
ATSUI Programming Guide

[Carbon > Text & Fonts](#)



2007-07-10



Apple Inc.
© 2002, 2007 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Carbon, Geneva, Mac, Mac OS, Macintosh, New York, Quartz, QuickDraw, and TrueType are trademarks of Apple Inc., registered in the United States and other countries.

Finder and Numbers are trademarks of Apple Inc.

Helvetica, Palatino, and Times are registered trademarks of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO

THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction to ATSUI Programming Guide 11**

- Organization of This Document 11
- Overview of ATSUI 12
 - ATSUI and Font Data 13
 - ATSUI and Quartz 13
- See Also 13

Chapter 1 **Typography Concepts 15**

- Characters, Glyphs, and Fonts 15
- Character Encoding and Glyph Codes 16
- Unicode 17
- Text Storage 18
- Text Measurements 19
- Typestyles 21
 - Font Variations 21
 - Font Instances 21
- Text Layout 22
 - Text Direction 22
 - Baselines 23
 - Leading Edges and Trailing Edges 24
 - Text Runs, Style Runs, and Direction Runs 25
 - Contextual Forms and Ligatures 26
 - Alignment and Justification 26
 - Kerning and Tracking 27
 - Special Font Features 28
 - Line Breaking 29
 - Font Substitution 30
- Caret Handling 30
 - Caret Positioning 30
 - Highlighting 33
 - Converting Screen Position to Text Offset 34

Chapter 2 **ATSUI Style and Text Layout Objects 37**

- Style Objects 37
 - Style Attributes 38
 - Font Variations 53
 - Font Features 54
- Text Layout Objects 54
 - Text Information 55

Style Run Information 57
 Line and Layout Attributes 57

Chapter 3 Basic Tasks: Working With Objects and Drawing Text 69

Guidelines for Using ATSUI 70
 Creating Style Objects and Setting Attributes 70
 Creating a Text Layout Object and Setting Attributes 73
 Determining Paragraphs in a Unicode Text Block 74
 Drawing Horizontal Text 76
 Drawing Text Using a Quartz Context 77
 Drawing Equations 79
 Drawing Vertical Text 81
 Breaking Lines 83
 Measuring Text 85
 Calculating Line Height 86
 Flowing Text Around a Graphic 88
 Setting Up a Tab Ruler 96

Chapter 4 Interactive Tasks: Supporting Carets and Highlighting Text 99

Positioning the Caret 99
 Setting the Insertion Point 99
 Moving the Caret 101
 Highlighting Selected Text 101

Chapter 5 Advanced Tasks: Substituting Fonts and Modifying Layouts 103

Using Font Fallback Objects 103
 Setting a Baseline 105
 Kerning Text 108
 Adjusting Interglyph Positions 110
 Retrieving Glyph Metrics 111

Chapter 6 Direct-Access Tasks: Working With Glyph Data 113

Overriding ATSUI Layout Operations 113
 Guidelines for Overriding Layout Operations 117
 Extending the Space Between Glyphs 117
 Positioning Glyphs Along a Curve 120
 Retrieving and Drawing Glyph Outlines 124

Appendix A ATSUI Implementation of the Unicode Specification 131

Unsupported Control Characters 131
 Surrogates 132

Character Properties 133

Appendix B Font Features 135

- All-Typographic Features Feature Type 135
- Annotation Feature Type 135
- Cursive Connection Feature Type 137
- Character Alternatives Feature Type 137
- Character Shape Feature Type 138
- CJK Italic Roman Feature Type 139
- CJK Roman Spacing Feature Type 139
- Design Complexity Feature Type 140
- Diacritical Marks Feature Type 141
- Fractions Feature Type 142
- Ideographic Spacing Feature Type 142
- Kana Spacing Feature Type 143
- Letter Case Feature Type 143
- Ligatures Feature Type 144
- Linguistic Rearrangement Feature Type 146
- Mathematical Extras Feature Type 147
- Number Case Feature Type 148
- Number Spacing Feature Type 149
- Ornament Sets Feature Type 150
- Overlapping Glyphs Feature Type 151
- Ruby Kana Feature Type 151
- Smart Swashes Feature Type 152
- Style Options Feature Type 154
- Text Spacing Feature Type 154
- Transliteration Feature Type 155
- Typographical Extras Feature Type 156
- Unicode Decomposition Feature Type 157
- Vertical Position Feature Type 158
- Vertical Substitution Feature Type 158

Document Revision History 161

ATSUI Glossary 163

Figures, Tables, and Listings

Introduction Introduction to ATSUI Programming Guide 11

Figure I-1 ATSUI and text drawing in Mac OS X 12

Chapter 1 Typography Concepts 15

Figure 1-1 Contextual forms in a Roman font 16
Figure 1-2 Elements that distinguish glyphs of a Roman font 16
Figure 1-3 Input order and display order 18
Figure 1-4 Terms for glyph measurements 20
Figure 1-5 Image bounding rectangle and typographic bounding rectangle 21
Figure 1-6 Font variations along a variation axis 21
Figure 1-7 Bidirectional text 23
Figure 1-8 Baselines for different sizes of a glyph and for different writing systems 24
Figure 1-9 The baseline delta 24
Figure 1-10 Drop capitals 24
Figure 1-11 Leading edges and trailing edges 25
Figure 1-12 Three style runs in a line of text 25
Figure 1-13 Two direction runs in a line of text 25
Figure 1-14 Contextual forms of the Arabic letter “ha” 26
Figure 1-15 Examples of Roman ligatures 26
Figure 1-16 Left, center, and right text alignment (or flushness) 27
Figure 1-17 Glyphs with and without kerning 27
Figure 1-18 Tracking settings for normal, tight, and loose tracking 28
Figure 1-19 Text width and line breaking 29
Figure 1-20 Caret position and insertion point 30
Figure 1-21 Caret positions at direction boundaries 31
Figure 1-22 Dual caret at direction boundaries in mixed-directional text 32
Figure 1-23 Single carets at direction boundaries in mixed-directional text 33
Figure 1-24 Highlighting a selection range in unidirectional text 33
Figure 1-25 Highlighting a selection range in bidirectional text 34
Figure 1-26 Interpreting caret position from a mouse-down event 35
Figure 1-27 Mouse-down regions and caret positions in bidirectional text 36
Table 1-1 Special font features 28

Chapter 2 ATSUI Style and Text Layout Objects 37

Figure 2-1 Triples for text color and font size style attributes 38
Figure 2-2 Straight and angled caret positions 39
Figure 2-3 Cross-stream kerning applied to a minus sign in an equation 40
Figure 2-4 Using a font transformation matrix to achieve text effects 41
Figure 2-5 Vertical and horizontal glyphs 42

Figure 2-6	A line of text without and with an imposed glyph width	43
Figure 2-7	Hanging punctuation glyphs	43
Figure 2-8	Effects of changing the hanging punctuation attribute	44
Figure 2-9	A question mark forced to extend beyond the margin	44
Figure 2-10	When kerning can and cannot occur	45
Figure 2-11	Caret position between two kerned glyphs	46
Figure 2-12	Apparent kerning by a glyph that extends beyond its advance width	46
Figure 2-13	Apparent misalignment of curved letters	48
Figure 2-14	Optical alignment at line edges	48
Figure 2-15	Text drawn with three different tracking settings	51
Figure 2-16	Negative and positive with-stream shift	52
Figure 2-17	Combined with-stream and cross-stream shift	53
Figure 2-18	Caret position between with-stream shifted glyphs	53
Figure 2-19	A range of text in a text block	56
Figure 2-20	Text offsets of type <code>UniCharArrayOffset</code>	56
Figure 2-21	Bidirectional text	57
Figure 2-22	Text that has twelve style runs	57
Figure 2-23	Use of the alignment attribute with text whose width is shorter than the line's width	58
Figure 2-24	Text with multiple baselines aligned to the default baseline	59
Figure 2-25	Alignment and justification of Roman text	60
Figure 2-26	Alignment and justification in Arabic	60
Figure 2-27	Use of the justification line and layout control attribute	61
Figure 2-28	Justification and alignment in a text layout object	62
Figure 2-29	A horizontal line and a line rotated -90 degrees	67
Table 2-1	Style attribute tags, data types, and default values for text styles	51
Table 2-2	Interactions between the width, justification, and alignment attributes	62
Table 2-3	Line layout option flags	65

Chapter 3

Basic Tasks: Working With Objects and Drawing Text 69

Figure 3-1	Text drawn at an angle	78
Figure 3-2	Rotated text	79
Figure 3-3	A scientific equation drawn by adjusting cross-stream shift values	79
Figure 3-4	A rotated line of text that does not use vertical forms of the glyphs	82
Figure 3-5	A rotated line of text that uses vertical forms of the glyphs	82
Figure 3-6	Text flowed around a graphic	89
Figure 3-7	Variables needed for line breaking	91
Listing 3-1	Code that keeps style objects around until the application quits	72
Listing 3-2	A function to find a paragraph in a block of Unicode text	74
Listing 3-3	Using a Quartz context to rotate text	78
Listing 3-4	A code fragment that performs line breaking	84
Listing 3-5	Calculating line height in Mac OS X version 10.2	86
Listing 3-6	Calculating line height in Mac OS 8, Mac OS 9, and CarbonLib	87
Listing 3-7	Code that flows text around a graphic	91
Listing 3-8	Setting tab values for a text layout object	96

Listing 3-9 Obtaining an array of tab values 97

Chapter 4 **Interactive Tasks: Supporting Carets and Highlighting Text 99**

Figure 4-1 Caret positions returned for an insertion point on a direction boundary 100
 Figure 4-2 The anchor point and active end of a selection range 102

Chapter 5 **Advanced Tasks: Substituting Fonts and Modifying Layouts 103**

Figure 5-1 Text drawn using a Roman baseline and a hanging baseline for capitals 108
 Table 5-1 Common baseline classes 106
 Listing 5-1 Code that creates and sets up a font fallback object for a text layout object 104
 Listing 5-2 A function that sets the baseline class for a style object 107
 Listing 5-3 A function that sets baseline values for a text layout object 107
 Listing 5-4 A function that sets kerning inhibition for a style object 109
 Listing 5-5 A function that sets a tracking value 110
 Listing 5-6 Code that applies a tracking setting to a line of text 110

Chapter 6 **Direct-Access Tasks: Working With Glyph Data 113**

Figure 6-1 Unadjusted text compared to text with modified advance delta values 114
 Figure 6-2 Text drawn with replacement characters in place of space characters 114
 Figure 6-3 Text with modified baseline delta values 115
 Figure 6-4 Text positioned along a sine curve 121
 Table 6-1 Some values you can modify using direct-access functions 114
 Table 6-2 Data selectors used to obtain glyph data 116
 Table 6-3 Selectors for layout operations 116
 Listing 6-1 A callback that modifies advance delta values 118
 Listing 6-2 A function that installs a justification override callback 119
 Listing 6-3 A callback that positions glyphs along a sine curve 121
 Listing 6-4 Installing a callback for a post-layout override operation 123
 Listing 6-5 Setting up the quadratic curve callbacks 125
 Listing 6-6 A function that draws glyph outlines using quadratic curve data 128

Appendix B **Font Features 135**

Figure B-1 Glyphs drawn with annotations 136
 Figure B-2 Cursive connection in a Roman font 137
 Figure B-3 The character “g” shown as two glyph forms 138
 Figure B-4 Traditional and simplified versions of a Chinese character 138
 Figure B-5 Four levels of design complexity 140
 Figure B-6 Hebrew text with diacritical marks shown (upper) and hidden (lower) 141
 Figure B-7 Accented forms 141
 Figure B-8 Fractions drawn with three different fractions feature selectors 142
 Figure B-9 Text drawn with different letter case feature selectors 143

Figure B-10	Levels of ligature formation controlled with ligature feature selectors	144
Figure B-11	Use of diphthong ligatures	145
Figure B-12	The word “hindi” drawn with rearrangement turned on and off	146
Figure B-13	Drawing text without and with mathematical extras	147
Figure B-14	Uppercase and lowercase numerals	148
Figure B-15	Fixed-width and proportional-width numerals	149
Figure B-16	Ornamental glyphs	150
Figure B-17	Allowing and preventing glyph overlap	151
Figure B-18	Ruby text above the base text	152
Figure B-19	Text drawn with different swash feature selectors	153
Figure B-20	Normal, monospaced, and proportional text spacing	155
Figure B-21	Hanja text (top) displayed as Hangul (bottom)	155
Figure B-22	Normal, superior, inferior, and ordinal vertical positions	158
Figure B-23	Vertical substitution forms in a font	159
Table B-1	Selectors for the all-typographic features feature type	135
Table B-2	Selectors for the annotation feature	136
Table B-3	Selectors for the cursive connection feature type	137
Table B-4	Selector for the character alternatives feature	138
Table B-5	Selectors for the character shape feature	138
Table B-6	Selectors for the CJK italic Roman feature type	139
Table B-7	Selectors for the CJK Roman spacing feature	140
Table B-8	Selectors for the design complexity feature	140
Table B-9	Selectors for the diacritical marks feature	141
Table B-10	Selectors for the fractions feature	142
Table B-11	Selectors for the ideographic spacing feature	142
Table B-12	Selectors for the kana spacing feature	143
Table B-13	Selectors for the letter case feature	144
Table B-14	Selectors for the ligatures feature type	145
Table B-15	Selectors for the linguistic rearrangement feature	147
Table B-16	Selectors for the mathematical extras feature	147
Table B-17	Selectors for the number case feature	149
Table B-18	Selectors for the number spacing feature	149
Table B-19	Selectors for the ornament sets feature	150
Table B-20	Selectors for the overlapping characters feature	151
Table B-21	Selectors for the ruby kana feature type	152
Table B-22	Selectors for the smart swash feature	153
Table B-23	Selectors for the style options feature	154
Table B-24	Selectors for the text spacing feature	155
Table B-25	Selectors for the transliteration feature	155
Table B-26	Selectors for the typographical extras feature	156
Table B-27	Selectors for the Unicode decomposition feature type	157
Table B-28	Selectors for the vertical position feature	158
Table B-29	Selectors for the vertical substitution feature	159

Introduction to ATSUI Programming Guide

Note: This document was previously titled *Rendering Unicode Text With ATSUI*.

Apple Type Services for Unicode Imaging (ATSUI) is the technology behind all text drawing in Mac OS X. This document gives an overview of ATSUI, provides an introduction to the concepts and terms you need to understand ATSUI, discusses the core data types you use to control text layout and styles, and shows you how to use ATSUI in your application.

You should read this document if you plan to write an application that

- allows fine control over layout features—for example, you can precisely position individual glyphs and lines of text as well as draw Unicode text at any angle of rotation, including vertically
- provides text-processing services when Multilingual Text Engine (MLTE) isn't sufficient
- accesses Quartz text effects from Carbon in Mac OS X
- supports high-end typography

You might also find parts of this document useful if you are using an API (such as MLTE) that calls into ATSUI, as this document discusses typographical concepts and terms that are relevant to the APIs that call ATSUI.

Organization of This Document

The chapters in the rest of this document cover the following topics:

- [“Typography Concepts”](#) (page 15) provides an overview of how text is laid out and displayed and defines the typographical concepts you need to know to understand how ATSUI implements typography.
- [“ATSUI Style and Text Layout Objects”](#) (page 37) describes the style and text layout objects used by ATSUI and outlines many of the style, line, and layout features you can control with ATSUI.
- [“Basic Tasks: Working With Objects and Drawing Text”](#) (page 69) discusses general guidelines for using ATSUI, provides step-by-step instructions for getting and setting line, layout, and style attributes, and shows you how to do the most common tasks with ATSUI, such as measuring and drawing text.
- [“Interactive Tasks: Supporting Carets and Highlighting Text”](#) (page 99) describes how to support user interaction with the text your application draws onscreen.
- [“Advanced Tasks: Substituting Fonts and Modifying Layouts”](#) (page 103) provides detailed instructions for doing such advanced tasks as using font fallbacks and kerning text.
- [“Direct-Access Tasks: Working With Glyph Data”](#) (page 113) shows how you can use direct-access functions to control the internal layout processes of ATSUI to affect how glyphs are drawn.
- [“ATSUI Implementation of the Unicode Specification”](#) (page 131) gives additional details on how ATSUI implements Unicode.

- “Font Features” (page 135) lists font feature types and the selectors available for each feature type.

Overview of ATSUI

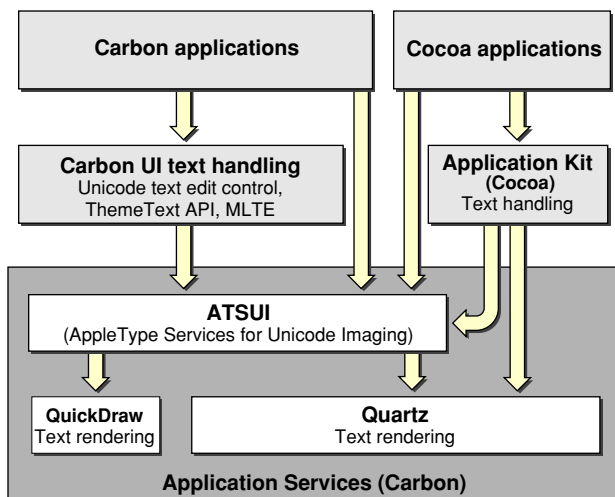
Mac OS X is an international operating system. It fully supports Unicode and comes with high-quality fonts capable of displaying many languages, including Japanese and Chinese. Mac OS X supports input of non-phonetic characters used by Chinese, Japanese, and Korean through the use of input methods. Because Mac OS X uses Unicode for all onscreen text display, ATSUI is at the heart of all text drawing in Mac OS X, as shown in Figure I-1. From the Mac OS X Human Interface Toolbox to the Menu Manager, all Mac OS X text drawing at some point uses ATSUI to render Unicode text. For example, the Finder uses ATSUI to display menu items and create text editing fields:

- Menu items are static text displayed by calling the Menu Manager function `DrawThemeTextBox`, which in turn uses ATSUI to render the Unicode text.
- Editable text fields are created using MLTE (Multilingual Text Engine) text objects, but MLTE relies on ATSUI to render the Unicode text in the editable text fields.

TextEdit is another application among the many Mac OS X applications that use ATSUI to lay out and render Unicode text. The Unicode text rendering of ATSUI includes support for accents and ligatures, bidirectional text, contextual forms and vowel reordering, vertical text, and surrogates.

ATSUI is not just a technology that works behind the scenes; you can call directly into the ATSUI API from your application to render Unicode-encoded text. With ATSUI, you can control many aspects of text layout, including justification, alignment (flushness), kerning, tracking, shifting, and ligature formation. Using ATSUI direct-access functions, you can, if necessary, control how individual segments of glyphs are drawn.

Figure I-1 ATSUI and text drawing in Mac OS X



ATSUI uses opaque objects to keep track of style and layout information that specify how you want text displayed. Using the information in these objects, ATSUI can support high-end typographic control of

- bidirectional text

INTRODUCTION

Introduction to ATSUI Programming Guide

- vertical text with variants, kanji variants, and proportional Chinese-Japanese-Korean (CJK)
- kashida justification, rearrangement, ligatures, and contextual forms
- combined characters that require ligation, morphing, and other advanced effects
- optical alignment
- stylistic sophistication of font design such as swash variants and arbitrary ligatures

ATSUI has a number of functions that support text editing, including highlighting, hit-testing, caret movement, and caret positioning functions. For developers who need to manipulate glyphs directly, ATSUI also provides a set of functions that allows access to glyph outlines and details of the glyph array.

ATSUI provides full Unicode 3.2 layout support and supports text rendering support for all the features required by scripts included with version 2.1 of the Unicode standard or later. The ability of ATSUI to render Unicode text is limited only by the available fonts the user has installed. (ATSUI does not provide other Unicode-related text processing services such as formatting the date and time.)

ATSUI and Font Data

ATSUI takes advantage of all data included in a font. This means if a font designer includes ligatures, swashes, alternate forms, or any data that describes the font, ATSUI can use the data. For example, the Zapfino font provides several forms for a ligature. Using ATSUI, you can access the Zapfino font data to automatically form the appropriate ligature. When ATSUI changes glyphs to provide an alternate form, ligature, or other modification, the text doesn't change. This ensures that such services as spelling checking and text searching work properly even when the text is redrawn. If you don't want to use the data provided with a font, you can set up ATSUI to override the data with the settings you provide.

ATSUI and Quartz

ATSUI renders text with Quartz, Apple's 2D rendering engine, giving your text the look and feel that makes Mac OS X unique. ATSUI is fully integrated with Quartz in Mac OS X. You can use the two technologies together to get superior anti-aliasing and to take advantage of the attributes associated with a Quartz context (`CGContext`). For example, some features, such as line rotation, are supported by both ATSUI and Quartz. By using ATSUI and Quartz together, you have the option to choose which technology does the rotation. You'll see how to do this later in ["Drawing Text Using a Quartz Context"](#) (page 77).

Quartz provides a number of other features that you may want to take advantage of, such as transparency. You can set a transparency level for a Quartz context, then when you use ATSUI to draw text into the context, the text is drawn with the transparency level specified for the Quartz context.

