an URL. For an explanation of the role of `beginEditing` and `endEditing` (shown in the sample), see "Fixing Inconsistencies" (page 18).

```
NSMutableAttributedString *string; // assume string exists
NSRange selectedRange; // assume this is set

NSURL *linkURL = [NSURL URLWithString:@"http://www.apple.com/"];

[string beginEditing];
[string addAttribute:NSLinkAttributeName
               value:linkURL
               range:selectedRange];

[string addAttribute:NSForegroundColorAttributeName
               value:[NSColor blueColor]
               range:selectedRange];

[string addAttribute:NSUnderlineStyleAttributeName
               value:[NSNumber numberWithInt:NSSingleUnderlineStyle]
               range:selectedRange];
[string endEditing];
```

Attribute values assigned to an attributed string become the property of that string, and should not be modified "behind the attributed string" by other objects. Doing so can render inconsistent the attributed string's internal state. There are two main reasons for this:

- How an attribute value propagates through an attributed string is not predictable. If you change the value, you might be editing more of the attributed string than you thought. In fact the value could have been copied to the undo stack, or to a totally different document, and so on.
- Attributed strings do caching and uniquing of attributes, which assumes attribute values do not change. The assumption is that `isEqual:` and `hash` on attribute values will not change once the attribute value has been set.

If you must change attribute values, and are sure that the change will apply to the correct range, there are two strategies you can adopt:

- Use an attribute value whose `isEqual:` and `hash` do not depend on the values you are modifying.
- Use indirection: use the attribute value as a lookup key into a table where the actual value can be changed. For instance, this might be the appropriate approach for having a "stylesheet"-like attribute.

Fixing Inconsistencies

All of the methods for changing a mutable attributed string properly update the mapping between characters and attributes, but after a change some inconsistencies can develop. Here are some examples of attribute consistency requirements:

- Paragraph styles must apply to entire paragraphs.
- Scripts may only be assigned fonts that support them. For example, Kanji and Arabic characters can't be assigned the Times-Roman font, and must be reassigned fonts that support these scripts.
- Deleting attachment characters from the string requires the corresponding attachment objects to be released. Similarly, removing attachment objects requires the corresponding attachment characters to be removed from the string.
- A code editing application that displays all language keywords in boldface can automatically assign this attribute as the user changes the font or edits the text.

The Application Kit's extensions to `NSMutableAttributedString` define methods to fix these inconsistencies as changes are made. This allows the attributes to be cleaned up at a low level, hiding potential problems from higher levels and providing for very clean update of display as attributes change. There are four methods for fixing attributes and two to group editing changes:

```
fixAttributesInRange:
fixAttachmentAttributeInRange:
fixFontAttributeInRange:
fixParagraphStyleAttributeInRange:
beginEditing
endEditing
```

The first method, `fixAttributesInRange:`, invokes the other three `fix...` methods to clean up deleted attachment references, font attributes, and paragraph attributes, respectively. The individual method descriptions explain what cleanup entails for each case.

`NSMutableAttributedString` provides `beginEditing` and `endEditing` methods for subclasses of `NSMutableAttributedString` to override. These methods allow instances of a subclass to record or buffer groups of changes and clean themselves up on receiving an `endEditing` message. The `endEditing` method also allows the receiver to notify any observers that it has been changed. `NSTextStorage`'s implementation of `endEditing`, for example, fixes changed attributes and then notifies its layout managers that they need to re-lay and redisplay their text. The default implementations do nothing.

Drawing Attributed Strings

The Application Kit's NSStringDrawing extensions let you draw an attributed string in a focused graphics context (typically an NSView) using a number of methods: `drawAtPoint:`, `drawInRect:`, and (with Mac OS X v10.4 and later) `drawWithRect:options:`. These methods are designed for drawing small amounts of text or text that must be drawn rarely. They create and dispose of various supporting text objects every time you call them. To draw strings repeatedly, it is more efficient to use NSLayoutManager, as described in "Drawing Strings".

Note that the Application Kit defines drawing methods for NSString as well, allowing any string object to draw itself. These methods, `drawAtPoint:withAttributes:`, `drawInRect:withAttributes:`, and (with Mac OS X v10.4 and later) `drawWithRect:options:attributes:`, are described in NSString Additions.

