Text Attributes

Cocoa > Text & Fonts

2004-02-16

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

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction to Text Attributes

Text Attributes describes the text-related attributes maintained by the Cocoa text system. Text attributes provide the distinguishing characteristics of rich text and other formatting information for paragraphs and documents.

Who Should Read This Document

You should read this document to understand the different types of text attributes in the text system, especially if you deal directly with attributed strings and need to understand how the text system manages their attributes.

To understand the information in this document, you should have prior general knowledge of the text system's capabilities and architecture, as well as basic Cocoa programming conventions.

Organization of This Document

This document includes the following articles:

- "About Text Attributes" (page 9) introduces and defines the five types of text-related attributes used in Cocoa. It also provides cross-references to more detailed documentation.
- "Setting Text Attributes" (page 13) explains how you can programmatically set the attributes of text displayed in a text view object using the methods of NSTextView and its superclass NSText.
- "Accessing Attributes" (page 15) describes the attributes stored with an attributed string and explains how to manipulate them.
- "Changing an Attributed String" (page 19) describes the methods available to alter the characters and attributes of an NSMutableAttributedString. This article also discusses attribute fixing.
- "Plain and Rich Text Objects" (page 21) discusses text attributes of the rich text format (RTF) standard recognized by text objects.
- "RTF Files and Attributed Strings" (page 23) explains how to read and write character and document attributes to RTF files.

See Also

For more information, refer to the following documents:

- *Text System Overview* provides more information about how the text system stores and manipulates text.

- *Attributed Strings Programming Guide* presents more detailed information about text strings and attributes.

About Text Attributes

The Cocoa text system handles five kinds of text attributes: character attributes, temporary attributes, paragraph attributes, glyph attributes, and document attributes. Character attributes include traits such as font, color, and subscript, which can be associated with an individual character or a range of characters. Temporary attributes are character attributes that apply only to a particular layout and are not persistent. Paragraph attributes are traits such as indentation, tabs, and line spacing. Glyph attributes affect the way the layout manager renders glyphs and include traits such as overstriking the previous glyph. Document attributes include document-wide traits such as paper size, margins, and view zoom percentage.

This article provides a brief introduction to the various types of text attributes with cross references to more detailed documentation.

Character Attributes

The text system stores character attributes persistently in attributed strings along with the characters to which they apply. The text system's predefined character attributes control the appearance of characters (font, foreground color, background color, and ligature handling) and their placement (superscript, baseline offset, and kerning).

Two special character attributes pertain to links and attachments. A link attribute points to a URL (encapsulated in an NSURL object) or any other object of your choice. An attachment attribute is associated with a special attachment character and points to an NSFileWrapper object containing the attached file or in-memory data.

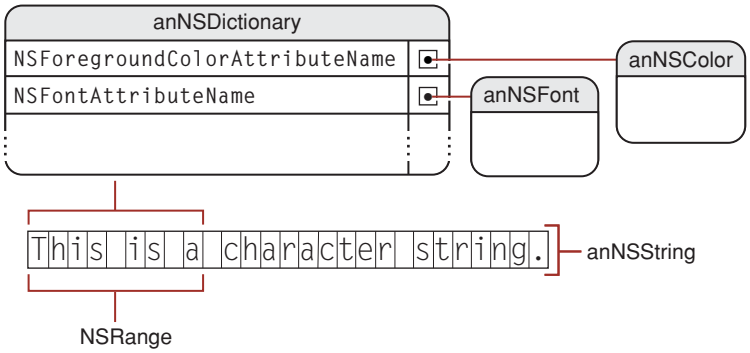
Two of the predefined character attributes, `NSCharacterShapeAttributeName`and `NSGlyphInfoAttributeName`, are rarely used but described here for completeness. `NSCharacterShapeAttributeName`enables you to set a value for the character shape feature used in font rendering by Apple Type Services. This feature is currently used to specify traditional shapes in Chinese and Japanese scripts, but font developers could use it for other scripts as well.

In Mac OS X version 10.2 and later, the predefined character attribute `NSGlyphInfoAttributeName`points to an NSGlyphInfo object that provides a means to override the standard glyph generation process and substitute a specified glyph over the attribute's range.

Storing Character Attributes

An attributed string stores character attributes as key-value pairs in NSDictionary objects. The key is an attribute name, represented by an identifier (an NSString constant) such as `NSFontAttributeName`. Figure 1 shows an attributed string with an attribute dictionary applied to a range within the string.

Figure 1 Composition of an NSAttributedString including its attributes dictionary



Conceptually, each character in an attributed string has an associated dictionary of attributes. Typically, however, an attribute dictionary applies to a longer range of characters. The NSAttributedString class provides methods that take a character index and return the associated attribute dictionary and the range to which its attribute values apply. See "Accessing Attributes" (page 15) for more information about using these methods.