See Also

ATSUI Reference, a complete reference to the ATSUI programming interface

To download ATSUI-related sample code and to obtain documentation about other programming interfaces related to text and international services, see the Apple developer website:

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

INTRODUCTION

Introduction to ATSUI Programming Guide

For detailed information about Unicode:

<http://www.unicode.org>

Typography Concepts

This chapter introduces and defines important typographic terms related to **text**—traditionally defined as the written representation of spoken language. The terms and concepts discussed in this chapter are used throughout the rest of the book and in *ATSUI Reference*. If you plan to use ATSUI, or any API that uses ATSUI (for example, MLTE), you should read this chapter.

This chapter starts by outlining the different components that make up text. It then describes how text is

- measured and stored
- arranged on a display device
- adjusted in various ways to affect the text direction, kerning, alignment (or flushness), justification, and line breaks
- drawn, highlighted, and hit-tested

Characters, Glyphs, and Fonts

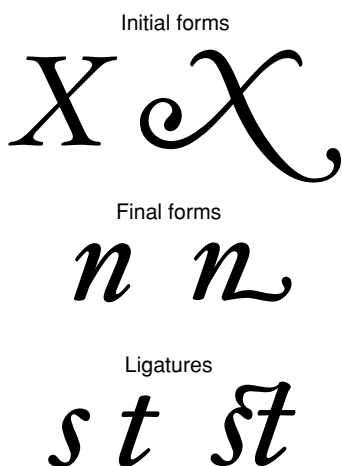
A writing system’s alphabet, numbers, punctuation, and other writing marks consist of characters. A **character** is a symbolic representation of an element of a writing system; it is the concept of, for example, “lowercase a” or “the number 3.” It is an abstract object, defined by custom in its own language.

As soon as you write a character, however, it is no longer abstract but concrete. The exact shape by which a character is represented is called a **glyph**. The “characters” that ATSUI places on the screen are really glyphs.

Glyphs and characters do not necessarily have a one-to-one correspondence. For example, a single character may be represented by one or more glyphs (the character “lowercase i” could be represented by the combination of a “dotless-i” glyph and a “dot” glyph), and a single glyph can represent two or more characters (the two characters “f” and “i” can be represented by a single glyph as shown in [Figure B-10](#) (page 144)).

Context—where the glyph appears in a line of text—also affects which glyph represents a character. Figure 1-1 shows examples of such contextual forms in a Roman font. Different forms of a glyph are used according to whether the glyph stands alone, occurs at the beginning of a word, occurs at the end of a word, or forms part of a new glyph, as in a ligature. For example, both forms of “X” can be used at the beginning of a word, but the form on the right side of the figure can be used only at the start of a word, whereas the form on the left side can be used alone or at the start of a word. Similarly, the “n” on the left side of the figure can be used alone or at the end of a word whereas the “n” on the right side can be used only to end a word. The last line in Figure 1-1 illustrates how “s” and “t” can be drawn separately or combined to form a ligature.

Figure 1-1 Contextual forms in a Roman font



A **font** is a collection of glyphs, all of similar design, that constitute one way to represent the characters of one or more languages. Fonts usually have some element of design consistency, such as the shape of the ovals (known as the **counter**), the design of the stem, stroke thickness, or the use of **serifs**, which are the fine lines stemming from the upper and lower ends of the main strokes of a letter. Figure 1-2 shows some of the elements of glyphs that indicate they are members of the same family.

Figure 1-2 Elements that distinguish glyphs of a Roman font



A **font family** is a group of fonts that share certain characteristics and have a common family name. Each font family has its own name, such as “New York,” “Geneva,” or “Symbol.” In contrast, a font always has a full name—for example, Geneva Regular or Times Bold. The full name determines which family the font belongs to and what typestyle it represents. (See “Typestyles” (page 21) for more information.) Several fonts may have the same family names (such as Geneva, Geneva Bold, and so on) but are stored separately—these fonts are still part of the same font family. The font Geneva Italic, for example, shares many characteristics with Geneva Regular, but all of the glyphs slant at a certain angle. Though different, these fonts are part of the same font family.

Character Encoding and Glyph Codes

A computer represents characters in numeric form. A **character encoding** is a mapping that assigns numeric values (character codes) to characters. The mapping varies depending on the character set, which in turn may depend on the language as well as other factors.

Fonts associate each glyph with a double-byte code called its **glyph code** (which is not the same as a character code). Different fonts may have different glyph codes for the same glyphs, and a single font may have several glyph codes associated with a particular character because several glyphs may represent that character. Because the font and general textual context determine which glyph and which glyph codes represent characters, ATSUI transparently handles the details of mapping character codes to the correct glyph codes. Your application does not have to handle the details of obtaining glyphs from a font.

Note: Your application usually deals with character codes, since those correspond most closely with what the user types. However, ATSUI permits you to deal with glyph codes, if that is more appropriate to your application's functionality and needs.

Different languages may have different requirements in terms of which glyphs they use from a font. A font contains some number of character encodings. Each encoding is an internal conversion table for interpreting a specific character set—that is, a way to map a character code to a glyph code for that font.

The reason a font can have multiple encodings is that the requirements for each writing system that the font supports may be different. A **writing system** is a method of depicting words visually. It consists of a character set and a set of rules for displaying, ordering, and formatting the glyphs associated with those characters. Writing systems can differ in line direction, the direction in which their glyphs are read, the size of the character set used to represent the script, and contextual variation (that is, whether a glyph changes according to its position relative to other glyphs). Writing systems have specific requirements for text display, text editing, character set, and fonts. A writing system can serve one or several languages. For example, the Roman writing system serves many languages, including French, Italian, and Spanish.

Unicode

Most character sets and character encoding schemes developed in the past are limited in that they supported just one language or a small set of languages. Multilingual software has traditionally had to implement methods for supporting and identifying multiple character encodings. To interpret a character encoded numerically, you needed to know the text encoding system used to encode the character. Because text encoding systems are not unique, the same numeric encoding used in different systems may not represent the same character. The adoption of Unicode has changed this.

Unicode is a character encoding system designed to support the interchange, processing, and display of all the written texts of the diverse languages of the modern world. Unicode supplies enough numeric values to encode all characters available to all written texts. It provides a single model for text display and editing. Unicode also simplifies the handling of bidirectional text and characters that change according to their position in the sentence.

Unicode has three primary formats available for encoding: UTF-8, UTF-16, and UTF-32 (UTF stand for Unicode Transformation Format). UTF-8 is a single-byte format; UTF-16 is a double-byte format; and UTF-32 is a quadruple-byte format. ATSUI uses UTF-16, which uses two bytes to specify a character. Text that uses an encoding other than Unicode can be converted to Unicode using the Text Encoding Converter. See *Programming With the Text Encoding Conversion Manager* for more information.

Because Unicode includes the character repertoires of most common character encodings, it facilitates data interchange with other platforms. Using Unicode, text manipulated by your application and shared across applications and platforms can be encoded in a single coded character set.

Unicode provides some special features, such as combining or nonspacing marks and conjoining Jamo. These features are a function of the variety of languages that Unicode handles. If you have coded applications that handle text for the languages these features support, they should be familiar to you. If you have used a single coded character set such as ASCII almost exclusively, these features will be new to you. ATSUI lets you control how the special features available through Unicode are rendered.

For more information on Unicode, see <http://www.unicode.org>. For additional details on how ATSUI implements the Unicode specification, see “ATSUI Implementation of the Unicode Specification” (page 131).

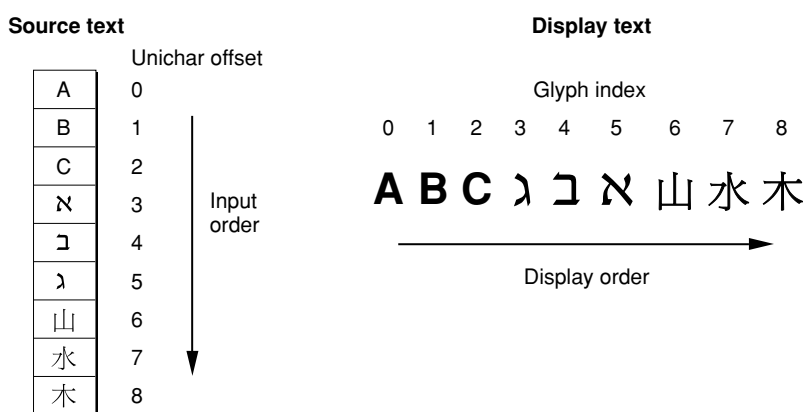
Text Storage

ATSUI doesn’t allocate and manage buffers for your text; you are responsible for memory management. Instead ATSUI caches the text that you want rendered, storing the cached text as a sequence of glyph codes. Because ATSUI uses Unicode (UTF-16), each character uses at least 2 bytes of storage. (Surrogate and other characters may require more than 2 bytes of storage.) The storage order is the order in which text is stored. The text stored in memory is the **source text**; the text displayed is the **display text**, as shown in [Figure 1-3](#) (page 18).

Display order is the left-to-right (or top-to-bottom) order in which glyphs are drawn. ATSUI expects you to store your text in **input order**, which is the “logical” order, or the order in which the characters, not glyphs, would be read or pronounced in the language of the text. Because text of different languages may be read from left to right, right to left, or top to bottom, the input order is not necessarily the same as the display order of the text when it is drawn. Your application needs to differentiate between the order in which the character codes are stored and the order in which the corresponding glyphs are displayed. Figure 1-3 shows Hebrew glyphs that are stored one way and displayed another way.

As shown in Figure 1-3, the character codes that make up the text are numbered using zero-based **offsets**. Therefore, the first character code in the figure has an offset of 0.

Figure 1-3 Input order and display order



ATSUI also uses a zero-based numbering scheme to index the glyphs that are actually displayed. The **glyph index**, which gives the glyph’s position in the display order, always starts at 0. Each glyph has a single index, even though its character code is 2 bytes (one `Unichar`) long.

Note: Throughout this document, text in computer memory is drawn as a vertical table representing sequential (downward) storage of text characters in a buffer.

Text Measurements

Most users use point size to specify the size of the glyphs in a document. **Point size** indicates 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**, and there are 72.27 points per inch. However, ATSUI and the PostScript language both define 1 point to be exactly 1/72 of an inch. Although point size is a useful measure of the size of text, you may wish to use more exact measurements for greater control over placement of the glyphs on the display device. ATSUI permits fractional point sizes.

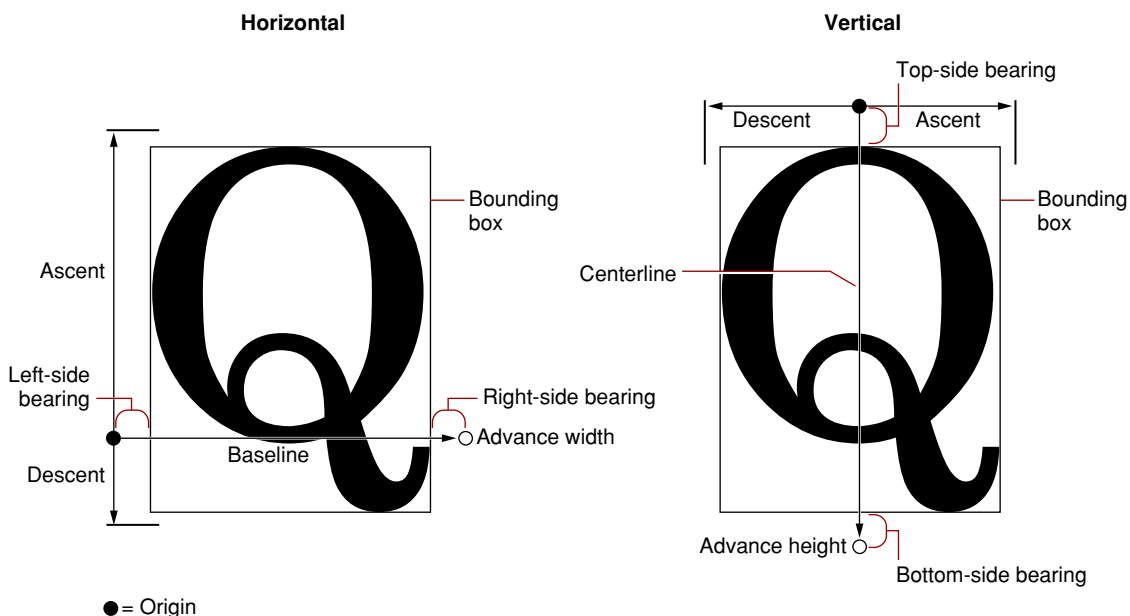
Font designers use a special vocabulary for the measurements of different parts of a glyph. [Figure 1-4](#) (page 20) shows the terms describing the most frequently used measurements. The **bounding box** of a glyph is the smallest rectangle that entirely encloses the drawn parts of the glyph. The **glyph origin** is the point that ATSUI uses to position the glyph when drawing. In [Figure 1-4](#), notice that there is some space between the glyph origin and the edge of the bounding box: this space is the glyph's **left-side bearing**. The left-side bearing value can be negative, which decreases the space between adjacent characters. The **right-side bearing** is space on the right side of the glyph; this value may or may not be equal to the value of the left-side bearing. The **advance width** is the full horizontal width of the glyph as measured from its origin to the origin of the next glyph on the line, including the side bearings on both sides.

Most glyphs in Roman fonts appear to sit astride the **baseline**, an imaginary horizontal line. The **ascent** is a distance above the baseline, chosen by the font's designer and the same for all glyphs in a font, that often corresponds approximately to the tops of the uppercase letters in a Roman font. Uppercase letters are chosen because, among the regularly used glyphs in a font, they are generally the tallest.

Note: Don't confuse the ascent, which is the distance from the baseline to the top of the uppercase letters in a font, with the ascender, shown in [Figure 1-2](#) (page 16). An ascender is that portion of a glyph that extends above the height of lowercase letters that do not have ascenders. (See x-height in [Figure 1-2](#).)

The **descent** is a distance below the baseline that usually corresponds to the bottoms of the descenders (the "tails" on glyphs such as "p" or "g"). The descent is the same distance from the baseline for all glyphs in the font, whether or not they have descenders. The sum of ascent plus descent marks the **line height** of a font.

Figure 1-4 Terms for glyph measurements

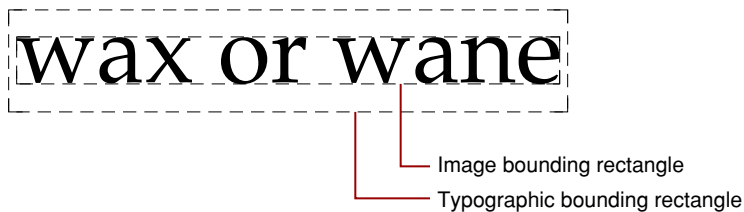


For vertical text, font designers may use additional measurements. The **top-side bearing** is the space between the top of the glyph and the top edge of the bounding box. The **bottom-side bearing** is the distance from the bottom of the bounding box to the origin of the next glyph. For vertical text, the **advance height** is the sum of the top-side bearing, the bounding-box height, and the bottom-side bearing. These metrics are useful if, for example, you want to display a horizontal font vertically. Likewise, vertical fonts such as kanji may also have horizontal metrics.

Every block of text has an **image bounding rectangle**, which is the smallest rectangle that completely encloses the filled or framed parts of a block of text. However, because of the height differences between glyphs—for example, between a small glyph, such as a lowercase “e”, and a taller, larger glyph, such as an uppercase “M”, or even between glyphs of different fonts and point sizes—the image bounding rectangle may not be sufficient for your application’s purposes. Therefore, you can also use the **typographic bounding rectangle**, which, in most cases, is the smallest rectangle that encloses the full span of the glyphs from the ascent line to the descent line.

Note: In some cases (such as with the Zapfino font), the font designer allows glyphs to extend beyond the typographic bounding rectangle, even if this may cause collisions with glyphs from another line of text. The ATSUI function `ATSUMeasureTextImage` obtains the actual image rectangle, even when glyphs extend beyond the typographic bounding rectangle.

Figure 1-5 shows an example of how the typographic bounding rectangle and image bounding rectangle relate. The two rectangles are markedly different because the text has no ascenders or descenders. Whereas the image bounding rectangle encloses just the black bits of the drawn text, the typographic bounding rectangle takes into account the ascent and descent needed to accommodate all glyphs in a font. If text includes glyphs with ascenders and descenders, the typographic bounding rectangle doesn’t change, but the image bounding rectangle does.

Figure 1-5 Image bounding rectangle and typographic bounding rectangle

Typestyles

Glyphs can be differentiated not only by font but by typestyle. A **typestyle** is a specific variation in the appearance of a glyph that can be applied consistently to all the glyphs in a font family. Some of the typical typestyles available on the Macintosh computer include plain, bold, italic, underline, outline, shadow, condensed, and extended. Other styles that may be available are demibold, extra-condensed, or antique.

Font Variations

A **font variation** is a setting along a particular variation axis. Font variations allow your application to produce a range of typestyles algorithmically. Each **variation axis** has a name that usually indicates the typestyle that the axis represents, a tag to represent that name (such as 'wght'), a set of maximum and minimum values for the axis, and the default value of the axis. The weight axis, for example, governs the possible values for the weight of the font; the minimum value may produce the lightest appearance of that font, the maximum value the boldest. The default value is the position along the variation axis at which that font falls normally. Because the axis is created by the font designer, font variations can be optimized for their particular font. Figure 1-6 shows a range of possible weights for a glyph, from the minimum weight to the maximum weight.

Figure 1-6 Font variations along a variation axis

Font Instances

A **font instance** is a set of named variations identified by the font designer that matches specific values along one or more variation axes and associates those values with a name. For example, suppose a font has the variation axis 'wght' with a minimum value of 0.0, a default of 0.5, and a maximum value of 1.0. The corresponding font instance might have the name "Demibold" with a value along that variation axis of 0.8.

In Figure 1-6, the variation axis value of the glyph at the far right could represent the named instance "Extra Bold," whereas the glyph at the far left could represent the named instance "Light." The other values represented in the figure could likewise have instance names.

Font variations and font instances give your application the ability to provide whatever timesteps the font designer has decided to include with the font. They are available through ATSUI only if the font designer has defined variations and instances for a font.

Text Layout

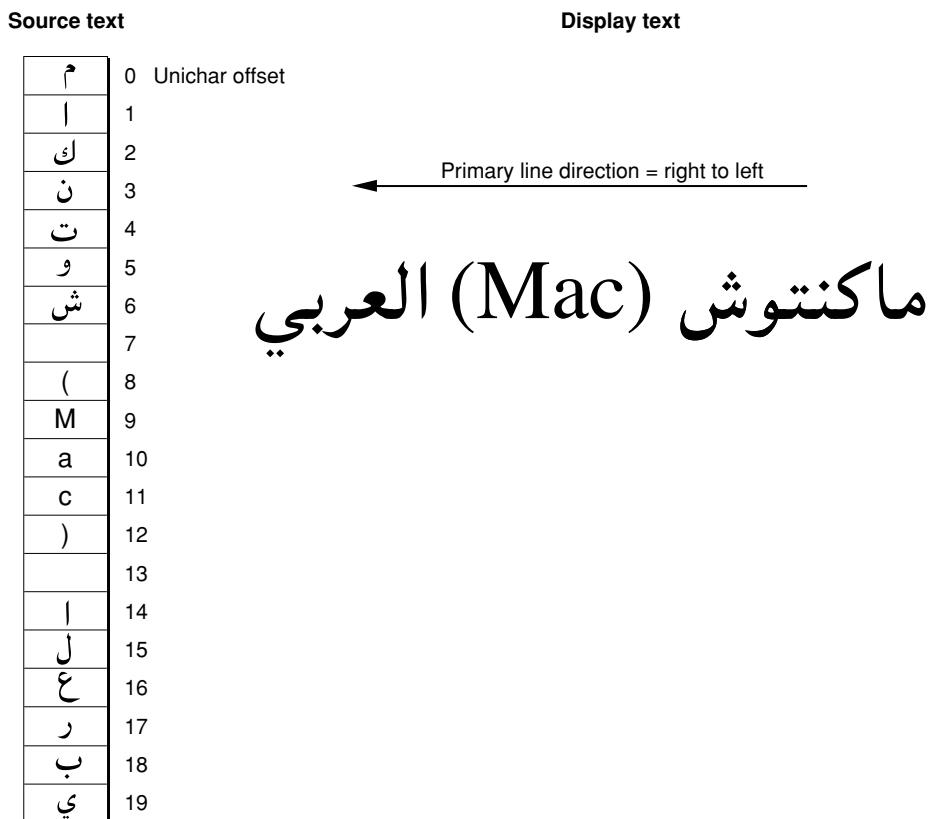
When you arrange text on a display device, you can let ATSUI use default values to automatically handle many aspects of the text display or you can specify precisely how ATSUI should present and arrange the text. The specifics of text layout using ATSUI are discussed in later chapters. This section describes some of the general concepts of laying out text on a display device. For example, when laying out text, the following need to be considered:

- text direction
- baselines
- text runs, style runs, and direction runs
- contextual forms and ligatures
- alignment (or flushness) and justification
- kerning and tracking
- line breaks
- font substitution

Text Direction

Text direction consists of text orientation (horizontal or vertical) and the direction in which the text is read. Text in your application can be oriented in three common directions: horizontally, left to right; horizontally, right to left; and vertically, top to bottom. ATSUI allows your application, for example, to draw lines of text in multiple directions, as shown in Figure 1-7.