With Mac OS X v10.4 and later, you can find out the rectangle required to lay out an attributed string using the method, `boundingRectWithSize:options:`. Again, there is an analogous method to determine the rectangle required to render an NSString object, given a set of attributes—`boundingRectWithSize:options:attributes:`.

RTF Files and Attributed Strings

Rich Text Format (RTF) is a text formatting language devised by Microsoft Corporation. You can represent character, paragraph, and document format attributes using plain text with interspersed RTF commands, groups, and escape sequences. RTF is widely used as a document interchange format to transfer documents with their formatting information across applications and computing platforms. The Application Kit has support for reading and writing RTF. For text attributes not available in standard RTF, Apple has extended RTF with custom commands.

Reading and Writing RTF Data

The Application Kit's extensions for `NSAttributedString` add support for reading text attributes from, and writing them to, RTF files or RTFD (rich text with attachments) files.

Important: The Application Kit extensions write the standard character-level attributes from the attributed string and the standard document-level attributes from the document attributes dictionary; however, custom attributes that you define and add to an attributed string are not written to the RTF file. Standard character-level attribute keys are described in "Standard Attributes" (page 31), and the document attributes are described in Table 1 (page 22).

The `NSAttributedString` methods for writing rich text are defined in *NSAttributedString Application Kit Additions Reference*:

`RTFFromRange:documentAttributes:`	Returns an NSData object that contains an RTF stream corresponding to the characters and attributes within the given range, omitting all attachment attributes.
`RTFDFromRange:documentAttributes:`	Returns an NSData object that contains an RTFD stream corresponding to the characters and attributes within aRange.
`RTFDFileWrapperFromRange:documentAttributes:`	Returns an NSFileWrapper object that contains an RTFD document corresponding to the characters and attributes within the given range.
`initWithRTF:documentAttributes:`	Initializes a new NSAttributedString object by decoding the stream of RTF commands and data contained in the given data object.
`initWithRTFD:documentAttributes:`	Initializes a new NSAttributedString object by decoding the stream of RTFD commands and data contained in the given data object.

`initWithRTFDFileWrapper:documentAttributes:`	Initializes a new NSAttributedString object from the given NSFileWrapper object containing an RTFD document.

In addition to these explicit RTF-reading methods, four methods implicitly allow loading RTF data from a file or URL-specified resource. `NSAttributedString` defines:

`initWithPath:documentAttributes:`	Initializes a new NSAttributedString object from RTF or RTFD data contained in the file at the given path.
`initWithURL:documentAttributes:`	Initializes a new NSAttributedString object from the data at the given URL.

`NSMutableAttributedString` defines:

`readFromURL:options:documentAttributes:`	Sets the contents of receiver from the file at url.
`readFromData:options:documentAttributes:`	Sets the contents of the receiver from the stream at data.

Handling Document Attributes

Attributed strings store attribute information for characters and paragraphs only, while RTF also supports more general attributes of a document, such as paper size and page layout. The Application Kit methods that work with RTF read and write some RTF directives for document attributes, stored in an `NSDictionary` object.

Many `init` methods return a dictionary containing the attributes read from RTF data, which you can use to set up a page layout. Similarly, RTF extraction methods such as `RTFFromRange:documentAttributes:`, accept a dictionary containing those attributes and write them into the RTF data, thus preserving the page layout information.

Table 1 lists the RTF document attributes supported by the Application Kit.

Table 1 Document attributes supported by RTF-handling methods

Attribute Key	Type
`PaperSize`	`NSValue`, containing `NSSize`
`LeftMargin`	`NSNumber`, containing a float, in points
`RightMargin`	`NSNumber`, containing a float, in points
`TopMargin`	`NSNumber`, containing a float, in points
`BottomMargin`	`NSNumber`, containing a float, in points
`HyphenationFactor`	`NSNumber`, containing a float