In addition to the predefined attributes, you can assign any attribute key-value pair you wish to a range of characters. You add the attributes to the appropriate character range in the NSTextStorage object using the NSMutableAttributedString`addAttribute:value:range:` method. You can also create an NSDictionary containing the names and values of a set of custom attributes and add them to the character range in a single step using the `addAttributes:range:` method. To make use of your custom attributes, you need a custom subclass of NSLayoutManager that understands what to do with them. Your subclass should override the `drawGlyphsForGlyphRange:atPoint:` method first to call the superclass to draw the glyph range, then draw your own attributes on top, or else draw the glyphs entirely your own way.

Attribute Fixing

Editing attributed strings can cause inconsistencies that must be cleaned up by **attribute fixing**. The Application Kit extensions to NSMutableAttributedString define `fix...` methods to fix inconsistencies among attachment, font, and paragraph attributes. These methods ensure that attachments don't remain after their attachment characters are deleted, that font attributes apply only to characters available in that font, and that paragraph attributes are consistent throughout paragraphs.

See *Attributed Strings Programming Guide* for more details about character attributes and attribute fixing.

Temporary Attributes

Temporary attributes are character attributes that are not stored with an attributed string. Rather, the layout manager assigns temporary attributes during the layout process and uses them only when drawing the text. For example, you can use temporary attributes to underline misspelled words or color key words in a programming language.

Temporary attributes affect only the appearance of text, not the way in which it is laid out. You store temporary attributes in an NSDictionary using the same keys as regular character attributes, or using custom attribute names (if you have an NSLayoutManager subclass that can handle them). Then you add the attributes using

an NSLayoutManager method such as `addTemporaryAttributes:forCharacterRange:`. By default, the only temporary attributes recognized are those affecting color and underlines. During layout, temporary attributes supersede regular character attributes. So, for example, if a character has a stored `NSForegroundColorAttributeName`value specifying blue and a temporary attribute of the same identifier specifying red, then the character is rendered in red.

For more information on temporary attributes, see the NSLayoutManager reference documentation.

Paragraph Attributes

Paragraph attributes affect the way the layout manager arranges lines of text into paragraphs on a page. The text system encapsulates paragraph attributes in objects of the NSParagraphStyle class. The value of one of the predefined character attributes, `NSParagraphStyleAttributeName`, points to an NSParagraphStyle object containing the paragraph attributes for that character range. Attribute fixing ensures that only one NSParagraphStyle object pertains to the characters throughout each paragraph.

Paragraph attributes include traits such as alignment, tab stops, line-breaking mode, and line spacing (also known as leading). Users of text applications control paragraph attributes through ruler views, defined by the NSRulerView class.

See *Rulers and Paragraph Styles* for more details about paragraph attributes.

Glyph Attributes

Glyphs are the concrete representations of characters that the text system actually draws on a display. Glyph attributes are not complex data structures like character attributes but are simply integer values that the layout manager uses to denote special handling for particular glyphs during rendering.

The text system uses glyph attributes rarely, and applications should have little reason to be concerned with them. Nonetheless, NSLayoutManager provides public methods that handle glyph attributes, so you can use subclasses to extend the mechanism to handle custom glyph attributes if necessary.

The glyph generator sets built-in glyph attributes as required on glyphs during typesetting. They are maintained in the layout manager's glyph cache during that process, but they are not stored persistently. Two examples of glyph attributes are the elastic attribute for spaces, used to lay out fully justified text, and the attribute `NSGlyphAttributeInscribe`, which is used for situations such as drawing an umlaut over a character when the font does not include a built-in character-with-umlaut.

For more information about glyph attributes, see the NSLayoutManager reference documentation describing the `setIntAttribute:value:forGlyphAtIndex:` method.

Document Attributes

Document attributes pertain to a document as a whole. Document attributes include traits such as paper size, margins, and view zoom percentage. Although the text system has no built-in mechanism to store document attributes, initialization methods such as `initWithRTF:documentAttributes:` can populate

an NSDictionary object that you provide with document attributes derived from a stream of RTF or HTML data. Conversely, methods that write RTF data, such as `RTFFromRange:documentAttributes:`, write document attributes if you pass a reference to an NSDictionary object containing them with the message.

See "RTF Files and Attributed Strings" (page 23) and the NSAttributedString Additions reference documentation for more information.

Setting Text Attributes

NSTextView allows you to change the attributes of its text programmatically through various methods, most inherited from the superclass, NSText. NSTextView adds its own methods for setting the attributes of text that the user types, for setting the baseline offset of text as an absolute value, and for adjusting kerning and use of ligatures. Most of the methods for changing attributes are defined as action methods and apply to the selected text or typing attributes for a rich text view, or to all of the text in a plain text view.