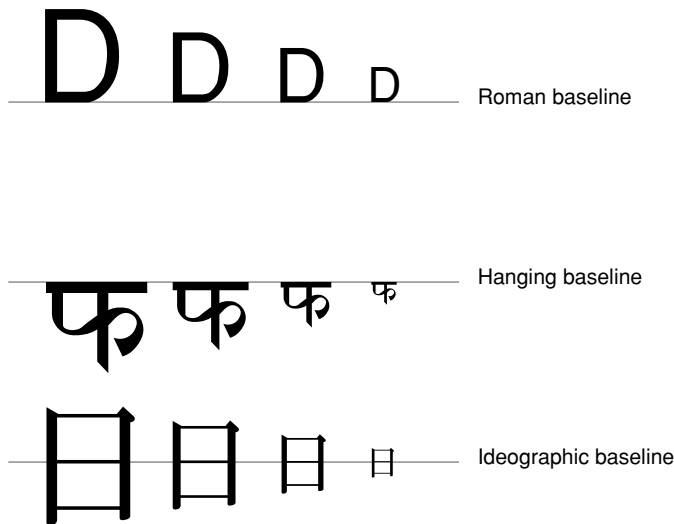
Figure 1-7 Bidirectional text



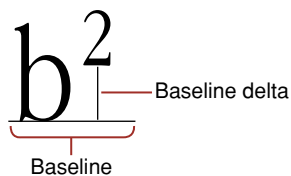
Baselines

A **baseline** is an imaginary line that coincides with some point in a font—for example, the bottom, middle, or top of each glyph. The baseline of a glyph defines the position of the glyph with respect to other glyphs at different point sizes when all the glyphs are aligned. It represents a stable platform from which glyphs of different sizes and different writing systems grow proportionally.

Depending on the writing system, the baseline may be above, below, or through the center of each glyph, as shown in Figure 1-8. ATSUI provides your application with capabilities for using multiple baselines. For more information, see “[Baseline Offsets Attribute Tag](#)” (page 59).

Figure 1-8 Baselines for different sizes of a glyph and for different writing systems

A **baseline delta** is the distance between the baseline and the position of the glyph with respect to the baseline, as shown in Figure 1-9.

Figure 1-9 The baseline delta

Various baselines can also be used to create special effects, such as drop capitals. A **drop capital** is an initial capital letter that is much larger than surrounding glyphs and embedded in them. Figure 1-10 is an example of drop capitals formed solely on the basis of the baselines in the font. The default baseline for this text is the Roman baseline for 18-point type. The hanging baseline of the drop capitals aligns with the hanging baseline of the regular text, creating the effect shown.

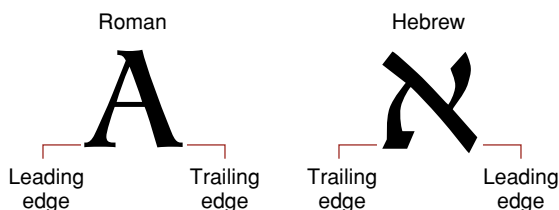
Figure 1-10 Drop capitals

Leading Edges and Trailing Edges

Because text has a direction, the concept of which glyph comes “first” in a line of text cannot always be limited to the visual terms “left” and “right.” The **leading edge** is defined as the edge of a glyph you first encounter—such as the left foot of a Roman glyph—when you first read the text that includes that glyph. The **trailing edge** is the edge of a glyph encountered last.

Figure 1-11 shows how the concepts of leading edge and trailing edge change depending on the characteristics of the glyph. In the first example—a Roman glyph—the leading edge is on the left, because the reader encounters that side first. In the second example, the leading edge of the Hebrew glyph is on the right for the same reason.

Figure 1-11 Leading edges and trailing edges



Text Runs, Style Runs, and Direction Runs

In any segment of contiguous text, certain parts stand out as belonging together, because the glyphs share a certain font, typestyle, or direction. For the purposes of referring to individual segments of text, you can think of sequences of glyphs that are contiguous in memory and share a set of common attributes as **runs**.

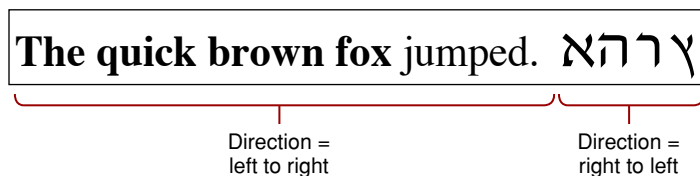
ATSUI associates a block of text with a **text run**. A sequence of glyphs contiguous in memory that share the same style is a **style run**. As Figure 1-12 shows, a text run can be subdivided into several style runs. The figure shows three style runs.

Figure 1-12 Three style runs in a line of text



A sequence of contiguous glyphs that share the same text direction is a **direction run**. As with text runs and style runs, the number of direction runs does not necessarily correlate to the number of style runs available, as shown in Figure 1-13, which has two direction runs.

Figure 1-13 Two direction runs in a line of text



Contextual Forms and Ligatures

A glyph's position next to other glyphs or its position in a word or a line of text may determine its appearance. For some writing systems, such as Roman, alternate glyphs are used for aesthetic reasons; in other writing systems, use of alternate forms is required.

A **contextual form** is an alternate form of a glyph that is chosen depending on the glyph's placement in a certain context, such as a certain word or line. Some contextual forms of initial and final forms of glyphs in a Roman font are shown in [Figure 1-1](#) (page 16). Other writing systems, such as Arabic, require different contextual forms of glyphs according to where they appear. [Figure 1-14](#) shows the forms of the Arabic letter “ha” that appear alone and at the beginning, middle, or end of a word. The same character code is used for each case; ATSUI finds the appropriate glyph code.

Figure 1-14 Contextual forms of the Arabic letter “ha”

Independent	Final	Medial	Initial
ه	ـه	هـ	هـ

Ligatures are two or more glyphs combined to form a single new glyph (whereas contextual forms are variations on the shape of one glyph). In the Roman writing system, ligatures are generally an optional aesthetic refinement. In other writing systems, special ligatures are required when certain glyphs appear next to one another. Some examples of ligatures used in a Roman font are shown in [Figure 1-15](#).

Figure 1-15 Examples of Roman ligatures

f + l = fl

f + f = ff

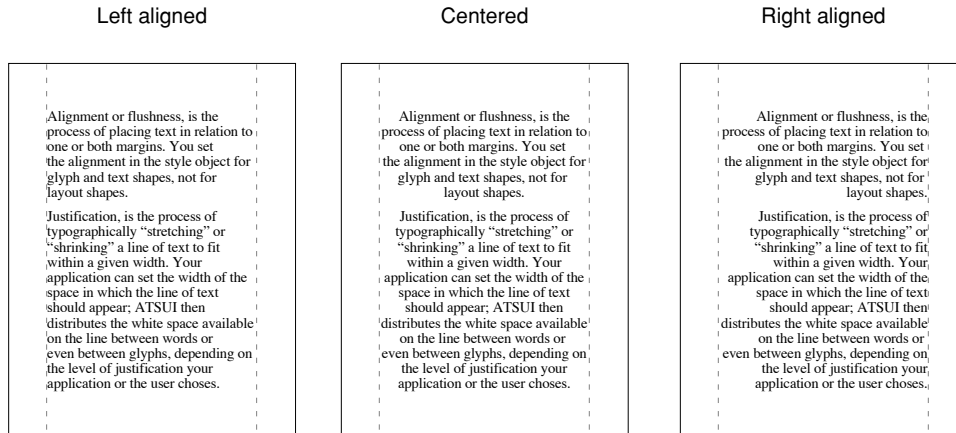
In general, the font contains all of the information needed to determine when your application should use the appropriate contextual forms and ligatures. If your application allows alternate forms of glyphs to be used, ATSUI does the substitution for you.

Alignment and Justification

The **text area** is the space on the display device in which text you want to display should fit. The left, right, top, and bottom sides of that area are the **margins**. How you arrange the text depends on the effect you want to achieve. There are two primary methods of arranging text: alignment and justification. **Alignment** (also called flushness) is the process of placing text in relation to one or both margins. [Figure 1-16](#) shows left, right, and center alignment of text. **Justification** is the process of typographically stretching or shrinking a line of text to fit within a given width. Your application can set the width of the space in which the line of text should appear. ATSUI then distributes the white space available on the line between words or even between glyphs, depending on the level of justification chosen.

There are other means of aligning or justifying a line—for example, stretching a glyph or decomposing a ligature. ATSUI can also handle complex justification such as that used in Arabic writing systems.

Figure 1-16 Left, center, and right text alignment (or flushness)



Kerning and Tracking

Kerning is an adjustment to the normal spacing between two or more specific glyphs. A **kerning pair** consists of two adjacent glyphs such that the position of the second glyph is changed with respect to the first. The font designer determines which glyphs participate in kerning and in what context. Any adjustments to glyph positions are specified relative to the point size of the glyphs. Kerning usually improves the apparent letter-spacing between glyphs that fit together naturally.

Figure 1-17 shows how glyphs are positioned differently with and without kerning. Note that the phrase is shorter when kerning is applied than when it is not applied.

Figure 1-17 Glyphs with and without kerning



Cross-stream kerning allows the automatic movement of glyphs perpendicular to the line orientation of the text. For example, when ATSUI applies cross-stream kerning to horizontal text, the automatic movement is vertical; this feature is required for writing systems such as Taliq, which is used in the Urdu language.

When your application lays out text, it has the option of using the interglyph spacing specified by the font designer or altering the spacing slightly in order to achieve a tighter fit between letters and improve the look of a line of text.

Your application can also use tracking if it has been specified by the font designer. In **tracking**, space is adjusted between all glyphs in the run. You can increase or decrease interglyph spacing by using a **tracking setting**, which is a value that specifies the relative tightness or looseness of interglyph spacing. Positive tracking settings result in an increase in the looseness of all glyphs in the run. Negative tracking settings result in an increase in the tightness of all glyphs. Normal tracking, tight tracking, and loose tracking are shown in Figure 1-18.

Figure 1-18 Tracking settings for normal, tight, and loose tracking

A B C a b c Normal tracking
(tracking = 0)

A B C a b c Tight tracking
(tracking < 0)

A B C a b c Loose tracking
(tracking > 0)

Special Font Features

Some special features are available in certain fonts. You can increase the control a user has over the presentation of text in a document if you provide access to these features when they are available in a font. The font provides the functionality for using these features. Table 1-1 shows some of the currently defined features. See “[Font Features](#)” (page 135) for a more complete list.

Table 1-1 Special font features

Feature	Description
Ligatures	Permits selection from different ranges of ligatures.
Cursive connections	Controls the level of cursive connection in the font. This feature is used in fonts, such as cursive Roman fonts or Arabic fonts, in which glyphs are connected to each other. See Figure B-2 (page 137) for an example.
Vertical substitution	Specifies that glyphs need to change their appearance in vertical runs of text. Figure B-23 (page 159) shows how a vertical form of a glyph can be substituted for a horizontal form.
Smart swashes	A swash is a variation, often ornamental, of an existing glyph. The smart swash feature controls contextual swash substitution, such as substituting a final glyph when a particular glyph appears at the end of a word. See Figure B-19 (page 153) for an example of swashes.
Vertical position	Controls superscripts, subscripts, and ordinal forms. See Figure B-22 (page 158) for an example of vertical positions.
Fractions	Governs selection and generation of fractions. See Figure B-8 (page 142) for examples of how fractions can be drawn.
Overlapping glyphs	Prevents the collision of long tails on glyphs with the descenders of other glyphs. See Figure B-17 (page 151) for an example.

Feature	Description
Typographical extras	Allows fine typographic effects, such as the automatic conversion of two adjacent hyphens to an em dash.
Ornament sets	Governs nonletter ornament sets of glyphs. See Figure B-16 (page 150) for an example of ornamental glyphs.
Style options	Allows the font designer to group together collections of noncontextual substitutions into named sets.
Character shape	Specifies the use, with Chinese fonts, of the traditional or simplified character forms. See Figure B-4 (page 138) for an example.

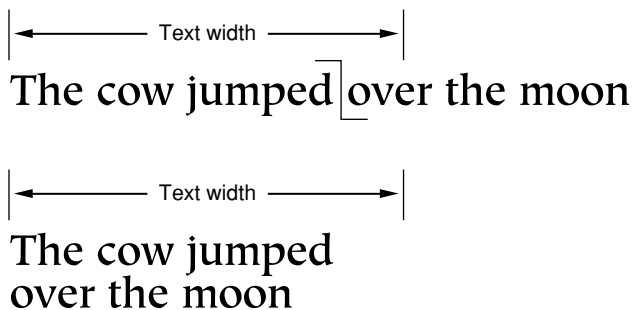
Some of these features, such as typographic extras, are fancy elements that provide the user with alternate forms of glyphs or other ornamental designs. Other features are contextual and absolutely necessary for using that font correctly—for example, cursive connectors for Arabic. Fonts that require these features include them, whereas optional elements are included at the discretion of the font designer.

Line Breaking

Your application determines how wide the text area for a line should be. At times, a user enters a line of text that does not fit neatly in the given text area and overlaps one of the margins. When this happens, you break the line of text and wrap the text onto the next line.

Figure 1-19 shows a line break made on the basis of a simple algorithm: The application backs up in the source text to the trailing edge of the last white space and then carries over all of the text following that white space to the next line.

Figure 1-19 Text width and line breaking



Your application can devise more complex algorithms, such as breaking a word at an appropriate hyphenation point, if possible. ATSUI leaves the final decision about where to break the line up to your application. However, ATSUI provides you with functions designed to help you determine the best place to break a line. For more information, see [“Breaking Lines”](#) (page 83) and [“Flowing Text Around a Graphic”](#) (page 88).

Font Substitution

In some cases there may not be a glyph defined for a character code in a font. Your application can specify a search order for ATSUI to use when trying to locate a replacement glyph. If ATSUI cannot find any suitable replacement, it recommends substituting a glyph from the Last Resort font. The **Last Resort** font is a collection of glyphs that represent types of Unicode characters. If the font cannot represent a particular Unicode character, the appropriate "missing" glyph from the Last Resort font is used instead. For more information on font substitution, see [“Using Font Fallback Objects”](#) (page 103).

Caret Handling

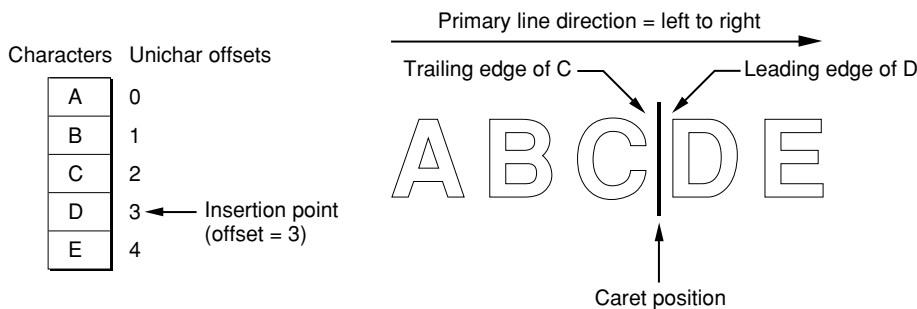
When the user places a pointer over a line of text and presses and releases a mouse button, the user expects that mouse click to produce a caret at the corresponding onscreen position. Similarly, when the user drags the mouse while pressing and holding the mouse button, the user expects the selected text to be highlighted. In each case, your application must recognize and respond to mouse events appropriately. This section discusses caret positioning, highlighting, and conversion of screen position to text offset in memory.

Caret Positioning

A caret position is a location on the screen that corresponds to an insertion point in memory. It lets the user know where in the text file the next insertion (or deletion) will occur. A caret position is always between glyphs on the screen, usually on the leading edge of one glyph and the trailing edge of another. The leading edge of a glyph is the edge that is encountered first when reading text of that glyph’s script system; the trailing edge is opposite from the leading edge. In left-to-right text, a glyph’s leading edge is its left edge; in right-to-left text, a glyph’s leading edge is its right edge.

In most situations for most text applications, the caret position is on the leading edge of the glyph corresponding to the character at the insertion point in memory, as shown in Figure 1-20. When a new character is inserted, it displaces the character at the insertion point, shifting it and all subsequent characters in the buffer forward by one character position.

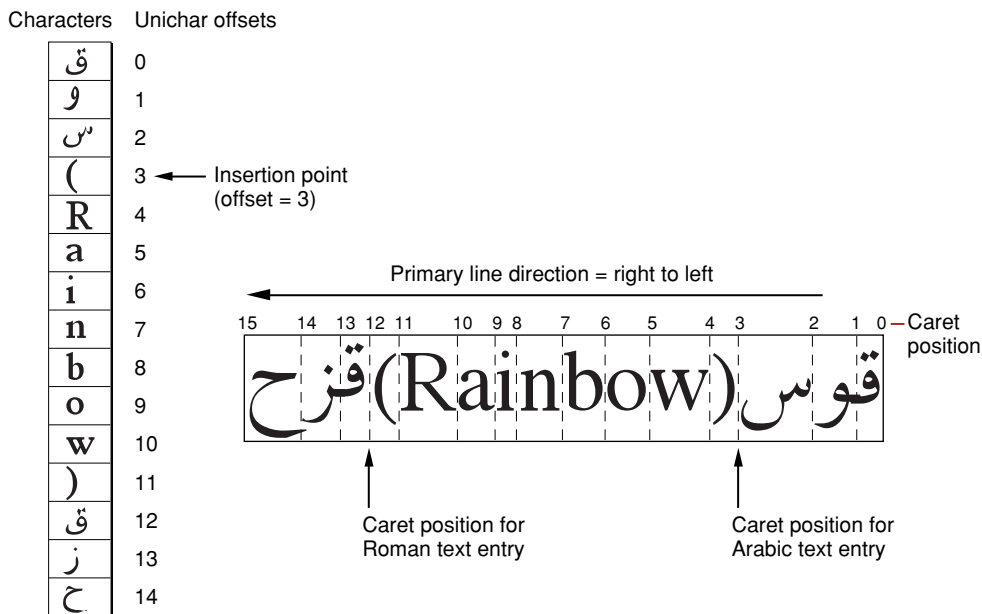
Figure 1-20 Caret position and insertion point



The caret position is unambiguous in text with a single line direction. In such a case, the caret position is on the trailing and leading edges of characters that are contiguous in the text buffer; it thus corresponds directly to a single offset in the buffer. This is not always the case in bidirectional text.

In determining caret position, an ambiguous case occurs at direction boundaries because the offset in memory can map to two different glyph positions on the screen—one for the text in each line direction. In Figure 1-21, for example, the insertion point is at offset 3 in the buffer. If the next character to be inserted is Arabic, the caret should be drawn at position 3 on the screen; if the next character is English, the caret should be drawn at caret position 12.

Figure 1-21 Caret positions at direction boundaries



The Mac OS codifies this relationship between text offset and caret position as follows:

- For any given offset in memory, there are two potential caret positions:
 - the leading edge of the glyph corresponding to the character at that offset
 - the trailing edge of the glyph corresponding to the previous (in memory) character
- In unidirectional text, the two caret positions coincide: the leading edge of the glyph for one character is at the same location as the trailing edge of the glyph for the previous character. In Figure 1-20 (page 30), the offset of 3 yields caret positions on the leading edge of “D” and the trailing edge of “C”, which are the same unambiguous location.
- At a boundary between text of opposite directions, the two caret positions do not coincide. Thus, in Figure 1-21, for an offset of 3 there are two caret positions: 12 and 3. Likewise, an offset of 12 yields two caret positions (also 12 and 3, but on the edges of two different glyphs).

At an ambiguous character offset, the current line direction (the presumed direction of the next character to be inserted) determines which caret position is the correct one:

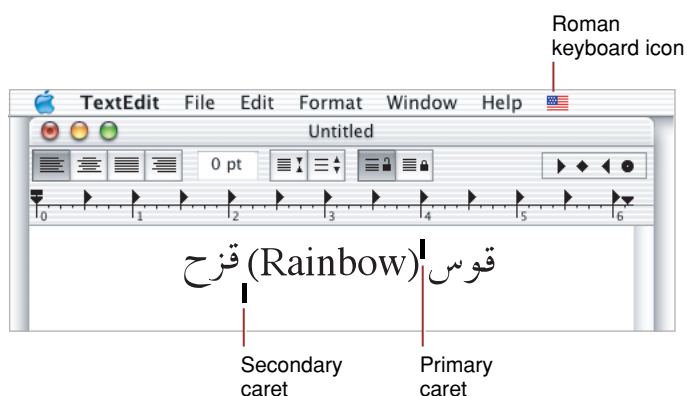
- If the current direction equals the direction of the character at that offset, the caret position is the leading edge of that character’s glyph. In Figure 1-21, if Roman text is to be inserted at offset 3 (occupied by a Roman character), the caret position is on the leading edge of that character’s glyph—that is, at caret position 12.