Attribute Key	Type
DocumentType	NSString; may be NSPlainTextDocumentType, NSRTFTextDocumentType, NSRTFDTextDocumentType, NSMacSimpleTextDocumentType, or NSHTMLTextDocumentType.
CharacterEncoding	NSNumber, containing an int specifying the NSStringEncoding used to interpret the file; for plain text files only.
ViewSize	NSValue, containing NSSize.
ViewZoom	NSValue, containing a float. 100 = 100% zoom.
ViewMode	NSValue, containing an int. 0 = normal; 1 = page layout (use value of PaperSize attribute).
CocoaRTFVersion	NSNumber, containing an int. If RTF file, stores the version of Cocoa with which the file was created. Absence of this value indicates RTF file not created by Cocoa or its predecessors. 0 = Not Cocoa writer, 1 = NextStep, 40 = OpenStep, 100 = Mac OS X 10.0, 102 = 10.2. (Other than incrementing the number for future versions, no assumptions should be made as to how the number will change in the future.)
Converted	NSNumber, containing an int. Indicates whether the file was converted by a filter service. If missing or zero, the file was originally in the format specified by document type. If 1 or more, it was converted to this type by a filter service. If negative, the file was converted "lossily," meaning that some features of the original document were left out.

Handling Attachments

Attachments, such as embedded images or files, are represented in an attributed string by both a special character and an attribute. The character is identified by the global name NSAttachmentCharacter, and indicates the presence of an attachment at its location in the string. The attribute, identified in the string by the attribute name NSAttachmentAttributeName, is an NSTextAttachment object . An NSTextAttachment object contains the data for the attachment itself, as well as an image to display when the string is drawn.

You can use NSAttributedString's attributedStringWithAttachment: class method to construct an attachment string, which you can then add to a mutable attributed string using appendAttributedString: or insertAttributedString:atIndex:. To write rich text data containing one or more attachments, use the RTFDFromRange:documentAttributes: method and the RTFDFileWrapperFromRange:documentAttributes: method. To initialize an attributed string with rich text data containing attachments, use the initWithRTFD:documentAttributes:, and initWithRTFDFileWrapper:documentAttributes: methods.

Apple's RTF Extensions

Apple has extended the RTF language to support text attributes and formatting constructs available in the Cocoa text system but not representable with standard RTF. The Apple extensions take the same form as standard RTF commands, groups, and escapes. RTF commands consist of a backslash followed by a string of alphabetic characters (case sensitive) followed by an optional integer parameter value which can be positive or negative. RTF groups begin with a left brace ({), followed by RTF sequences optionally including other groups, closed by a right brace (}). RTF escapes consist of a backslash followed by a special character, such as `\{`, which indicates a literal left brace instead of the beginning of a group.

RTF includes the concept of a *destination*, which is a group containing an RTF command and text possibly to be inserted at a different location in a document, such as a footnote. The escape sequence `\*` indicates that RTF readers that don't understand the command that follows should ignore the contents of the destination.

Dimensions in RTF are expressed in *twips*—one twip is one twentieth of a point.

Table 2 lists Apple's RTF extensions for character attributes.

Table 2 Character attribute RTF extensions

RTF Sequence	Description	Parameter(s)
\CocoaLigature*N*	Ligature control	Value of `NSLigatureAttributeName`. 0 = no ligatures, 1 = default ligatures, 2 = all ligatures. Default value 1.
\expansion*N*	Expansion factor to be applied to glyphs	2000 * value of `NSExpansionAttributeName` (log of expansion factor). Default value 0.
\obliqueness*N*	Skew to be applied to glyphs	2000 * value of `NSObliquenessAttributeName`. 0 = no skew. Default value 0.
\fsmilli*N*	A finer specification for font size	1000 * font size. Written in addition to \fs when \fs is not an integral or half-point value; value is overridden by \fs, so this should be written immediately after \fs. Default driven by \fs.
\shadx*N* \shady*N*	Shadow offset, written in conjunction with \shad	X and Y offsets in twips (0 = no offset). Defaults are \shadx3 and \shady-3.
\shadr*N*	Shadow blur, written in conjunction with \shad	Blur radius in twips. 0 = no blur. Default value 0.
\strikec*N*	Strikethrough color	Color number. Default same as foreground text color.