An NSTextView maintains a set of typing attributes (font, size, color, and so on) that it applies to newly entered text, whether typed by the user or pasted as plain text. It automatically sets the typing attributes to the attributes of the first character immediately preceding the insertion point, of the first character of a paragraph if the insertion point is at the beginning of a paragraph, or of the first character of a selection. The user can change the typing attributes by choosing menu commands and using utilities such as the Font panel (Fonts window). You can also set the typing attributes programmatically using `setTypingAttributes:`, though you should rarely find need to do so unless creating a subclass.

NSText defines the action methods `superscript:`, `subscript:`, and `unscript:`, which raise and lower the baseline of text by predefined increments. NSTextView gives you much finer control over the baseline offset of text by defining the `raiseBaseline:` and `lowerBaseline:` action methods, which raise or lower text by one point each time they're invoked.

Kerning

NSTextView provides convenient action methods for adjusting the spacing between characters. By default, an NSTextView object uses standard kerning (as provided by the data in a font's AFM file). A `turnOffKerning:` message causes this kerning information to be ignored and the selected text to be displayed using nominal widths. The `loosenKerning:` and `tightenKerning:` methods adjust kerning values over the selected text and `useStandardKerning:` reestablishes the default kerning values.

Kerning information is a character attribute that's stored in the text view's NSTextStorage object. If your application needs finer control over kerning than the methods of this class provide, you should operate on the NSTextStorage object directly through methods defined by its superclass, NSMutableAttributedString. See the reference documentation for NSAttributedString Additions for information on setting attributes.

Ligatures

NSTextView's support for ligatures provides the minimum required ligatures for a given font and script. The required ligatures for a specific font and script are determined by the mechanisms that generate glyphs for a specific language. Some scripts may well have no ligatures at all—English text, as an example, doesn't require ligatures, although certain ligatures such as "fi" and "fl" are desirable and are used if they're available. Other scripts, such as Arabic, demand that certain ligatures must be available even if a `turnOffLigatures:`

message is sent to the NSTextView. Other scripts and fonts have standard ligatures that are used if they're available. The `useAllLigatures:` method extends ligature support to include all possible ligatures available in each font for a given script.

Ligature information is a character attribute that's stored in the text view's NSTextStorage object. If your application needs finer control over ligature use than the methods of this class provide, you should operate on the NSTextStorage object directly through methods defined by its superclass, NSMutableAttributedString. See the reference documentation for NSAttributedString Additions for information on setting attributes.

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 19) 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 15)—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 20).

```
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.

Plain and Rich Text Objects

Text objects such as NSText and NSTextView can contain either plain text or rich text. Plain text objects allow only one set of text attributes for all of their text; rich text objects allow multiple fonts, sizes, indents, and other attributes for different sets of characters and paragraphs. You can control whether a text object is plain or rich using the `setRichText:` method. Rich text objects are also capable of allowing the user to drag images and files into them. This behavior is controlled by the `setImportsGraphics:` method.

A rich NSText object can use RTF (Rich Text Format) as an interchange format. Not all RTF control words are supported, however. On input, an NSText object ignores any control word it doesn't recognize; some of those it can read and interpret it doesn't write out. Table 1 lists the RTF control words that any text object recognizes. Subclasses may recognize more.

Table 1 RTF control words recognized by all text objects

Control word	Can be written out
`\ansi`	yes
`\b`	yes
`\cb`	yes
`\cf`	yes
`\colortbl`	yes
`\dnn`	yes
`\fin`	yes
`\fn`	yes
`\fonttbl`	yes
`\fsn`	yes
`\i`	yes
`\lin`	yes
`\margrn`	yes
`\paperwn`	yes
`\mac`	no
`\margln`	yes
`\par`	yes

Control word	Can be written out
\pard	no
\pca	no
\qc	yes
\ql	yes
\qr	yes
\sn	no
\tab	yes
\upn	yes

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" in *Attributed Strings Programming Guide*, and the document attributes are described in Table 1 (page 24).

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 aRange, 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 aRange.
`initWithRTF:documentAttributes:`	Initializes a new NSAttributedString by decoding the stream of RTF commands and data contained in rtfData.
`initWithRTFD:documentAttributes:`	Initializes a new NSAttributedString by decoding the stream of RTFD commands and data contained in rtfdData.

`initWithRTFDFileWrapper:documentAttributes:`	Initializes a new NSAttributedString from wrapper, an 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 from RTF or RTFD data contained in the file at path.
`initWithURL:documentAttributes:`	The contents of aURL are examined to best load the file in whatever format it's in.

`NSMutableAttributedString` defines:

`readFromURL:options:documentAttributes:`	Sets the contents of receiver from the file at url.
`readFromData:options:documentAttributes:`	On return, the documentAttributes dictionary (if provided) contains the various keys described in the "Constants" section of NSAttributedString Additions.

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.

Document Revision History

This table describes the changes to *Text Attributes*.

Date	Notes
2004-02-16	Rewrote introduction and added an index.
2003-05-06	First content added to this programming topic. Renamed from *Character Attributes* to *Text Attributes*.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

Index

U