- If the current direction equals the direction of the previous (in memory) character, the screen position is on the trailing edge of the glyph corresponding to that previous (in memory) character. In Figure 1-21, if Arabic text is to be inserted at offset 3, the caret position is on the trailing edge of the glyph of the character at offset 2—that is, at caret position 3.

Two common approaches for drawing the caret at direction boundaries involve the use of a dual caret and a single caret. A **dual caret** consists of two lines, a high caret and a low caret, each measuring half the text height (see Figure 1-22). The high caret is displayed at the primary caret position for the insertion point; the low caret is displayed at the secondary caret position for that insertion point. Which position is primary, and which is secondary, depends on the primary line direction:

- The primary caret position is the screen location associated with the glyph that has the same direction as the primary line direction. If the current line direction corresponds to the primary line direction, inserted text will appear at the primary caret position. A primary caret is a caret drawn at the primary caret position.
- The secondary caret position is the screen location associated with the glyph that has a different direction from the primary line direction. If the current line direction is opposite to the primary line direction, inserted text will appear at the secondary caret position. In Figure 1-22, the display of the Roman keyboard icon shows that the current line direction is not the same as the primary line direction, so the next character inserted will appear at the secondary caret position. A secondary caret is a caret drawn at the secondary caret position.

Figure 1-22 Dual caret at direction boundaries in mixed-directional text



A single caret (or moving caret) is simpler than a dual caret (see Figure 1-23). It is a single, full-length caret that appears at the screen location where the next glyph will appear. At direction boundaries, its position depends on the keyboard script. At a direction boundary, the caret appears at the primary caret position if the current line direction corresponds to the primary line direction; it appears at the secondary caret position if the current line direction is opposite to the primary line direction. The moving caret is also called a jumping caret because its position jumps between the primary and secondary caret positions as the user switches the keyboard script between the two text directions represented.

Figure 1-23 Single carets at direction boundaries in mixed-directional text

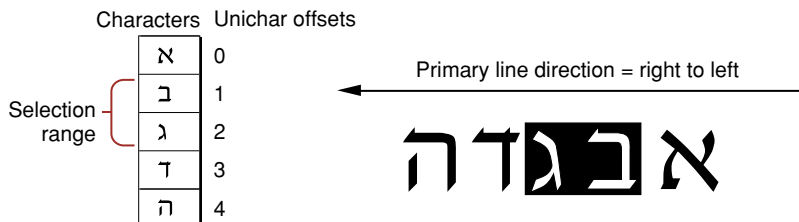


When you place a caret between glyphs, you need to be able to relate the caret to the insertion point: a point between `Unichar` offsets in the source text. The **edge offset** is a `Unichar` offset between character codes in the source text that corresponds to the caret location between glyphs. In Figure 1-20 (page 30), the edge offset is 3.

Highlighting

When displaying a selection range, an application typically marks it by **highlighting**, drawing the glyphs in inverse video or with a colored or outlined background. As part of its text-display tasks, your application is responsible for knowing what the selection range is and highlighting it properly, as well as for making the necessary changes in memory that result from any cutting, pasting, or editing operations involving the selection range. Figure 1-24 shows highlighting for a selection range in unidirectional text.

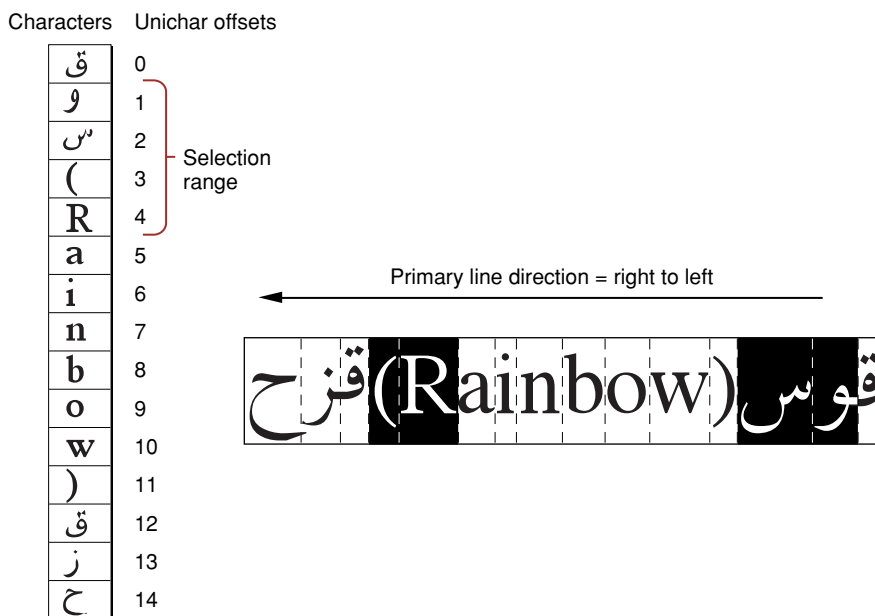
Figure 1-24 Highlighting a selection range in unidirectional text



The Mac OS measures the limits of highlighting rectangles in terms of caret position. Thus, in Figure 1-24, in which the selection range consists of the characters at offsets 1 and 2 in memory, the ends of the highlighting rectangle correspond to caret positions for offsets 1 and 3. It's equivalent to saying that the highlighting extends from the leading edge of the glyph for the character at offset 1 to the leading edge of the glyph for the character at offset 3.

If the displayed text has bidirectional runs, the selection range may appear as discontinuous highlighted text. This is because the characters that make up the selection range are always contiguous in memory, but characters that are contiguous in memory may not be contiguous onscreen. Figure 1-25 is an example of text whose selection range consists of a contiguous sequence of characters in memory, whereas the highlighted glyphs are displayed discontinuously.

Figure 1-25 Highlighting a selection range in bidirectional text



In describing the boundaries of the highlighting rectangles in terms of caret position, note that for Figure 1-25, it is not possible to simply say that the highlighting extends from the caret position of offset 2 to the caret position of offset 6. Using the definitions of caret position given earlier, however, it is possible to define the selection range as two separate rectangles, one extending from offset 4 to offset 2, and another extending from offset 12 to offset 6 (assuming for the ambiguous offsets—4 and 12—that the current text direction equals the primary line direction).

Converting Screen Position to Text Offset

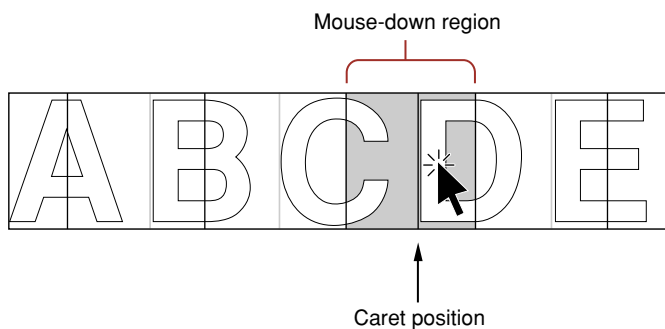
Caret positioning and highlighting, as just discussed, require conversion from text offset to screen position. But that is only half the picture; it is just as necessary to be able to convert from screen position to text offset. For example, if the user clicks the cursor within your displayed text, you need to be able to determine the offset in your text buffer equivalent to that mouse-down event. **Hit-testing** is the process of obtaining the location of an event relative to the position of onscreen glyphs and to the corresponding offset between character codes in memory. You can use the location information obtained from hit-testing to set the insertion point (that is, the caret) or selection range (highlighting).

ATSUI does most of this work for you. It provides functions that convert a screen position to a memory offset (and vice versa). These functions work properly with bidirectional text and with text that has been rendered with ligatures and contextual forms.

Determining the character associated with a screen position requires first defining the caret position associated with a given screen position. Once that is done, the previously defined relationship between caret position and text offset can be used to find the character.

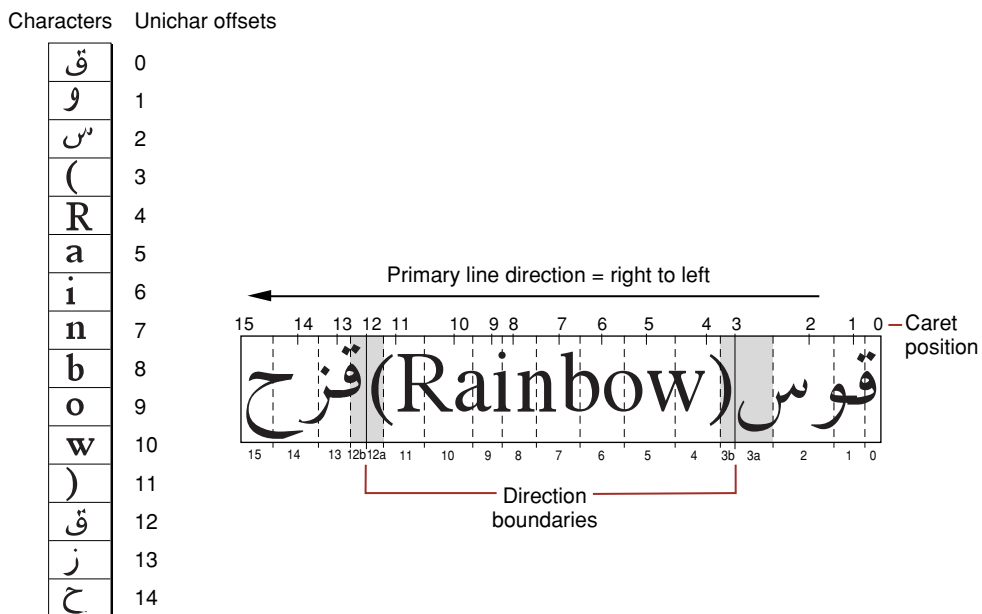
Figure 1-26 shows the cursor positioned within a line of text at the moment of a mouse click. A mouse-down event can occur anywhere within the area of a glyph, but the caret position that is to be derived from that event must be a thin line that falls between two glyphs.

Figure 1-26 Interpreting caret position from a mouse-down event



A line of displayed glyphs is divided by ATSUI into a series of mouse-down regions. A mouse-down region is the screen area within which any mouse click yields the same caret position. For example, a mouse click that occurs anywhere between the leading edge of a glyph and the center of that glyph results in a caret position at the leading edge of that glyph. For unidirectional text, mouse-down regions extend from the center of one glyph to the center of the next glyph (except at the ends of a line), as Figure 1-26 shows. A mouse click anywhere within the region results in a caret position between the two glyphs.

At line ends, and at the boundaries between text of different line directions, mouse-down regions are smaller and interpreting them is more complex. As Figure 1-27 shows, the mouse-down regions at direction boundaries extend only from the leading or trailing edges of the bounding glyphs to their centers. Note that the shaded part of Figure 1-26 is a single mouse-down region, whereas each of the shaded parts of Figure 1-27 are two mouse-down regions.

Figure 1-27 Mouse-down regions and caret positions in bidirectional text

How do mouse-down regions relate to offsets? Using Figure 1-27 as an example, consider the two mouse-down regions 3a and 12a. Keep in mind that the primary line direction is right to left.

- A mouse click within region 3a is associated with the trailing edge of the Arabic character. To insert Arabic text, your application might draw a primary caret (or single caret) at caret position 3, and place the insertion point at offset 3 in the buffer. (If you are drawing a dual caret, the secondary caret should be at caret position 12, which also corresponds to an insertion point at offset 4 in the buffer.)
- A mouse click within region 12a is associated with the leading edge of the Roman character. To insert Roman text, your application might, draw a secondary caret (or single caret) at caret position 12, and place the insertion point at offset 3 in the buffer. (If you are drawing a dual caret, the primary caret should be at caret position 3, which also corresponds to an insertion point at offset 3 in the buffer.)

Thus mouse clicks in two widely separated areas of the screen can lead to an identical caret display and to a single insertion point in the text buffer. One, however, permits insertion of Roman text, the other Arabic text, and the insertions occur at different screen locations.

Mouse clicks in regions 3b and 12b in Figure 1-27 would lead to just the opposite situation: a primary caret at caret position 12, a secondary caret at caret position 3, and an insertion point at offset 12 in the text buffer.

ATSUI Style and Text Layout Objects

This chapter describes the objects that ATSUI uses for text layout and style information: text layout objects and style objects. A **text layout object** contains the information that controls the display and formatting of the text associated with the text layout object. A **style object** contains a collection of stylistic attributes that can be applied to text runs within the text associated with a text layout object.

Any ATSUI function that sets or retrieves stylistic or layout information uses text layout and style objects, which is why you should read this chapter if you plan to call ATSUI functions in your code. This chapter

- describes the information contained in style and text layout objects
- provides a description of the ATSUI tags you use to specify style attributes and control text layout
- provides illustrations to show how settings effect the way ATSUI renders text

To understand this chapter, you should first read [“Typography Concepts”](#) (page 15).

Style Objects

ATSUI style objects are opaque objects that represent a collection of stylistic attributes. A style object can represent one or more attributes, including information about the following:

- **Style attributes.** You can set attributes such as font, size, color, and directionality, and you can override font-specified default behavior governing how glyphs in a style run are displayed and formatted, including such behavior as kerning and optical alignment.
- **Font features.** You can select or deselect font features (ligatures, swashes, and so forth) to specify whether to employ, and at what level, various special typographic and layout features provided by the font for a particular style run.
- **Font variations.** You can manipulate the font variation axes and values of the font used in a style run to get different stylistic variations built into the font that are available for drawing text.

In ATSUI, a style object is created by calling the function `ATSUCreateStyle` with an `ATSUStyle` data structure as a parameter. Because style objects are opaque, you must use ATSUI accessor functions to get or set style attributes, font features, font variations, and other information associated with a style object. You’ll see how to use those functions in [“Basic Tasks: Working With Objects and Drawing Text”](#) (page 69).

Once a style object has been defined, it can be applied to a run of text. A **style run** is a sequence of contiguous characters in memory that share the same style attributes, font features, and font variations. The same style object can be used by more than one style run in one or more text layout objects.

Note: ATSUI style objects are thread-safe; you can share them between threads.

The specific style attributes, font variations, and font features that you can manipulate using ATSUI are discussed in the sections that follow.

Style Attributes

Style attributes are a collection of values and settings that override the font-specified behavior for displaying and formatting text in a style run. For example, a text color of red is a style attribute that overrides the default text color of black.

ATSUI uses a triple to specify an attribute. A **triple** is an attribute tag, a value for that tag, and the size of the value. Attribute tags are constants. ATSUI provides tags for most style attributes. For example, the ATSUI tag for text color is `kATSURGBAlphaColorTag`. Its tag for font size is `kATSUSizeTag`. The triples associated with each of these tags are shown in Figure 2-1.

Figure 2-1 Triples for text color and font size style attributes

Tag: <code>kATSURGBAlphaColorTag</code>	Tag: <code>kATSUSizeTag</code>
Size: <code>sizeof(ATSURGBAlphaColor)</code>	Size: <code>sizeof(Fixed)</code>
Value: <code>{ 0, 0, 0, 1 }</code>	Value: <code>12.0</code>

As you can see in Figure 2-1, color is specified by data of type `ATSURGBAlphaColor`, so the size is `sizeof(ATSURGBAlphaColor)`. The value in the figure is the default color value, which specifies black with an alpha-channel (translucency) setting of 1—`(0, 0, 0, 1)`. Size is specified by a `Fixed` value. In this example, the value `12.0` specifies 12-point size.

This section discusses most of the style attributes for which ATSUI provides tags. It describes a style attribute, the default setting for the attribute, and in many cases shows the effect of applying the attribute to text. See *Inside Mac OS X: ATSUI Reference* for a complete reference of the style attribute tags provided by ATSUI.

You can create a custom attribute tag for a style attribute for which ATSUI does not provide a tag. For more information, see [“Custom Attribute Tags”](#) (page 53).

Ascent Attribute Tag

The ascent attribute tag represents the ascent associated with a style object’s font. You can obtain the ascent value by using the following style attribute tag:

Attribute tag:	<code>kATSUAscentTag</code>
Data type:	<code>ATSUTextMeasurement</code> ; the value must be ≥ 0 .
Default value:	The ascent value of the style’s font with the current point size.
Comment:	This tag is available starting with Mac OS X version 10.2.

Baseline Attribute Tag

The baseline attribute type represents the preferred baseline (such as Roman, hanging, or ideographic-centered) to use for text of a given font in a style run. You can set and retrieve the baseline type value using the following style attribute tag:

Attribute tag:	<code>kATSUBaselineClassTag</code>
Data type:	<code>BslnBaselineClass</code> (defined in <code>SFNTLayoutTypes.h</code>)
Default value:	<code>kATSURomanBaseline</code>
Comment:	If you want to use the intrinsic baselines, set the value of this tag to <code>kBSLNNoBaselineOverride</code> .

See [Figure 1-8](#) (page 24) for illustrations of baselines. For an example of how to apply a baseline attribute tag to a style, see [“Setting a Baseline”](#) (page 105).

Caret Angle Attribute Tag

The caret angle attribute tag specifies whether the text caret or edges of a highlighted area are always parallel to the slant of the style run’s text or always perpendicular to the baseline. You can set and retrieve the value of the caret angle using the following style attribute tag:

Attribute tag:	<code>kATSUNoCaretAngleTag</code>
Data type:	<code>Boolean</code>
Default value:	<code>false</code>
Comment:	The default setting is to use the character’s angularity to determine caret angle and the edges of a highlighted area.

For example, when the caret appears in italic text or text that has an intrinsic angle, you may want to display an angled (slanted) caret rather than a straight one. ATSUI supports this capability by using data present in a font that identifies the intrinsic font angle. [Figure 2-2](#) shows the position of straight (top line) and angled (bottom line) carets.

Figure 2-2 Straight and angled caret positions



Cross-Stream Kerning Attribute Tag

Cross-stream kerning is the movement of glyphs (as specified by the font) perpendicular to the line orientation of the text. (For horizontal text, the automatic movement is vertical.) Cross-stream kerning is required for such scripts as Taliq (used in Urdu). It can also be used to assist in the creation of automatic fractions. You can set and retrieve the cross-stream kerning value using the following style attribute tag:

Attribute tag:	kATSUSuppressCrossKerningTag
Data type:	Boolean
Default value:	false
Comment:	The default setting specifies not to suppress automatic cross-kerning (defined by the font).

Figure 2-3 shows how cross-stream kerning raises the minus sign between two uppercase glyphs to reflect the centers of those characters. The text on the left does not use cross-stream kerning whereas the text on the right does.

Figure 2-3 Cross-stream kerning applied to a minus sign in an equation

X – Y X – Y

For an example of how to apply the cross-stream kerning tag to a style, see [“Drawing Equations”](#) (page 79).

Descent Attribute Tag

The descent attribute tag represents the descent associated with a style object’s font. You can obtain the descent value by using the following style attribute tag:

Attribute tag:	kATSUDescentTag
Data type:	ATSUTextMeasurement; the value must be ≥ 0 .
Default value:	The descent value of the style’s font with the current point size.
Comment:	This tag is available starting with Mac OS X version 10.2. The leading value is not included as part of the descent.

Font ID Attribute Tag

A **font ID** is a value that identifies a font to the font management system. You can set and retrieve the font ID value using the following style attribute tag:

Attribute tag:	kATSUFontTag
Data type:	ATSUFontID
Default value:	The application font for the current script system.
Comments:	If the application font is not usable by ATSUI, the default is Helvetica.

The font ID is used by ATSUI functions that retrieve or manipulate fonts. When you copy a style object, you copy the font ID of the font used in the style run in addition to the style object's style attributes, font features, and font attributes. The font ID does not persist across system startups. In other words, a font does not have the same font ID when the computer is restarted.

Font Matrix Attribute Tag

You can associate a font transformation matrix with a style object to achieve such effects as reversing glyphs across the x-axis and rotating glyphs. You can set and retrieve the font matrix using the following style attribute tag:

Attribute tag:	<code>kATSUIFontMatrixTag</code>
Data type:	<code>CGAffineTransform</code>
Default value:	<code>CGAffineTransformIdentity</code> , which is <code>[1, 0, 0, 1, 0, 0]</code>
Comment:	This tag is available in Mac OS X version 10.2 and later.

Figure 2-4 shows four instances of the phrase “Hello World.” The first instance is drawn normally; no transformation is applied. The second and third instances are the result of scaling the font transformation matrix to stretch or shrink the glyphs. The fourth instance of “Hello World” is created by rotating the transformation matrix. Some of the glyphs overlap in this case because the rotation does not modify the advance widths.

Figure 2-4 Using a font transformation matrix to achieve text effects

The figure displays four lines of the text "Hello World!".
 1. The first line is "Hello World!" in a standard, upright orientation.
 2. The second line is "Hello World!" where the characters are stretched horizontally, making them wider.
 3. The third line is "Hello World!" where the characters are shrunk horizontally, making them narrower.
 4. The fourth line is "Hello World!" rotated counter-clockwise, causing the characters to overlap.