RTF Sequence	Description	Parameter(s)
\strikestyle*N*	Strikethrough style, written where \strike, \striked, \strikew are not sufficient	Style and pattern mask, value of `NSObliquenessAttributeName`. 0 = none; 0x8000 = by word; styles: 1 = single, 2 = thick, 9 = double; patterns: 0x100 = dotted, 0x200 = dash, 0x300 = dash dot, 0x400 = dash dot dot. Default value 0.
\strokec*N*	Stroke color	Color number. Default same as foreground text color.
\strokewidth*N*	Glyph stroke width, written in conjunction with \outl.	20 * stroke width as percentage of font point size. 0 = no stroke. Default value 0. Negative values indicate that glyphs are both stroked and filled; the stroke width is taken from the absolute value of the parameter.
\ulstyle*N*	Underline style, written where the standard \ul commands are not sufficient	Style and pattern mask, value of `NSUnderlineStyleAttributeName`. 0 = none; 0x8000 = by word; styles: 1 = single, 2 = thick, 9 = double; patterns: 0x100 = dotted, 0x200 = dash, 0x300 = dash dot, 0x400 = dash dot dot. Default value 0.
{\NeXTGraphic *attachment* \width*N* \height*N*} *string*}	Name of attachment file in the same folder as the RTF file (typically packaged within an RTFD document)	The *attachment* is the attachment file name, encoded in UTF-8 and properly RTF-escaped. The width and height parameters optionally specify the attachment size in twips. The *string* is always 0xAC.
{}{*\glid*N* *basestring*}*string*}	Glyph ID for explicitly specified glyphs. (The extra {} pair is necessary to work around an RTF reader bug in Mac OS X version 10.2 and earlier.)	Glyph identifier (parameter to \glid). The *basestring* is the string the glyph id is intended to override; this attribute is then applied to the specified *string*. Typically *string* and *basestring* are the same, although *string* might contain multiple instances of *basestring*.
{}{*\glid*N* *basestring*\glcol*N*} *string*}	Glyph ID for explicitly specified glyphs	Character identifier (parameter to \glid) and character collection (parameter to \glcol). Collection IDs: 0 = identity, 1 = Adobe-CNS1, 2 = Adobe-GB1, 3 = Adobe-Japan1, 4 = Adobe-Japan2, 5 = Adobe-Korea.
{}{*\glid *basestring*\glnam *glyphname*}*string*}	Glyph ID for explicitly specified glyphs	The *glyphname* is the glyph name in UTF-8 encoding.

RTF Sequence	Description	Parameter(s)
\AppleTypeServicesU*N*	Character shape control	Value of `NSCharacterShapeAttributeName`. The value is interpreted as Apple Type Services kCharacterShapeType selector + 1. The value 0 disables this attribute. Default value 0.

Table 3 lists Apple's RTF extensions for paragraph attributes.

Table 3 Paragraph attribute RTF extensions

RTF Sequence	Description	Parameter(s)
\pardeftab*N*	Default tab interval for paragraph	Tab interval value in twips. 0 = no tabs other than those explicitly specified. Default value 0.
\qnatural	Natural text alignment for paragraph (based on script), written along with \ql	None
\slleading*N*	Paragraph line spacing (NSParagraphStyle lineSpacing method)	Line spacing value in twips. Default value 0.
\slmaximum*N*	Maximum line height (NSParagraphStyle maximumLineHeight method), written along with \sl and if needed \slmult	Maximum line height value in twips. Default value 0, implying no maximum.
\slminimum*N*	Minimum line height (NSParagraphStyle minimumLineHeight method), written along with \sl and if needed \slmult	Minimum line height value in twips. Default value 0.

Table 4 lists Apple's RTF extensions for document attributes.

Table 4 Document attribute RTF extensions

RTF Sequence	Description	Parameter(s)
\readonlydoc*N*	Read-only document. This has nothing to do with the file system permissions or ownership of the file; it's just a hint that indicates that the document should be presented in a read-only fashion to the user, if the viewer or editor is capable.	0 = Not read-only, 1 = read-only. Default value 0.

RTF Sequence	Description	Parameter(s)
\cocoartf*N*	Cocoa RTF-writer version number. This is a number used by Apple to indicate the version number of the RTF writer used to write this document.	Incrementing version number. 0 = Not Cocoa writer, 1 = NextStep, 40 = OpenStep, 100 = Mac OS X 10.0, 102 = 10.2. (Other than incrementing the number for future versions, no assumptions should be made as to how the number will change in the future.) Default value 0, although some heuristics are used to recognize pre–Mac OS X documents as such.
\viewh*N* \vieww*N*	Size of display area (not window or view size) to be used for displaying the document	Display area dimension in twips. Default value unspecified.