Glyph Orientation Attribute Tag

The **glyph orientation** of a style run specifies which direction (vertical or horizontal) glyphs should be drawn. You can set and retrieve the glyph orientation using the following attribute tag:

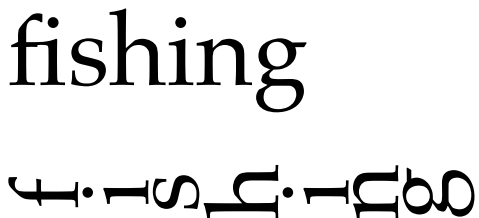
Attribute tag:	<code>kATSUVERTICALCharacterTag</code>
Data type:	<code>ATSUVERTICALCharacterType</code>

Default value: `kATSUStronglyHorizontal`
 Comment: The other possible value for this tag is `kATSUStronglyVertical`.

Note: Changing the glyph orientation affects the tracking settings. See “Tracking Setting Attribute Tag” (page 50).

Figure 2-5 shows two horizontal lines of text. The glyphs in the top line of Figure 2-5 are vertical and those in the bottom line are horizontal.

Figure 2-5 Vertical and horizontal glyphs



Glyph Selector Attribute Tag

Fonts that have large character sets, such as Chinese, Japanese, and Korean fonts, can be packaged as glyph collections, or character sets. Many characters in these large font sets are not readily accessible. Using the glyph selector tag allows you access to characters in the fonts by specifying a font-specific glyph ID or a collection ID (CID). You can use the glyph selector attribute tag to specify a glyph ID or collection by using the following style attribute tag:

Attribute tag: `kATSUGlyphSelectorTag`
 Data type: `ATSUGlyphSelector`
 Default value: `0`
 Comment: The default specifies to use the glyphs derived by the ATSUI layout process. This tag is available in Mac OS X version 10.2 and later.

For more information on CID conventions, see <http://www.adobe.com>. You can get the variant glyph information from an input method through the Text Services Manager using the Carbon event key, `kEventParamTextInputGlyphInfoArray`.

Glyph Width Attribute Tag

Glyph widths, by default, are specified by font-defined advance widths. You can override the glyph’s default metrics by imposing a width for ATSUI to use. You can set and retrieve the imposed-width value using the following style attribute tag:

Attribute tag: `kATSUImposeWidthTag`
 Data type: `ATSUTextMeasurement`; the value must be ≥ 0 .
 Default value: `kATSUNoImposedWidth`
 Comment: The default setting is for glyphs to use their own font-defined advance widths.

Imposing a width is useful if you need to embed a picture or other graphic in a line of text. Your application can create a gap at a specific point in that line by using a single whitespace character as its own style run and imposing a width on that style run, as shown in Figure 2-6. The specified glyph always has the imposed width, regardless of the point size of the text, to within a single pixel in device resolution.

Figure 2-6 shows text in which one of the style runs consists of only the whitespace character between the words “As” and “you”. The text is drawn twice. The first time (the top line) with no imposed width and then (the second line) with an imposed width on the whitespace character. By imposing a width on the style run, you can tailor the size of the gap between the words.

Figure 2-6 A line of text without and with an imposed glyph width

As you wish

As you wish

Hanging Punctuation Attribute Tag

Hanging punctuation refers to glyphs that extend beyond the left or right margins of the text area and whose widths are not counted when line length is measured. This property is font-specified, and is usually true for “lightweight” punctuation, such as quotation marks or periods. You can set and retrieve the hanging punctuation value using the following style attribute tag:

Attribute tag: `kATSUHangingInhibitFactorTag`
 Data type: `Fract`; must be a fractional value between 0 and 1.0.
 Default value: 0
 Comment: The default setting specifies no adjustment to the font-specified data.

The ability to control whether punctuation glyphs can hang permits automatic alignment of text lines such as that shown in Figure 2-7. Hanging glyphs typically extend beyond the left and right margins of the text area. Their widths are not counted when line length is measured. In Figure 2-7 the quotation mark on the left and the period on the right are hanging punctuation glyphs.

Figure 2-7 Hanging punctuation glyphs

*“The quick brown fox jumps
 over the lazy dog,” said the sage.*

By default, ATSUI uses font-specific information to automatically hang punctuation where appropriate. A value of 0 (the default) indicates that the glyph should hang by the normal amount. Your application has the ability to control the degree to which hanging punctuation occurs by supplying a value other than 0. A positive nonzero value lessens the amount of hanging proportionally; a value of 1 means “no hanging at all.” Figure 2-8 shows the same line of (justified) text laid out with the default value (top), with a value of 0.5 (middle), and with a value of 1.0 (bottom).

Figure 2-8 Effects of changing the hanging punctuation attribute

“It is ‘a great effect!’”

“It is ‘a great effect!’”

“It is ‘a great effect!’”

Forced Hanging Glyphs Attribute Tag

You can treat glyphs in a style run as hanging punctuation, whether or not the font designer intended them to be. You can set and retrieve the forced hanging punctuation setting using the following style attribute tag:

Attribute tag: `kATSUForceHangingTag`
 Data type: `Boolean`
 Default value: `false`
 Comment: The default setting specifies not to force the character to hang beyond line boundaries.

Figure 2-9 shows a line in which the question mark, which is not normally a hanging punctuation glyph, is in its own style run and is defined as hanging, causing the question mark to extend beyond the margin.

Figure 2-9 A question mark forced to extend beyond the margin

“Is it ‘a great effect’?”

Justification Override Attribute Tag

Justification override represents the degree to which ATSUI should override justification behavior for glyphs in the style run. You can set and retrieve the justification override value using the following style attribute tag:

Attribute tag: `kATSUPriorityJustOverrideTag`
 Data type: `ATSUJustWidthDeltaEntryOverride`
 Default value: Each field has a value of 0
 Comment: The default setting specifies no overrides.

The `ATSUJustWidthDeltaEntryOverride` structure contains an array of four width delta structures, one for each priority level, in index order. The width delta structure contains the information needed to override the distribution behavior of a glyph or set of glyphs during justification. You can use these structures to specify both a change in priority level and distribution behavior.

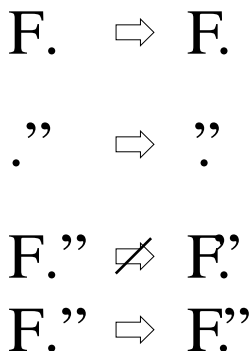
Kerning Attribute Tag

Kerning increases the overlap between glyphs that fit together naturally. It does not apply evenly to all glyphs in a style run. You can set and retrieve the kerning value using the following style attribute tag:

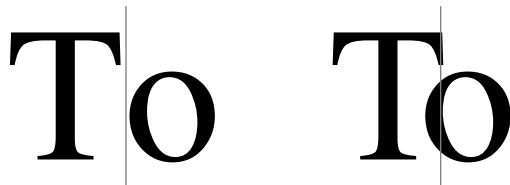
Attribute tag:	<code>kATSUKerningInhibitFactorTag</code>
Data type:	<code>Fract</code> ; must be a fractional value between 0 and 1.0.
Default value:	0
Comment:	The default setting specifies no inhibition of font-specified kerning.

ATSUI uses information in font tables to determine how much to increase or decrease the space between glyphs. In the general case, this amount can depend on more than just the two adjacent glyphs: It can also depend on preceding or following glyphs, or even on glyphs in other parts of the line. For example, the two pairs of glyphs in Figure 2-10 might kern, but the triple would not—at least not in the same manner as the two separate pairs.

Figure 2-10 When kerning can and cannot occur



To determine caret positions, kerning offset is effectively split between glyphs in a kerned pair. The example in Figure 2-11 shows where the caret would appear between the two glyphs without kerning (left side) and with kerning (right side).

Figure 2-11 Caret position between two kerned glyphs

Kerning does not refer to the apparent overlap that can be caused by glyphs that overhang their bounds (glyphs that extend beyond their leading or trailing edges defined by the character origin and advance width), as shown by the Q in Figure 2-12.

Figure 2-12 Apparent kerning by a glyph that extends beyond its advance width

Language Region Attribute Tag

The language region attribute tag represents the regional language code for glyphs in a style run. ATSUI uses the value associated with this tag to determine how to render region-dependent characteristics. You can set and retrieve the language region value using the following style attribute tag:

Attribute tag:	<code>kATSULangRegionTag</code>
Data type:	<code>RegionCode</code>
Default value:	<code>GetScriptManagerVariable (smRegionCode)</code>
Comment:	Region codes are defined in the Script Manager.

Leading Attribute Tag

The leading attribute tag represents the leading associated with a style object's font. You can obtain the leading value by using the following style attribute tag:

Attribute tag:	<code>kATSULEadingTag</code>
Data type:	<code>ATSUTextMeasurement</code> ; the value must be ≥ 0 .
Default value:	The leading value of the style's font with the current point size.
Comment:	This tag is available in Mac OS X version 10.2 and later.

Ligature Decomposition Attribute Tag

Ligature decomposition is the breaking up of a ligature into its component glyphs during justification, so that the individual glyphs may more evenly occupy the space allotted to the ligature. You can set and retrieve the ligature decomposition value using the following style attribute tag:

Attribute tag:	<code>kATSUDecompositionFactorTag</code>
Data type:	Fixed; the value must be ≥ -1.0 and ≤ 1.0
Default value:	0
Comment:	The value associated with this tag represents the fractional adjustment to the font-specified threshold at which ligature decomposition occurs during justification, with 0 representing no adjustment to the font-specified threshold.

Depending on the amount of white space surrounding a ligature, postcompensation action may replace that ligature with its component glyphs, after which ATSUI recalculates the positions of all glyphs on the line.

Ligature Splitting Attribute Tag

Ligature splitting is the division of a ligature for hit-testing purposes into regions corresponding to each of its component glyphs. You can set and retrieve the ligature splitting value using the following style attribute tag:

Attribute tag:	<code>kATSUNoLigatureSplitTag</code>
Data type:	Boolean
Default value:	false
Comment:	The default setting specifies that ligatures and compound characters in a style have divisible components.

If the value set by the ligature splitting attribute tag is `true` and the caret position is adjacent to a ligature, ATSUI considers the next valid caret position to be at the other side of the entire ligature rather than at any point within it.

Optical Alignment Attribute Tag

Optical alignment is the fine adjustment of glyph positions (as specified by the font) at the ends of lines to give a more even visual appearance to margins. You can set and retrieve the optical alignment value using the following style attribute tag:

Attribute tag:	<code>kATSUNoOpticalAlignmentTag</code>
Data type:	Boolean
Default value:	false
Comment:	The default setting specifies not to suppress automatic optical positioning alignment.

In multiline text, glyphs may seem to line up incorrectly at the margins. This is accounted for by two factors. First, glyph advance widths contain a certain amount of extra white space (side bearing) to account for the normal interglyph spacing. This produces certain anomalies at line margins, because the side bearing varies with font size.

The second problem is that due to optical effects, curved lines do not appear to line up properly with straight lines. To make them appear to line up, some compensation must occur. On baselines, for example, curved letters such as “C” or “S” are generally designed to extend slightly below the baseline, so that they appear to line up with straight letters such as “H”.

This same effect should happen at the edges of lines. On the left side of Figure 2-13, the “O” in “Oregon” and the “C” in “Connecticut” appear to be indented compared to the “H” and “D” glyphs. However, as shown by the vertical line on the right, the outlines of the four glyphs are exactly aligned. The apparent indentation is an optical effect.

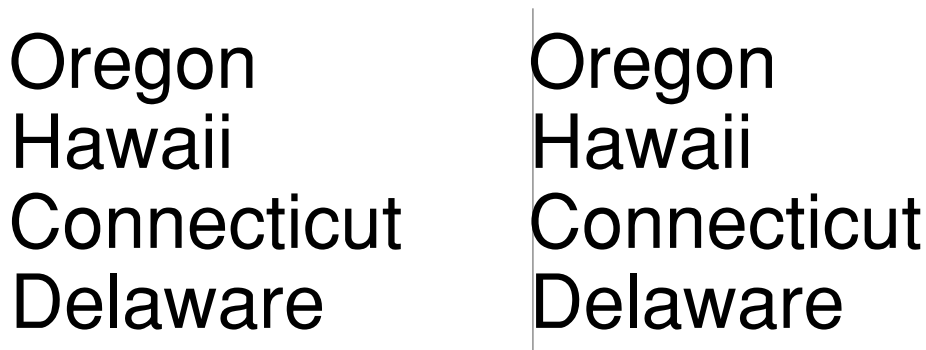
Figure 2-13 Apparent misalignment of curved letters



To compensate for these effects, ATSUI applies optical alignment information contained in the font. When determining the leading and trailing edges of a line of text, ATSUI uses the optical leading and trailing edges as specified by the font.

Figure 2-14 shows the same text as Figure 2-13, except that on the left in Figure 2-14, the glyphs appear to be aligned. However, as shown by the vertical line on the right, the outlines of the four glyphs are not exactly aligned; the glyphs have been shifted to compensate for optical effects.

Figure 2-14 Optical alignment at line edges



Postcompensation Justification Attribute Tag

Postcompensation justification is a set of processes (such as glyph stretching and ligature decomposition) that occur at the end of the justification process and should take place after glyph positions have been calculated. You can set and retrieve the postcompensation justification value using the following style attribute tag:

Attribute tag:	<code>kATSUNoSpecialJustificationTag</code>
Data type:	<code>Boolean</code>
Default value:	<code>false</code>
Comment:	The default setting specifies to perform postcompensation justification if needed.

Style Rendering Options Attribute Tag

Style rendering options provide you with fine control over how ATSUI renders a style. You can specify whether or not ATSUI should use hinting and whether or not anti-aliasing should be used. You can set and retrieve the style rendering option value using the following style attribute tag:

Attribute tag:	<code>kATSUStyleRenderingOptionsTag</code>
Data type:	<code>ATSStyleRenderingOptions</code>
Default value:	<code>kATSStyleNoOptions</code>
Comment:	See <i>Inside Mac OS X: ATSUI Reference</i> for a complete list of style rendering options.

Style Text Locator Attribute Tag

ATSUI uses text-locator information to determine where to break text and which options to apply to text breaks. The attribute value associated with this tag (`TextBreakLocatorRef`) contains the data needed by ATSUI to locate various kinds of text breaks. You can set and retrieve the style text locator value using the following style attribute tag:

Attribute tag:	<code>kATSUStyleTextLocatorTag</code>
Data type:	<code>TextBreakLocatorRef</code> (an opaque structure defined in Unicode Utilities)
Default value:	<code>NULL</code>
Comment:	The default value specifies to use the region-derived locator or the default Text Utilities locator.

Text Color (RGB) Attribute Tag

The text color attribute tag represents the color of the glyphs in a style run. You can set and retrieve the text color using the following style attribute tag:

Attribute tag:	<code>kATSUColorTag</code>
Data type:	<code>RGBColor</code>
Default value:	<code>(0,0,0)</code>
Comment:	The default setting designates black.

The `RGBColor` structure lets you specify intensity values for the three additive primary colors—red, green, and blue.

Text Color (RGB Alpha) Attribute Tag

The text color attribute tag represents the color of the glyphs in a style run and the alpha channel (translucency) setting for those glyphs. You can set and retrieve the text color and alpha channel setting using the following style attribute tag:

Attribute tag:	kATSURGBAlphaColorTag
Data type:	ATSURGBAlphaColor
Default value:	(0,0,0,1)
Comment:	The default setting designates black, with an alpha-channel setting of 1.

The `ATSURGBAlphaColor` structure lets you specify intensity values for the three additive primary colors—red, green, and blue—as well as an alpha-channel value.

Text Size Attribute Tag

The text size attribute tag represents the size, in typographic points (72 per inch), of the text in the style run. You can set and retrieve the text size using the following style attribute tag:

Attribute tag:	kATSUSizeTag
Data type:	Fixed
Default value:	The application font size for the current script system.

ATSUI uses `Fixed` values to specify points for coordinates and measurements, rotated text, and angled and dual carets, unlike `QuickDraw`, which uses integer coordinates.

Tracking Setting Attribute Tag

Tracking represents the relative proportion of font-defined adjustments to apply to interglyph positions. You can expand or contract the spacings of all glyphs in a style run by applying a tracking value, called the **tracking setting**, to that style run. You can set and retrieve the tracking setting using the following style attribute tag:

Attribute tag:	kATSUTrackingTag
Data type:	Fixed
Default value:	0
Comment:	No change to the font-defined adjustments.

Tracking is different from with-stream shifting because the actual amount of space added or removed is controlled by the font, not by your application. The positional shifts are the result of two-dimensional interpolation based on the tracking setting, the text size in points, and the threshold values present in the font's tracking table. These threshold values are used to permit nonlinear tracking amounts. For example, a single tracking setting can specify different sets of spacings for text below 8 points, from 8 to 12 points, from 12 to 15, from 15 to 36, and over 36 points if the font designer specifies it.

Specifying a tracking setting of 0 means “space normally” according to the specifications set by the font designer. That does not necessarily mean that no adjustment to spacing occurs. The font designer may decide that “normal spacing” includes some spacing adjustment in certain point size ranges.

The glyph orientation also affects the tracking settings. A font designer can specify different spacings for a tracking setting of 0 (normal spacing) depending on the orientation of the glyphs.

Figure 2-15 shows the same phrase written three times in a particular font. The top line is drawn when the application specifies a tracking setting of 0; the middle is drawn with a large positive tracking setting (+2); and the bottom with a large negative tracking setting (-2).

Figure 2-15 Text drawn with three different tracking settings

Tracking can be loose or tight

Tracking can be loose or tight

Tracking can be loose or tight

Text Style Attribute Tags

Text styles are the visual attributes, other than size, applied as a systematic variation to the plain (unstyled) characteristics of a font's glyphs. The set of text styles supported by ATSUI include bold, italic, underline, condensed, and extended.

Text style attribute tags are included for compatibility with the `Style` type used by QuickDraw's `TextFace` function. If a font variant for one of these text styles exists, that variant is used. Otherwise, the variant is generated algorithmically. QuickDraw-compatible style features are not supported by all fonts. Some fonts have finer controls for setting QuickDraw text styles; these controls can be accessed with font variations.

The tags you can use to set and retrieve text style attributes are listed in Table 2-1 along with the data type for each tag and its default value.

Table 2-1 Style attribute tags, data types, and default values for text styles

Attribute tag	Data type	Default value	Comments
kATSUQDBoldfaceTag	Boolean	false	The default specifies not boldfaced
kATSUItalicTag	Boolean	false	The default specifies not italicized
kATSUUnderlineTag	Boolean	false	The default specifies not underlined
kATSUCondensedTag	Boolean	false	The default specifies not condensed
kATSUExtendedTag	Boolean	false	The default specifies not extended

With-Stream Shift and Cross-Stream Shift Attribute Tags

With-stream shift represents a uniform shift parallel to the baseline of the positions of individual pairs or sets of glyphs in the style run. It can be used for manual kerning or letter spacing. You can apply with-stream shift before (to the left of) the glyphs of the style run using the following style attribute tag:

Attribute tag: `kATSUBeforeWithStreamShiftTag`
 Data type: Fixed
 Default value: 0
 Comment: The default value represents no adjustment the amount of space to the left edge (or top, for vertical text).

You can apply with-stream shift after (to the right of) the glyphs of the style run using the following style attribute tag:

Attribute tag: `kATSUAfterWithStreamShiftTag`
 Data type: Fixed
 Default value: 0
 Comment: The default value represents no adjustment to the amount of space to add to the right edge (or bottom, for vertical text).

You can apply both tags (before and after) to the same style run. Positive with-stream shift moves the glyphs farther apart; negative shift moves them closer together.

Cross-stream shift represents the distance to raise or lower glyphs in the style run perpendicular to the text stream. This shift is vertical for horizontal text and horizontal for vertical text. Each glyph in the style run is shifted by the same amount from the baseline. It can be used for superscript and subscript effects. Positive cross-stream shift moves the glyphs upward from the baseline (as in superscripts); negative shift moves them downward (as in subscripts). You can apply cross-stream shift using the following style attribute tag:

Attribute tag: `kATSCrossStreamShiftTag`
 Data type: Fixed
 Default value: 0
 Comment: The default value represents no adjustment.

Figure 2-16 shows an example of a negative and a positive with-stream shift applied before (to the left of) the glyphs of a style run. The glyphs “c” and “d” constitute a single style run. The line is drawn first with no shift, then with a large negative with-stream shift for that style run, and finally with an even larger positive with-stream shift.

Figure 2-16 Negative and positive with-stream shift

abcdef abꞖdef ab c def

Figure 2-17 illustrates the simultaneous use of with-stream and cross-stream shifts. The layout consists of two style runs of a single glyph each. On the left the layout is drawn with no shifting. On the right, a negative with-stream shift is applied before the “2”, and a positive cross-stream shift is applied to the “2”. The net result is a well-proportioned and well-placed superscript. (There are other ways to make superscripts, including the use of superiors; see “[Vertical Position Feature Type](#)” (page 158).)

Figure 2-17 Combined with-stream and cross-stream shift

When text is shifted in a with-stream direction, the boundary (caret position) between pairs of glyphs is adjusted to be halfway between the advance width of the earlier glyph and the origin of the later glyph, as shown in Figure 2-18. Using with-stream and cross-stream shifts can give your application a full manual letter-spacing capability.

Figure 2-18 Caret position between with-stream shifted glyphs

Custom Attribute Tags

If Apple provides a tag for an attribute, you should use the tag. In the rare case in which there isn't a tag available for an attribute you want to control, you can create your own custom tag. Apple reserves values 0 to 65,535 (0 to 0x0000FFFF) for Apple-defined attribute tags. Your attribute tag must have a value outside this range.

Font Variations

As you may recall from “[Font Variations](#)” (page 21), a font variation is a setting along a particular style variation axis defined by the font designer. The font variations associated with a style object specify the variations that can be applied to the style's font. A style object may specify the values for any number of variation axes.

You can use ATSUI to obtain the variation axes and variation settings defined for a font. You can also use ATSUI to supply your own setting for any variation axes defined for the font. If the font does not support the variation axis you specify, your custom variations have no visual effect.

Font Features

Font features are typographic and layout capabilities that can be selected or deselected by an application and that control many aspects of glyph selection, ordering, and positioning. Font features include fundamental controls such as whether or not contextual forms are to be used and details of appearance such as whether or not alternate forms of glyphs are to be used at the beginning of a word. To a large extent, how text looks when it is laid out is a function of the number and kinds of font features chosen.

Font vendors create tables that implement a set of font features from which your application can pick and choose. The architecture of font features is open-ended; as font vendors create new kinds of features, ATSUI automatically takes advantage of them. When your application selects a group of features, ATSUI uses them, plus any font-specified features not overridden by your feature selections, when it draws the text layout object.

Font features are grouped into categories called **feature types**, within which individual **feature selectors** are used to define particular feature settings or selections. Feature types can be exclusive or nonexclusive. If a feature type is **exclusive** you can choose only one of the available feature selectors, such as whether numbers are to be proportional or fixed-width. If a feature type is **nonexclusive**, you can enable any number of feature selectors at once. For example, for the ligature feature type you can choose any combination of the available classes of ligatures that the font supports.

Note: You can select only those features added to a font by the font designer. If you select features that are not available in a font, you won't see a change in the glyph's appearance.

Some features are contextual while others are noncontextual. The manner in which **contextual features** are applied to a glyph depends on the glyph's position compared to adjacent glyphs. Much of ATSUI's text layout power results from its ability to apply sophisticated contextual processing automatically.

Noncontextual features are applied in the same manner to a glyph regardless of the adjacent glyphs. These features include the selection of alternate glyph sets to give text a different appearance and glyph substitution for purposes of mathematical typesetting or enhancing typographic sophistication.

"Font Features" (page 135) describe font features types and the selectors available for each feature type. As feature types can be added at any time, you should check Apple's font feature registry website for the most up-to-date list of font feature types and selectors:

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

Text Layout Objects

Text layout objects contain the stylistic information that influences the display and formatting of the text associated with the text layout object. In ATSUI, you can create a text layout object by calling the function `ATSUCreateTextLayout` with an `ATSUTextLayout` data structure as a parameter. Like style objects, text layout objects are opaque. You use ATSUI accessor functions to retrieve and manipulate the data associated with a text layout object.

Note: Beginning with Mac OS X version 10.2, ATSUI text layout objects are thread-safe; you can share them between threads.

Each text layout object contains the text layout attributes, soft line breaks, and style runs that have been set for a block of Unicode text. The text layout object keeps track of the following information:

- A pointer to the Unicode text string associated with the text layout object, and the length of the text associated with the object (which could also include a subrange of the text). See “[Text Information](#)” (page 55).
- The number of style runs in the text layout object, the length (in bytes) of each style run, and a list of references to the style objects that correspond to the text layout object’s style runs. See “[Style Run Information](#)” (page 57).
- Attributes that control the layout of the text at the line and paragraph level. They influence the width of the text area from the left margin to the right margin, the alignment of the text, the justification of the text, the rotation of the text, the direction of the text, the locations of the various baselines for the text, and other layout controls. See “[Line and Layout Attributes](#)” (page 57).

Text Information

When you create a text layout object, you need to provide a pointer to a Unicode text buffer along with information about the text. You must allocate and manage the memory for the text associated with the text layout object.

The following is a list of the text-related information you must supply to ATSUI for each text layout object you create:

- address of the start of the text array in memory

ATSUI assumes that the block of text associated with a text layout object is found in memory as a contiguous block. To handle text that is broken up into discontinuous memory blocks, you must use one text layout object for each block.
- offset to the beginning of the portion of text to be drawn

You must specify the portion of the text buffer to which ATSUI should apply style attributes. This allows you to draw only a portion of text—for example, the visible page and the areas immediately around it—and avoid the overhead of drawing the rest of the text. The starting position is represented as a double-byte character offset of type `UniCharArrayOffset` relative to the beginning of the text buffer.

ATSUI assumes that the memory between the beginning of the text and the beginning of the text that is to be drawn, contains text. ATSUI may need to examine the text prior to the starting offset to read Unicode control characters that are embedded in the Unicode text string.
- length of the text to be drawn

The length of the text is measured in the number of double-byte Unicode characters of type `UniCharCount`.
- total length (offset plus length of text to be drawn)

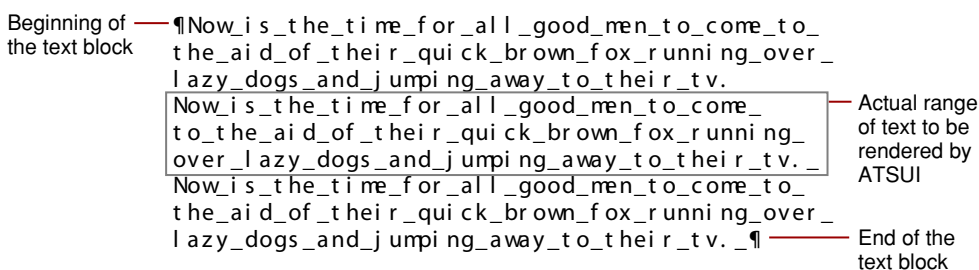
For best text-rendering performance, an ATSUI text layout object should contain at least a paragraph of text.

Text Length and Unicode Separator Characters

Typically the text associated with a text layout object spans a block of Unicode text—a sequence of characters terminated by Unicode block separator characters such as hard breaks or tabs. However, this does not have to be the case. The text associated with a text layout object may span multiple blocks of Unicode text or part of a block. Regardless of whether the text is shorter, longer, or equal to a block of text as defined by Unicode, ATSUI always behaves as if there are Unicode block separator characters preceding and terminating the text associated with the text layout object.

Figure 2-19 shows the text associated with a text object and calls out the portion of the text to be rendered by ATSUI. Even though ATSUI needs to draw only a portion of the text, ATSUI might scan the remainder of the text block for formatting controls or other information.

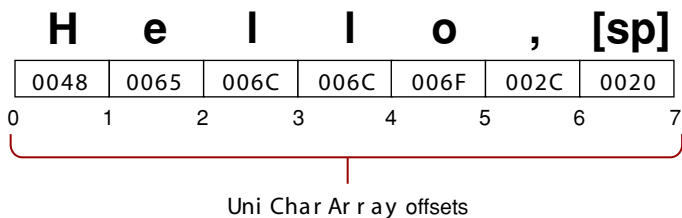
Figure 2-19 A range of text in a text block



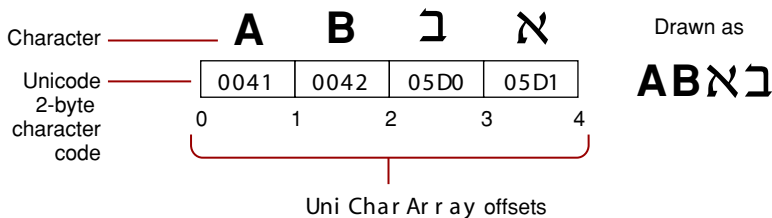
More About Text Offsets

Although ATSUI requires Unicode text, you should not assume that the caret can move through the text one character at a time by moving at least two bytes at a time. The backing store position of each character in the text relative to the beginning of the text buffer is specified by a double-byte character offset of type `UniCharArrayOffset`, illustrated in Figure 2-20. The `UniCharArrayOffset` is a zero-based offset that represents the position between two Unicode characters in a text buffer. Because of the possible presence of surrogates, the double-byte `UniCharArrayOffset` edge offset does not necessarily represent a position between two logical Unicode characters.

Figure 2-20 Text offsets of type `UniCharArrayOffset`



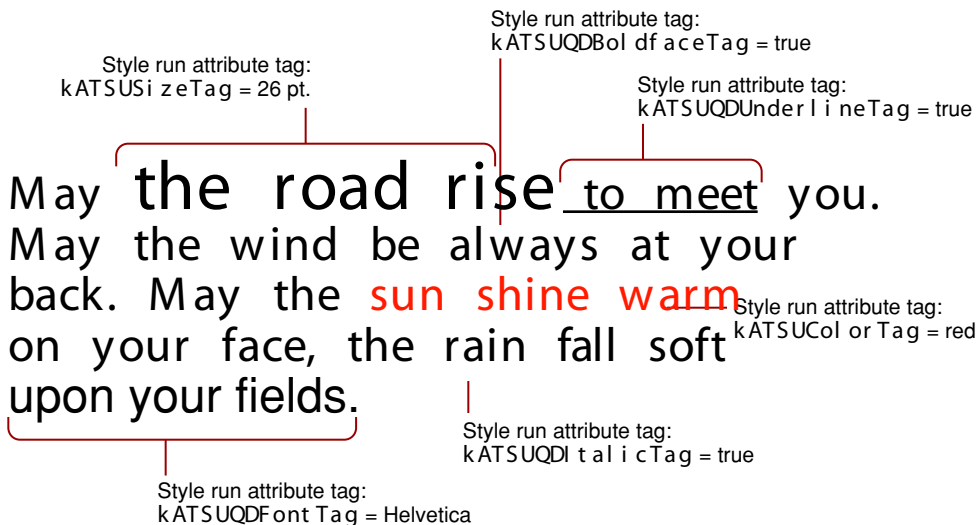
Some ATSUI functions need more information than a text offset. They also need to know whether the offset comes after the preceding character or before the next character. This is important for bidirectional text, where some of the text has a left-to-right direction and some has a right-to-left direction, as shown in Figure 2-21.

Figure 2-21 Bidirectional text

Style Run Information

The text associated with a text layout object is divided into one or more style runs. Each style run is specified by a style object that contains a collection of style attributes, font features, and font variations that influence the display and formatting of a style run in a text layout object.

Figure 2-22 shows text that's divided into twelve style runs. Six of the style runs in the figure are marked with the style attribute unique to that run. The other six runs that aren't marked have two attributes set—Palatino font and a point size of 18.0.

Figure 2-22 Text that has twelve style runs

When you create a text layout object, you must provide the number of style runs in the text, the length of each style run, and the style object for each run. Style objects and style attributes are discussed in detail in [“Style Objects”](#) (page 37).

Line and Layout Attributes

Line and layout attributes control how the lines of text associated with the text layout object are displayed and formatted. Similar to style attributes, ATSUI uses a triple to specify line and layout attributes: an attribute tag, the value of the attribute it sets, and the size (in bytes) of the attribute value. Attribute tags are constants supplied by ATSUI. Attribute values may be a scaler, a structure, or a pointer.

This section discusses most of the line and layout attributes for which ATSUI provides tags. It describes a line or layout attribute, the default setting for the attribute, and in many cases shows the effect of applying the attribute to a text layout. For the complete list of line and layout attributes that can be applied to a text layout object, see *Inside Mac OS X: ATSUI Reference*.

You can create a custom line or layout attribute for an attribute for which ATSUI does not provide a tag. For more information, see “[Custom Attribute Tags](#)” (page 53).

Alignment (Flushness) Attribute Tag

Alignment, or flushness, is the process of placing text in relation to one or both margins, which are the left and right sides (or top and bottom sides) of the text area. You can set and retrieve the alignment value using the following line and layout control attribute tag:

Attribute tag:	<code>kATSULineFlushFactorTag</code>
Data type:	<code>Fract</code>
Default value:	<code>kATSUStartAlignment</code>
Comment:	Text is drawn to the right of the left margin for horizontal text, or below the top margin for vertical text.

Text can be aligned left (`kATSUStartAlignment`), right (`kATSUEndAlignment`), or center (`kATSUCenterAlignment`), as shown in [Figure 1-16](#) (page 27). You can also specify a fractional value to align text at any location between the margins. Note in [Figure 1-16](#) (page 27) how the words of the text are spaced normally. Unlike justification (see “[Justification Attribute Tag](#)” (page 60)), alignment does not affect the spacing between words or individual glyphs.

The alignment attribute has an effect only with text whose width is shorter than the width specified by the width attribute (`kATSULineWidthTag`), as shown in [Figure 2-23](#). You must specify a width if you want to have any alignment other than `kATSUStartAlignment`.

Figure 2-23 Use of the alignment attribute with text whose width is shorter than the line’s width



See “[Justification Attribute Tag](#)” (page 60) for additional information on how the width, alignment, and justification attributes interact.

Baseline Offsets Attribute Tag

Baseline offsets specify the positions of different baseline types with respect to one another in a line of text. You can set and retrieve these values using the following line and layout control attribute tag:

Attribute tag: `kATSULineBaselineValuesTag`
 Data type: `BsInBaselineRecord`
 Default value: 0 for each entry in the record

In general, a style run has a default baseline, the line to which all glyphs are visually aligned when the text is laid out. For example, in a run of Roman text, the default baseline is the Roman baseline, upon which glyphs sit (except for descenders, which extend below the baseline). In some other writing systems, glyphs hang from the baseline. When text in a line comprises runs using multiple baselines, the text layout object uses information in the baseline record to determine how to align the runs with each other.

The baseline structure contains an array of distances, in points, from a delta of 0 from the y-coordinate of the layout's origin to the other baseline types the text layout object contains. Positive values indicate baselines above the default baseline and negative values indicate baselines below it. ATSUI can use these values to position text relative to the default baseline.

Figure 2-24 shows an example of text with multiple baselines aligned according to information in the baseline structure. Baseline types are described in detail in [“Baselines”](#) (page 23).

Figure 2-24 Text with multiple baselines aligned to the default baseline



Font Fallbacks Attribute Tag

The font fallbacks attribute specifies a font fallback object to use with a text layout. You can set and retrieve the font fallback object for a text layout using the following line and layout control attribute tag:

Attribute tag: `kATSULineFontFallbacksTag`
 Data type: `ATSUFontFallbacks`
 Default value: `NULL`

Before you use this tag, you must associate a font fallback method with a font fallback object. Then you can call the function `ATSUSetLayoutControls`, passing as parameters the tag `kATSULineFontFallbacksTag` and the font fallback object you set up previously. For more information on using the font fallbacks attribute tag see [“Using Font Fallback Objects”](#) (page 103).

Justification Attribute Tag

Justification is the process of typographically fitting a line of text to a given width (or height, in the case of vertical text). You can set and retrieve the justification value using the following line and layout control attribute tag:

Attribute tag: `kATSULineJustificationFactorTag`
 Data type: `Fract`
 Default value: `kATSUNoJustification`

In ATSUI some of the information that controls justification behavior is contained in the text layout object, whereas other information is contained in the style objects associated with the text layout object. There are several different kinds of justification, including justification with white space, kashidas, glyph deformation, and ligature decomposition.

Note: Justification can occur only when a line width is also specified.

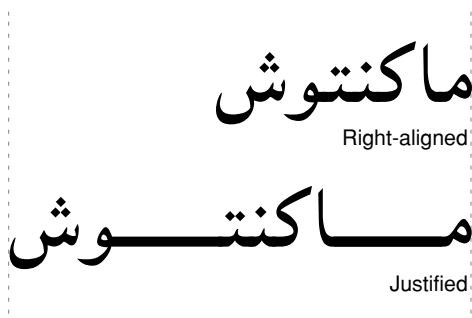
ATSUI justifies Roman text primarily by adjusting the white space between words and glyphs, as shown in Figure 2-25. Note that white space is added not only between words but also between glyphs. Also notice that ligatures such as “fi” and “ff” can be broken during justification.

Figure 2-25 Alignment and justification of Roman text



When Arabic text is justified, ATSUI distributes the available white space on the line by automatically lengthening or shortening the **kashidas**, which are the extender bars stretching between some of the glyphs of a word, as shown in Figure 2-26.

Figure 2-26 Alignment and justification in Arabic



The value of the justification attribute can have fractional values from 0 through 1. A value of 0.0 means no justification; a value of 1.0 means full justification using the width set by the width attribute tag `kATSULineWidthTag`. ATSUI interprets intermediate values, such as 0.5, to mean partial justification (also called ragged justification). Figure 2-27 shows a line with no justification followed by a partially justified line and a fully justified line.

Figure 2-27 Use of the justification line and layout control attribute

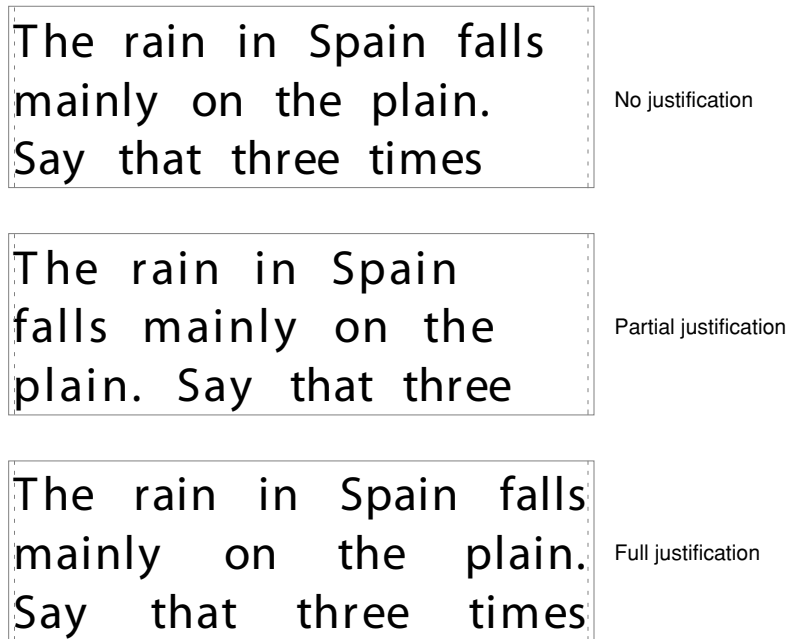
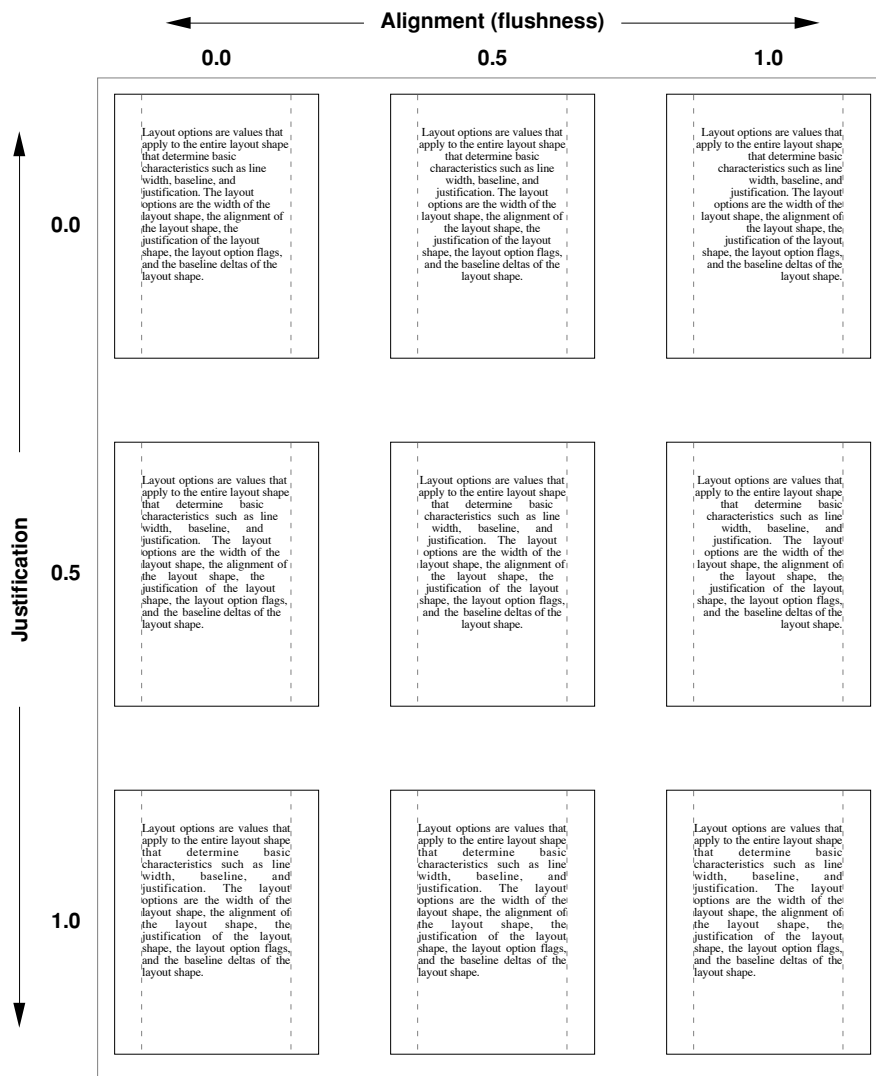


Figure 2-28 shows how the justification and alignment line and layout attributes interact with values ranging from 0.0 to 1.0. Note that when the user chooses full justification—that is, when the line justification attribute equals 1.0—the value set by the alignment attribute tag (`kATSULineFlushFactorTag`) has no effect.