Word and Line Calculations in Attributed Strings

The Application Kit's extensions to `NSAttributedString` support the typical behavior of text editors in selecting a word on a double-click with the `doubleClickAtIndex:` method, and finds word breaks with `nextWordFromIndex:forward:`. It also calculates line breaks with the `lineBreakBeforeIndex:withinRange:` method.

Standard Attributes

The identifiers listed in Table 1 are global `NSString` constants containing the attribute names. The value class is the class of the value corresponding to that attribute.

Table 1 Table of standard attributes

Attribute Identifier	Value Class	Default Value
`NSAttachmentAttributeName`	`NSTextAttachment`	none (no attachment)
`NSBackgroundColorAttributeName`	`NSColor`	none (no background)
`NSBaselineOffsetAttributeName`	`NSNumber`, as a float	0.0
`NSFontAttributeName`	`NSFont`	Helvetica 12-point
`NSForegroundColorAttributeName`	`NSColor`	black
`NSKernAttributeName`	`NSNumber`, as a float	0.0
`NSLigatureAttributeName`	`NSNumber`, as an int	1 (standard ligatures)
`NSLinkAttributeName`	`id`	none (no link)
`NSParagraphStyleAttributeName`	`NSParagraphStyle`	(as returned by `NSParagraphStyle`'s `defaultParagraphStyle` method)
`NSSuperscriptAttributeName`	`NSNumber`, as an int	0
`NSUnderlineStyleAttributeName`	`NSNumber`, as an int	none (no underline)

The natures of several attributes are not obvious from name alone:

- The baseline offset attribute is a literal distance, in pixels, by which the characters should be shifted above the baseline (for positive offsets) or below (for negative offsets).
- The kerning attribute indicates how much the following character should be shifted from its default offset as defined by the current character's font; a positive kern indicates a shift farther along and a negative kern indicates a shift closer to the current character.
- The ligature attribute determines what kinds of ligatures should be used when displaying the string. A value of 0 indicates that only ligatures essential for proper rendering of text should be used, 1 indicates that standard ligatures should be used, and 2 indicates that all available ligatures should be used. Which ligatures are standard depends on the script and possibly the font. Arabic text, for example, requires ligatures for many character sequences, but has a rich set of additional ligatures that combine characters. English text has no essential ligatures, and typically has only two standard ligatures, those for "fi" and "fl"—all others being considered more advanced or fancy.

- The link attribute specifies an arbitrary object that is passed to the `NSTextView` method `clickedOnLink:atIndex:` when the user clicks in the text range associated with the `NSLinkAttributeName` attribute. The text view's delegate object can implement `textView:clickedOnLink:atIndex:` or `textView:clickedOnLink:` to process the link object. Otherwise, the default implementation checks whether the link object is an `NSURL` object and, if so, opens it in the URL's default application.
- The superscript attribute indicates an abstract level for both super- and subscripts. The user of the attributed string can interpret this as desired, adjusting the baseline by the same or a different amount for each level, changing the font size, or both.
- The underline attribute has only two values defined, `NSNoUnderlineStyle` and `NSSingleUnderlineStyle`, but these can be combined with `NSUnderlineByWordMask` and `NSUnderlineStrikethroughMask` to extend their behavior. By bitwise-ORing these values in different combinations, you can specify no underline, a single underline, a single strikethrough, both an underline and a strikethrough, and whether the line is drawn for whitespace or not.

Document Revision History

This table describes the changes to *Attributed Strings Programming Guide*.

Date	Notes
2007-06-04	Added links to methods defined in AppKit Extension.
2006-11-07	Moved table of standard attributes into separate article. Added statement that custom attributes are not written out with RTF data.
2006-07-24	Augmented "Creating Attributed Strings in Cocoa" with code samples.
2006-01-10	Clarified use of NSLinkAttributeName in sample code.
2005-08-11	Added references to CFAttributedString; clarified the distinction between "effective" and "longest effective" ranges.
2005-04-29	Updated for Mac OS X v10.4 and added references to CFAttributedString. Included description of restrictions on modifying attribute values. Added example of setting attributes. Changed title from "Attributed Strings."
2004-04-14	Added section describing Apple's RTF extensions to the article "RTF Files and Attributed Strings" (page 21).
2004-02-05	Rewrote introduction and added an index.
2002-11-12	Revision history was added to existing topic.

Index