Figure 2-28 Justification and alignment in a text layout object



The width, justification, and alignment attributes interact, and may produce a text layout other than what you expect. Table 2-2 describe the text layout that results from the interaction of these three attributes set at various values.

Table 2-2 Interactions between the width, justification, and alignment attributes

Width	Justification	Alignment	Effect
0	0	0	Text is drawn flush left.
0	0	> 0	Text is drawn proportionally around the origin. For example, if the value set by the attribute tag <code>kATSULineFlushFactorTag</code> equals 1.0, the right edge of the text aligns to the origin.
0	> 0	0	Text is drawn compressed, flush left.

Width	Justification	Alignment	Effect
0	> 0	> 0	Text is drawn compressed at the point specified by the attribute tag <code>kATSULineFlushFactorTag</code> .
> 0	0	0	Text is drawn flush left, unless the unjustified width is greater than the specified width. In this case, the text is drawn compressed into the specific width.
> 0	0	> 0	Text is drawn at the point specified by the attribute tag <code>kATSULineWidthTag</code> within the specified width (rather than around the origin, as happens when the width is 0). If the unjustified width is greater than the specified width, the text is drawn compressed into the specified width.
> 0	> 0	0	Text is drawn in the specified width, flush left, unless the value set by the attribute tag <code>kATSULineJustificationFactorTag</code> is equal to 1.0. In this case, both edges are flush.
> 0	> 0	> 0	Text is drawn in the specified width at the point specified by the attribute tag <code>kATSULineFlushFactorTag</code> within the specified width, unless the value set by the attributes tag <code>kATSULineJustificationFactorTag</code> is equal to 1.0. In this case, both edges are flush.

Language Region Attribute Tag

The language attribute tag represents the regional language code for glyphs in a line or text layout. ATSUI uses the value associated with this tag to determine how to render region-dependent characteristics. You can set and retrieve the language region value using the following line and layout control attribute tag:

Attribute tag: `kATSULineLangRegionTag`
 Data type: `RegionCode`
 Default value: `kTextRegionDontCare`

Layout Operation Override Specifier Attribute Tag

The layout operation attribute specifies a layout operation (such as morphing or kerning) that you want to handle instead of letting ATSUI perform it. You can set and retrieve the layout operation value using the following line and layout control attribute tag:

Attribute tag: `kATSULayoutOperationOverrideTag`
 Data type: `ATSULayoutOperationOverrideSpecifier`
 Default value: `NULL`
 Comment: This attribute is available starting with Mac OS X version 10.2.

For information on how to use the layout operation override tag, see [“Overriding ATSUI Layout Operations”](#) (page 113).

Line Ascent Attribute Tag

The line ascent attribute tag represents the ascent associated with a line of text. You can obtain the line ascent value by using the following line and layout control attribute tag:

Attribute tag:	<code>kATSULineAscentTag</code>
Data type:	<code>ATSUTextMeasurement</code>
Default value:	Maximum typographical ascent of all fonts used in a line of text or a text layout.

You can retrieve the line ascent value when you need to calculate line height. See [“Calculating Line Height”](#) (page 86).

Line Descent Attribute Tag

The line descent attribute tag represents the sum of the descent and leading associated with a line of text or a text layout object. You can obtain the line descent value by using the following line and layout control attribute tag:

Attribute tag:	<code>kATSULineDescentTag</code>
Data type:	<code>ATSUTextMeasurement</code>
Default value:	The maximum line descent and leading of all fonts in a line of text or in a text layout object.

You can retrieve the line descent value when you need to calculate line height. See [“Calculating Line Height”](#) (page 86). Note the unlike the style descent value (see [“Descent Attribute Tag”](#) (page 40)), the line descent value includes the leading value.

Line Direction Attribute Tag

The **line direction** attribute specifies a left-to-right or right-to-left direction for the glyphs associated with a text layout object, regardless of their natural direction as specified in the font. You can set and retrieve the line direction value using the following line and layout control attribute tag:

Attribute tag:	<code>kATSULineDirectionTag</code>
Data type:	<code>Boolean</code>
Default value:	Derived from the system script by calling the function <code>GetSysDirection</code>

The dominant direction of a line is the overall, controlling direction within which the individual glyph directions are set. If a Hebrew word is embedded in a line of Roman text, the dominant direction for that line is left to right, but the Hebrew word is still laid out right to left, as expected. Conversely, Roman text embedded in a line of Hebrew, in which the dominant direction is right to left, is still displayed left to right. For this reason, dominant direction has significance only in mixed-direction text.

The ATSUI text layout model accounts for dominant direction as well as glyph direction, automatically performing any reordering needed for correct display of simple mixed-direction lines of text.

Line Layout Options Attribute Tag

Line layout options allow for fine control over the layout process. You can set and retrieve line layout options using the following attribute tag:

Attribute tag: `kATSULineLayoutOptionsTag`

Data type: `ATSLineLayoutOptions`

Default value: `kATSUNoLayoutOptions`

The value for the line layout option attributes is specified by setting flags of type `ATSLineLayoutOptions`. These flags are described in Table 2-3. Note that some of the flags override style attributes set in the style objects associated with the text layout object.

Table 2-3 Line layout option flags

Flag	Description
<code>kATSLineHasNoHangers</code>	This value overrides any adjustment to hanging punctuation set for a style run inside the text layout object using the style attribute tag <code>kATSUForceHangingTag</code> or <code>kATSUHangingInhibitFactorTag</code> .
<code>kATSLineHasNoOpticalAlignment</code>	Indicates that optical alignment of characters at the text margin does not occur. This value overrides any adjustment to optical alignment set for a style run inside the text layout object using the style attribute tag <code>kATSUNoOpticalAlignmentTag</code> .
<code>kATSLineKeepSpacesOutOfMargin</code>	Indicates that whenever a space occurs at the end of a line that space is within the margin. (The default is to allow trailing white spaces inside the margin.)
<code>kATSLineNoSpecialJustification</code>	This value overrides the setting of postcompensation actions (such as ligature decomposition or kashida insertions) for a style run inside the text layout object using the style attribute tag <code>kATSUNoSpecialJustificationTag</code> .
<code>kATSLineLastNoJustification</code>	Indicates that the last line of a paragraph should not be justified.
<code>kATSLineFractDisable</code>	Specifies that the displayed line glyph positions will adjust for device metrics and integer origins.
<code>kATSLineImposeNoAngleForEnds</code>	Specifies that the carets at the ends of the line are perpendicular to the baseline.
<code>kATSLineFillOutToWidth</code>	Specifies that highlights for the line-end characters are extended from 0 to the specified line width.
<code>kATSLineTabAdjustEnabled</code>	Specifies that the tab character width at the end of a line is automatically adjusted to fit the specified line width.
<code>kATSLineIgnoreFontLeading</code>	Specifies that any leading value specified by a font or the ATSUI style attribute for leading is ignored.

Flag	Description
<code>kATSLineApplyAntiAliasing</code>	Specifies that Apple Type Services (ATS) produce anti-aliased glyph images despite system preferences or Quartz settings.
<code>kATSLineNoAntiAliasing</code>	Specifies that ATS turn-off anti-aliasing glyph imaging despite system preferences or Quartz settings (negates the <code>kATSLineApplyAntiAliasing</code> flag if set).
<code>kATSLineDisableNegativeJustification</code>	Specifies that if the line width is not sufficient to hold all the line's glyphs, glyph positions are allowed to extend beyond the line's assigned width so negative justification is not used.
<code>kATSLineDisableAutoAdjustDisplayPos</code>	Specifies that lines with any integer glyph positioning (such as that due to characters that are not anti-aliased or to the <code>kATSLineFractDisable</code> flag being set) do not automatically adjust individual character positions so they are rendered more aesthetically.
<code>kATSLineUseQDRendering</code>	Specifies that rendering uses QuickDraw. The default rendering in ATSUI in Mac OS X is to use Quartz in a 4-bit mode unless you specifically set up a Quartz (<code>CGContextRef</code>) context. In this case the default is to use a subpixel 8-bit mode.
<code>kATSLineDisableAllJustification</code>	Specifies that any justification operations are not run.
<code>kATSLineDisableAllGlyphMorphing</code>	Specifies that any glyph morphing operations are not run.
<code>kATSLineDisableAllKerningAdjustments</code>	Specifies that any kerning adjustment operations are not run.
<code>kATSLineDisableAllBaselineAdjustments</code>	Specifies that any baseline adjustment operations are not run.
<code>kATSLineDisableAllTrackingAdjustments</code>	Specifies that any tracking adjustment operations are not run.
<code>kATSLineDisableAllLayoutOperations</code>	A convenience constant that specifies to turn off all adjustments to justification, morphing, kerning, baseline, and tracking.
<code>kATSLineUseDeviceMetrics</code>	Specifies to optimize for onscreen display of text. Rounded device metrics are used instead of fractional path metrics (not anti-aliased).

Line Truncation Attribute Tag

The line truncation attribute specifies where truncation should occur and whether any negative justification should also be applied. You can set and retrieve the line truncation value using the following line and layout control attribute tag:

Attribute tag: `kATSULineTruncationTag`
 Data type: `ATSULineTruncation`
 Default value: `kATSUTruncateNone`

Quartz Context Attribute Tag

The Quartz context attribute specifies how ATSUI should render text. You can set and retrieve the Quartz context value using the following line and layout control attribute tag:

Attribute tag:	kATSUCGContextTag
Data type:	CGContextRef
Default value:	NULL
Comment:	This tag is available in Mac OS X. The default is for ATSUI to render text using Quartz at an anti-aliasing setting that simulates QuickDraw rendering. That is, a 4-bit pixel-aligned anti-aliasing.

You can use this tag to set up ATSUI to draw using Quartz in an 8-bit, subpixel rendering mode. Using this method of rendering, glyph origins are positioned on fractional points, resulting in superior rendering compared to the default 4-bit pixel-aligned rendering. For information on how to use the Quartz context tag to set up a rendering mode, see [“Drawing Text Using a Quartz Context”](#) (page 77).

Rotation Attribute Tag

The **rotation** attribute specifies the angle (in degrees) by which the entire line should be rotated. You can set and retrieve the rotation value using the following line and layout control attribute tag:

Attribute tag:	kATSULineRotationTag
Data type:	Fixed
Default value:	0

In ATSUI, rotation is counterclockwise. To produce vertical text, set the rotation value to -90.0 degrees and the value accessed by the style attribute tag `kATSUVerticalCharacterTag` to the constant `kATSUStronglyVertical`, as described in [“Glyph Orientation Attribute Tag”](#) (page 41).

The line shown on the right side of Figure 2-29 is rotated -90.0 degrees, but the glyphs have not been rotated. This type of rotation can be used to label the axis of a scientific graph.

Figure 2-29 A horizontal line and a line rotated -90 degrees

fishing fishing

Text Locator Attribute Tag

ATSUI uses text-locator information to determine where to break text and which options to apply to text breaks. The attribute value associated with this tag (`TextBreakLocatorRef`) contains the data needed by ATSUI to locate various kinds of text breaks. You can set and retrieve the text locator value using the following line and layout control attribute tag:

Attribute tag:	<code>kATSULineTextLocatorTag</code>
Data type:	<code>TextBreakLocatorRef</code> (an opaque structure defined in Unicode Utilities)
Default value:	<code>NULL</code>
Comment:	The default value specifies to use the region-derived locator or the default Text Utilities locator.

Width Attribute Tag

The width attribute specifies the desired width of a line of text, in typographic points, of the line when drawn as justified or right-aligned text. ATSUI treats vertical text as if it were horizontal. You can set and retrieve the width value using the following line and layout control attribute tag:

Attribute tag:	<code>kATSULineWidthTag</code>
Data type:	<code>ATSUTextMeasurement</code>
Default value:	<code>0</code>
Comment:	If you don't set the line width, you can't justify and align the text.

If a line is not justified or right-aligned, the value set by the width attribute tag `kATSULineWidthTag` is still used to apply negative justification (unless the `kATSLineDisableNegativeJustification` tag is set). **Negative justification** is the process of condensing text that exceeds the specified width so that the text fits the specified width. See [“Justification Attribute Tag”](#) (page 60) for additional information on how the width, alignment, and justification attributes interact.

If the line contains glyphs with large negative side bearings, hanging punctuation, or optically aligned edges, the final width of the displayed text may be different from the value specified by the width attribute. See [“Hanging Punctuation Attribute Tag ”](#) (page 43) and [“Optical Alignment Attribute Tag ”](#) (page 47) for more information.

If you use the width of the image bounding rectangle and the typographic bounding rectangle as a measure of the overall line length, the measurements may be slightly different from each other. See [“Text Measurements”](#) (page 19) for more information.

Basic Tasks: Working With Objects and Drawing Text

This chapter provides general guidelines for using ATSUI and provides step-by-step instructions for the basic tasks you can perform with ATSUI. These tasks are described in the following sections:

- [“Creating Style Objects and Setting Attributes”](#) (page 70) shows how to use ATSUI functions to create a style object and provides instructions for associating one or more style attributes with an object.
- [“Creating a Text Layout Object and Setting Attributes”](#) (page 73) describes how to create a text layout object, set layout attributes to control the text associated with the text layout object, and associate style objects with the text layout.
- [“Determining Paragraphs in a Unicode Text Block”](#) (page 74) provides an example of how to find paragraph separators in a block of text.
- [“Drawing Horizontal Text”](#) (page 76) shows how to use ATSUI to render horizontal text.
- [“Drawing Text Using a Quartz Context”](#) (page 77) details how to use a Quartz drawing context in conjunction with ATSUI.
- [“Drawing Equations”](#) (page 79) shows how to use style attributes to draw equations that contain subscripts and superscripts.
- [“Drawing Vertical Text”](#) (page 81) describes how to use ATSUI to render vertical text.
- [“Breaking Lines”](#) (page 83) shows how to set soft line breaks programmatically and how to use ATSUI’s batch line-breaking function.
- [“Measuring Text”](#) (page 85) provides information on the functions to use to obtain text measurements.
- [“Calculating Line Height”](#) (page 86) discusses the preferred method for setting line height in Mac OS X version 10.2 and describes what to do if your application runs in an earlier version of the Mac OS.
- [“Flowing Text Around a Graphic”](#) (page 88) shows the preferred way to draw text so that it flows around an image.
- [“Setting Up a Tab Ruler”](#) (page 96) provides step-by-step instructions for setting up a tab ruler for a text layout object.

Before you read this chapter, you should be familiar with the concepts discussed in [“Typography Concepts”](#) (page 15) and [“ATSUI Style and Text Layout Objects”](#) (page 37), [“ATSUI Style and Text Layout Objects”](#) (page 37).

This chapter provides a number of code samples to illustrate how to use ATSUI. You can obtain the sample applications from which this code is taken, as well as additional code samples, from the developer sample code website:

<http://developer.apple.com/samplecode/>

Guidelines for Using ATSUI

There are a number of guidelines you should follow to assure optimal performance and efficient memory use when you use ATSUI. This section summarizes them. The sample code in this and the other task chapters follows these guidelines.

- Once you have created a style object, keep it until you no longer need it. That is, do not dispose of a style object and then recreate it each time you need to draw using that style. See [“Creating Style Objects and Setting Attributes”](#) (page 70).
- Associate a text layout object with a paragraph of text. If you associate a text layout object with a smaller unit of text such as a word, line, or style run, ATSUI won’t be able to lay out text efficiently. In addition, text may not be laid out properly. Using a unit of text that is longer than one paragraph may cause extra coding on your part if you want to treat the last line of a paragraph separately from the other lines (for example, justifying everything except the last line in a paragraph). See [“Creating a Text Layout Object and Setting Attributes”](#) (page 73).
- Keep a text layout object around even when the text associated with it is altered. Call the functions `ATSUSetTextPointerLocation`, `ATSUTextDeleted`, or `ATSUTextInserted` to manage the altered text.
- Use the QuickDraw functions `QDBeginCGContext` and `QDEndCGContext` to do Quartz drawing instead of the function `CreateCGContextForPort`. See [“Drawing Text Using a Quartz Context”](#) (page 77).
- Use the same Quartz context (`CGContext`) until all drawing is completed. Don’t create and destroy the context each time you draw unless it is absolutely necessary. See [“Drawing Text Using a Quartz Context”](#) (page 77).
- Call the function `ATSUGetGlyphBounds` to get the ascent and descent of a line and the typographic bounds after layout. Don’t call `ATSUGetUnjustifiedBounds` for these measurements because that function might need to perform a separate layout. See [“Measuring Text”](#) (page 85).
- Set soft line breaks programmatically rather than manually. You can set line breaks by calling the function `ATSUBreakLine` with the `iUseAsSoftLineBreak` parameter. See [“Breaking Lines”](#) (page 83).
- When you have text that must flow around a graphic, use one text layout object per paragraph. Flow the text around the graphic by varying the line width for each line appropriately rather than using multiple layouts to flow the text around a graphic. See [“Flowing Text Around a Graphic”](#) (page 88).
- Use font fallback objects (`ATSUFontFallbacks`) instead of the global font fallbacks, as global font fallbacks will be deprecated. A font fallback object works with a specific text layout object. See [“Using Font Fallback Objects”](#) (page 103).
- Keep font fallback objects (`ATSUFontFallbacks`) around as long as possible, and share them between text layout objects. See [“Using Font Fallback Objects”](#) (page 103).
- Create a standard Carbon Font menu or build a Font menu or list when your application first starts up. Get the font selection from the menu instead of calling the function `ATSUFindFontFromName` to find a font.

Creating Style Objects and Setting Attributes

ATSUI style objects are opaque objects that represent a collection of stylistic attributes. Each attribute is defined by three values (a triple):

- an attribute tag
- the size of the attribute value associated with the tag
- the value for the attribute specified by the tag

To create an ATSUI style object, you must do the following:

1. Declare storage for the style.

For example:

```
ATSUStyle      gDefaultStyle;
```

2. Create the style object.

You can use the function `ATSUCreateStyle`. For example:

```
OSStatus      status = noErr;

status = ATSUCreateStyle (&gDefaultStyle);
```

3. Set up a triple (tag, size, value) for each attribute associated with the style.

Typically, you declare three parallel arrays, one for attribute tags, one for sizes, and the third array for the value of the attribute. ATSUI supplies constants that specify attribute tags. For example, the following code sets up two attributes—a font size (`kATSUSizeTag`) and style (`kATSUQDBoldfaceTag`):

```
ATSUAttributeTag  theTags[] = {kATSUSizeTag, kATSUQDBoldfaceTag};
ByteCount        theSizes[] = {sizeof(Fixed), sizeof(Boolean)};

Fixed    atsuSize = Long2Fix (12);
Boolean  isBold = TRUE;
ATSUAttributeValuePtr theValues[] = {&atsuSize, &isBold};
```

See [“Style Objects”](#) (page 37) for more information on the attribute tags that are available in ATSUI.

4. Associate the attributes with the style object.

You can use the function `ATSUSetAttributes`. You pass in the number of attributes you want to set and the three parallel arrays that represent the attribute tags, sizes of the values, and the values themselves, as shown in the following code:

```
status = ATSUSetAttributes (gDefaultStyle,
                           2,
                           theTags,
                           theSizes,
                           theValues);
```

Note that a style object is not associated with any text; it simply specifies a collection of attributes. You can set up style objects for your application once, and then reuse them over and over as needed. For example, you could set up style objects to define such styles as emphasis, heading level, superscript, subscript, and bold.

Tip: Once you have created an ATSUI style object, keep it until you no longer need it. You can use it again and again, whenever your application needs to draw some text using the style defined by that object.

You can create a style object by copying another style object using the function `ATSUCreateAndCopyStyle`. Copying a style object is useful if you want to make a set of related styles. For example, if you have already created a style object to define a default font, font size, and font color, you could make an italic style based on the default by calling the following function:

```
ATSUStyle      italicStyle;
ATSUCreateAndCopyStyle (defaultStyle, &italicStyle);
```

Then you would set up a triple for the italic attribute:

```
ATSUAttributeTag  theTags[] = {kATSUODItalicTag};
ByteCount         theSizes[] = {sizeof(Boolean)};
Boolean           isItalic = TRUE;
```

```
ATSUAttributeValuePtr theValues[] = {&isItalic};
```

Finally, you associate the italic attribute with the `italicStyle` object:

```
status = ATSUSetAttributes (gItalicStyle,
                           1,
                           theTags,
                           theSizes,
                           theValues);
```

By using the function `ATSUCreateAndCopyStyle`, the `italicStyle` object inherits the attributes of the `defaultStyle` object and has any other attributes you set by calling the function `ATSUSetAttributes`.

When you are done using a style object, you must call the function `ATSUDisposeStyle` to dispose of it. For example, to dispose of the `italicStyle` object, you'd use the following line of code:

```
ATSUDisposeStyle (italicStyle);
```

Remember, it's best to reuse styles rather than to create and dispose of a style each time you need them in your application. You can create all the common styles you need when your application starts up, and dispose of them when your application quits. Listing 3-1 shows code that creates styles prior to the application event loop and disposes of styles when the application event loop terminates.

Listing 3-1 Code that keeps style objects around until the application quits

```
MyMakeStyles();
RunApplicationEventLoop();
MyDisposeStyles();
```

Once you have created a style object, you can pass it to any ATSUI function that takes a style object as a parameter, such as the function `ATSUSetRunStyle`. You can pass an array of style objects to such functions as `ATSUCreateTextLayoutWithTextPtr` to define the attributes for style runs in a text layout object.

Creating a Text Layout Object and Setting Attributes

ATSUI text layout objects are opaque objects that contain a pointer or handle to a block of Unicode text, style run information, and line and layout controls for the block of text.

To create an ATSUI text layout object, you must do the following:

1. Declare storage for the text layout.

For example:

```
ATSUITextLayout myTextLayout;
```

2. Create the text layout object by calling the functions `ATSUCreateTextLayout` or `ATSUCreateTextLayoutWithTextPtr`.

For example, the following code creates a text layout object (`myTextLayout`) for an entire text buffer (`myTextBlock`):

```
OSStatus      status = noErr;
UniCharCount length = kATSUToTextEnd;

status = ATSUCreateTextLayoutWithTextPtr ((UniChar*) myTextBlock,
                                          kATSUFromTextBeginning, // offset from beginning
                                          kATSUToTextEnd,           // length of text range
                                          myTextBlockLength,        // length of text buffer
                                          1,                        // number of style runs
                                          &length,                 // length of the style run
                                          &myArrayOfStyleObjects,
                                          &myTextLayout);
```

On output, `myTextLayout` refers to the newly created text layout object. Note that you own the text in the buffer you pass to the function `ATSUCreateTextLayoutWithTextPtr`.

Each ATSUI text layout can have no more than 64,000 different styles.

3. Set up a triple (tag, size, value) for each line and layout attribute associated with the text layout.

Typically, you declare three parallel arrays, one for attribute tags, one for sizes, and the third array for the value of the attribute. ATSUI supplies constants that specify attribute tags. For example, the following code sets up two attributes—flushness (`kATSULineFlushFactorTag`) and justification (`kATSULineJustificationFactorTag`):

```
ATSUIAttributeTag theTags[] = {kATSULineFlushFactorTag,
                               kATSULineJustificationFactorTag};
ByteCount theSizes[] = {sizeof(Fract), sizeof(Fract)};
Fract myFlushFactor = kATSUStartAlignment;
Fract myJustFactor = kATSUFullJustification;

ATSUIAttributeValuePtr theValues[] = {&myFlushFactor, &myJustFactor};
```

See “[Line and Layout Attributes](#)” (page 57) for more information on the attribute tags available in ATSUI.

4. Associate the line and layout attributes with the text layout object by calling the function `ATSUSetLayoutControls`.

For example:

```
status = ATSUISetLayoutControls (myTextLayout,
                                2,
                                theTags,
                                theSizes,
                                theValues);
```

You can also create a text layout object by calling the function `ATSUCreateAndCopyTextLayout`. This function is useful if you want all text layout objects in your application to use the same line and layout attributes. After you have created the new text layout object, you can call the function `ATSUISetTextPointerLocation` to associate the new text layout object with the appropriate block of text.

Determining Paragraphs in a Unicode Text Block

This section shows you how to find a paragraph within a block of Unicode text. Although determining paragraphs is not a task that uses ATSUI functions, breaking a block of text into paragraph-sized chunks is important if you want to use ATSUI efficiently. Once the text is broken into paragraphs, you can create a text layout object for each paragraph.

Tip: To get the best performance from ATSUI, a text layout object should be associated with a paragraph of text.

To find a paragraph, you can write code that performs steps similar to the following:

1. Define Unicode characters that act as paragraph separators. Exactly what you choose to define as a paragraph separator depends on the needs of your specific application. For example, these five character sequences can be used to separate paragraphs:
 - ASCII newline '\n'
 - ASCII return '\r'
 - ASCII return followed by ASCII newline
 - Unicode line separator
 - Unicode paragraph separator
2. Iterate through the block of text until you find a paragraph separator.

The code in Listing 3-2 shows a `MyFindParagraph` function that takes a block of Unicode text, the total length of the text block, and an offset into the text. The function finds the next paragraph separator and returns whether or not the function reached the end of the text block. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-2 A function to find a paragraph in a block of Unicode text

```
Boolean MyFindParagraph (UniChar *theText,
                        UniCharCount theTextLength,
                        UniCharArrayOffset paragraphStart,
```

```

        UniCharArrayOffset *paragraphEnd)
{
    UniChar          CR   = 0x000D;           // 1
    UniChar          LF   = 0x000A;           // 2
    UniChar          LSEP = 0x2028;           // 3
    UniChar          PSEP = 0x2029;           // 4
    UniChar          NL   = 0x0085;           // 5
    UniCharCount     currentPosition;
    UniChar          currentChar;
    Boolean          endOfText = false;

    if (theText == NULL)                       // 6
    {
        *paragraphEnd = 0;
        return true;
    }

    for (currentPosition=paragraphStart; (currentPosition < theTextLength);
        currentPosition++)                       // 7
    {
        currentChar = theText[currentPosition];

        if ( (currentChar == PSEP) || (currentChar == LSEP) ||
            (currentChar == LF) || (currentChar == NL))
        {
            break;
        }
        if ( currentChar == CR )
        {
            if ( currentPosition < (theTextLength - 1) )
            {
                if ( theText[currentPosition + 1] == LF )           // 8
                    currentPosition++;
            }
            break;
        }
    }

    if (currentPosition == theTextLength)
        currentPosition--;                       // 9
    if (currentPosition == (theTextLength - 1))
        endOfText = true;                       // 10

    *paragraphEnd = currentPosition + 1;         // 11
    return endOfText;                           // 12
}

```

Here's what the code does:

1. Declares a variable to represent the ASCII return character.
2. Declares a variable to represent the ASCII newline character.
3. Declares a variable to represent the Unicode line separator character.
4. Declares a variable to represent the Unicode paragraph separator character.

5. Declares a variable to represent the Unicode next line character that is a C1 control used by XML for line breaks.
6. Checks to see if there is text to process. If the pointer is `NULL`, returns from the function.
7. Iterates through the block of text until it finds a character sequence that indicates the end of a paragraph.
8. Treats the ASCII new line and ASCII return combination as if it were a single entity, by advancing through the text buffer accordingly.
9. Checks for the special case of reaching the end of the text without finding a paragraph. In this case, the current position must be decremented by one.
10. Checks to see if it reached the end of the text buffer.
11. Returns the position just after the end of the paragraph. This position is returned because an ATSUI-style offset is between array elements (an edge offset). See [“Typography Concepts”](#) (page 15) for information on edge offsets in ATSUI.
12. Returns a Boolean value to indicate whether the end of text was reached (`true`) or not (`false`).

Drawing Horizontal Text

Drawing text is easy once you have created a text layout object and associated text with the object. You call the function `ATSUDrawText`, as shown in the following code:

```
ATSUDrawText (myTextLayout,
              kATSUFromTextBeginning,
              kATSUToTextEnd,
              myXLocation,
              myYLocation);
```

The function `ATSUDrawText` take five parameters:

- A text layout object. See [“Creating a Text Layout Object and Setting Attributes”](#) (page 73) for information on how to create a text layout object.
- The offset from the beginning of the text to the first character that should be rendered.
- The length of the text range to render. The constant `kATSUToTextEnd` specifies to render to the end of the text buffer.
- The x-coordinate of the origin at which to render the text. You specify `QuickDraw` or `Quartz 2D` coordinates, depending on whether you are using a `QuickDraw` port or a `Quartz` graphics context (`CGContext`). Coordinates must be specified as `Fixed` values.
- The y-coordinate of the origin at which to render the text.

Note: If you want to draw highlighted text, see “[Highlighting Selected Text](#)” (page 101).

Drawing Text Using a Quartz Context

With Mac OS X version 10.2, ATSUI renders text through Quartz, even if you do not attach a `CGContext` to a text layout object. In this default case, ATSUI retrieves the internal canonical `CGContext` of the current port, and renders to that port using Quartz at an anti-aliasing setting that simulates QuickDraw rendering. That is, a 4-bit pixel-aligned anti-aliasing. With this method of rendering, the origins of the glyphs always fall on integer positions and can lead to less-than-ideal glyph placements even after ATSUI makes fine adjustments to the integer positioning. This section shows you how to set up ATSUI to instead use an 8-bit, subpixel rendering. Using this method of rendering, glyph origins are positioned on fractional points, resulting in superior rendering compared to ATSUI's default 4-bit pixel-aligned rendering.

To set up ATSUI to use an 8-bit, subpixel rendering through Quartz, you must perform the following tasks:

1. Set up a Quartz context (`CGContext`) for your application by using the QuickDraw function `QDBeginCGContext`. When you are done using the `CGContext`, you must call the function `QDEndCGContext`.

You need to call the function `QDBeginCGContext` only once in your application, as you should use the same `CGContext` until all drawing is completed.

2. Set the Quartz context as a layout attribute for the text layout object whose text you want to draw, using the tag `kATSUCGContextTag`, and by calling the function `ATSUSetLayoutControls`.

For example, if you already set up a Quartz context named `myCGContext`, you use the following code to set the Quartz context as an attribute of a text layout object (`myTextLayout`) you created previously:

```
ATSUIAttributeTag    theTags[0] = kATSUCGContextTag;
ByteCount           theSizes[0] = sizeof (CGContextRef);
ATSUIAttributeValuePtr theValues[] = &myCGContext;

ATSUSetLayoutControls (myTextLayout,
                      1,
                      theTags,
                      theSizes,
                      theValues);
```

When you use Quartz with ATSUI, you can use all the effects available through Quartz 2D. You can rotate the Quartz context to achieve a number of effects, such as the angled text as shown in Figure 3-1 and the rotated text shown in Figure 3-2.

Listing 3-3 shows the code necessary to draw rotated text using Quartz 2D. First, you call the Quartz 2D function `CGContextRotateCTM` to rotate the Quartz context by a specified angle. Then, when you call the function `ATSUDrawText`, the text is drawn into the rotated context. The text appears on the screen when you call the function `CGContextFlush`.

Figure 3-1 Text drawn at an angle

The image shows a line of Chinese text rotated approximately 45 degrees counter-clockwise. The text is: 子曰：「學而時習之，不亦說乎？有朋自遠方來，不亦樂乎？人不知而不愠，不亦君子乎？」

Listing 3-3 Using a Quartz context to rotate text

```
CGContextRotateCTM (myCGContext, myAngle);
ATSUDrawText (myTextLayout,
               kATSUFromTextBeginning,
               kATSUToTextEnd,
               myXLocation,
               myXLocation);
CGContextFlush (myCGContext);
```

Tip: You can draw more than once into a Quartz context without calling the function `CGContextFlush`. This is the most optimal way to do batch drawing using a Quartz context.

Figure 3-2 Rotated text



Drawing Equations

Drawing scientific and mathematical equations often requires drawing subscripts and superscripts, such as those shown in Figure 3-3. You can draw characters as subscripts or a superscripts by applying the tag `kATSUCrossStreamShiftTag` to the characters. For horizontal text, a positive cross-stream value shifts a glyph upwards, whereas a negative cross-stream value shifts a glyph downwards. You also need to adjust the font size of the subscripts and superscripts appropriately, as these characters are usually drawn with a smaller font size than that used for the main characters of the equation.

Figure 3-3 A scientific equation drawn by adjusting cross-stream shift values

$$m = m_0 e^{-\lambda t}$$

You can create styles for subscripts and superscripts when your application launches, and use the styles whenever you need to draw an equation. You need to perform the following steps to set up subscript and superscript styles:

1. Allocate space for the ATSUI style objects.

The subscript and superscript styles are based on a default style object.

```
ATSUIStyle      myDefaultStyle,
                mySubscriptStyle,
                mySuperscriptStyle;
```

2. Create and set attributes for the default style.

As for any style, you must set up a triple (tag, size, value) for each attribute and call the function `ATSUISetAttributes` to apply the attributes to the style.

```
ATSUICreateStyle (&myDefaultStyle);
ATSUIAttributeTag theTags[] = {kATSUISizeTag, kATSUIQDItalicTag};

ByteCount theSizes[] = {sizeof (Fixed), sizeof (Boolean)};

Fixed      myFontSize = Long2Fix (myDefaultSize);
Boolean    isItalic = FALSE;

ATSUIAttributeValuePtr theValues[] = {&myFontSize, &isItalic};

status = ATSUISetAttributes (gDefaultStyle, 2,
                            theTags, theSizes, theValues);
```

3. Create and set attributes for the subscript style.

```
ATSUICreateAndCopyStyle (myDefaultStyle, &mySubScriptStyle);
ATSUIAttributeTag theTags[] = { kATSUISizeTag,
                               kATSUICrossStreamShiftTag};
ByteCount theSizes[] = {sizeof(Fixed),
                        sizeof(Fixed)};
Fixed     mySubScriptSize = Long2Fix (myDefaultSize - 2);
Fixed     mySubScriptShift = Long2Fix (-6);

ATSUIAttributeValuePtr theValues[] = {&mySubScriptSize,
                                       &mySubScriptShift};

status = ATSUISetAttributes (mySubScriptStyle, 2,
                            theTags, theSizes, theValues);
```

4. Create and set attributes for the superscript style.

```
ATSUICreateAndCopyStyle (myDefaultStyle, &mySuperScriptStyle);
ATSUIAttributeTag theTags[] = { kATSUISizeTag,
                               kATSUICrossStreamShiftTag};
ByteCount theSizes[] = {sizeof(Fixed), sizeof(Fixed)};

Fixed     mySuperScriptSize = Long2Fix (myDefaultSize - 2);
Fixed     mySuperScriptShift = Long2Fix (6);

ATSUIAttributeValuePtr theValues[] = {&mySuperScriptSize,
                                       &mySuperScriptShift};

status = ATSUISetAttributes (mySuperScriptStyle, 2,
                            theTags, theSizes, theValues);
```


You should keep these styles around as long as you need them. Style objects are independent of text; you can use them over and over.

You can apply the subscript and superscript styles to the appropriate characters in an equation by following these steps:

1. Create a text layout object for the equation text.

Use the default style on the entire text string to create the text layout object.

```
length = sizeof(myPhysicsEquation)/sizeof(UniChar);
status = ATSUCreateTextLayoutWithTextPtr ((UniChar*) myPhysicsEquation,
    0,
    length,
    length,
    1,
    &length,
    &myDefaultStyle,
    &myEquationLayout);
```

2. You may need to set up transient font matching to assure that the glyphs in the equation are rendered appropriately.

```
status = ATSUSetTransientFontMatching (myEquationLayout, true);
```

3. Set the subscript and superscript styles.

You apply the subscript and superscript styles to the appropriate characters by calling the function `ATSUSetRunStyle`. You must specify the starting character to which the style applies and specify the number of characters in the style run.

```
status = ATSUSetRunStyle (myEquationLayout,
    mySubScriptStyle,
    myStartofSubScript,
    myCharsInSubScript);
status = ATSUSetRunStyle (myEquationLayout,
    mySuperScriptStyle,
    myStartofSuperScript,
    myCharsInSubScript);
```

4. Draw the equation.

```
ATSUDrawText (myEquationLayout,
    kATSUFromTextBeginning,
    sizeof(myPhysicsEquation)/sizeof(UniChar),
    Long2Fix (myXPenLocation),
    Long2Fix (myYPenLocation));
```

Drawing Vertical Text

If you want to draw text vertically—for example Chinese, Japanese, or Korean text—you must set up ATSUI to use vertical forms of the glyphs and rotate the line. The vertical form of a glyph is a style attribute; whereas line rotation is a line and layout attribute. If you simply rotate a line without also setting up ATSUI to use

vertical forms of the glyphs, you get a result similar to that shown in Figure 3-4. If you use vertical forms of the glyphs and then rotate the line, you get results similar to that shown in Figure 3-5. Compare the glyphs in the center of the third column in Figure 3-5 with Figure 3-4 to see the effect of rotating both the glyphs and the line.

Figure 3-4 A rotated line of text that does not use vertical forms of the glyphs

Figure 3-5 A rotated line of text that uses vertical forms of the glyphs

To draw vertical text (rotated line and rotated glyphs), perform the following steps:

1. Create a style object and set it to use vertical forms.

You can base this style object on a default style object that you've already created for your application.

As for any style, you must set up a triple (tag, size, value) for each attribute and call the function `ATSUSetAttributes` to apply the attributes to the style.

```
ATSUCreateAndCopyStyle(myDefaultStyle, &myVerticalStyle);
ATSUVerticalCharacterType verticalType = kATSUStronglyVertical;
theTags[0] = kATSUVerticalCharacterTag;
theSizes[0] = sizeof (ATSUVerticalCharacterType);
theValues[0] = &verticalType;

ATSUSetAttributes (myVerticalStyle, 1,
                  theTags, theSizes, theValues);
```

2. Rotate the line by setting line rotation as a layout attribute.

As for any line or layout attribute, you must set up a triple (tag, size, value) for each attribute and call the function `ATSUSetLayoutControls` to apply the attributes to the text layout object.

Note: It is also possible to rotate the line by setting up a Quartz context and then rotating it. However, you should use `ATSUI` line rotation if you need to perform hit-testing on the text.

```
ATSUCreateTextLayoutWithTextPtr(
    (UniChar *) myTextString,
    myOffset,
    length,
    totalLength,
    styleRunCount,
    &length,
    &myVerticalStyle,
    &myTextLayout);
Fixed myAngleToRotateText = FloatToFixed (-90.0); // degrees
theTags[0] = kATSULineRotationTag;
theSizes[0] = sizeof (Fixed);
theValues[0] = &myAngleToRotateText;
ATSUSetLayoutControls (myTextLayout, 1,
                      theTags, theSizes, theValuePtrs);
```

3. Draw the text.

```
ATSUDrawText (myTextLayout,
             kATSUFromTextBeginning,
             sizeof(myTextString)/sizeof(UniChar),
             Long2Fix (myXPenLocation),
             Long2Fix (myYPenLocation));
```

Breaking Lines

`ATSUI` provides two functions for calculating and setting soft line breaks programmatically: `ATSUBreakLine` and `ATSUBatchBreakLines`. Calling the function `ATSUBatchBreakLines` is equivalent to repeatedly calling the function `ATSUBreakLine`, as shown in Listing 3-4. It's preferable that you use `ATSUBatchBreakLines` because this function performs more efficiently than repeated calls to `ATSUBreakLine`.

The code fragment in Listing 3-4 compares batch line breaking to calculating breaks on a line-by-line basis. A detailed explanation for each numbered line of code appears following the listing. If you've used the `ATSUBreakLine` function before, you'll see that replacing it with `ATSUBatchBreakLines` reduces your code by a few lines and improves the performance of your application.

Listing 3-4 A code fragment that performs line breaking

```

ATSUTextLayout      myTextLayout;
Fixed               myLineBreakWidth;
ItemCount           myNumSoftBreaks;
UniCharCount       myTextLength;
UniCharArrayOffset *mySoftBreaks,
                   myStartingOffset,
                   myCurrentStart,
                   myCurrentEnd;

/* Insert your code to set up the text layout object

#if USE_BATCHBREAKLINES                                     // 1
    ATSUBatchBreakLines (myTextLayout,
                        myStartingOffset,
                        myTextLength,
                        myLineBreakWidth,
                        &myNumSoftBreaks);                 // 2
#else
    myCurrentStart = myStartingOffset;                       // 3
    myCurrentEnd = myTextLength;
    do
    {
        status = ATSUBreakLine (myTextLayout,
                                myCurrentStart,
                                myLineBreakWidth,
                                true,
                                &myCurrentEnd);             // 4
        myCurrentStart = myCurrentEnd;
    } while (myCurrentEnd < myTextLength);
#endif

ATSUGetSoftLineBreaks (myTextLayout,
                       kATSUFromTextBeginning,
                       kATSUToTextEnd,
                       0, NULL, &myNumSoftBreaks);         // 5
mySoftBreaks = (UniCharArrayOffset *) malloc(myNumSoftBreaks *
                                             sizeof(UniCharArrayOffset)); // 6
ATSUGetSoftLineBreaks (myTextLayout,
                       kATSUFromTextBeginning,
                       kATSUToTextEnd,
                       myNumSoftBreaks, mySoftBreaks, &myNumSoftBreaks); // 7

// Insert your code here to loop over all the soft breaks and draw them

free (mySoftBreaks);                                       // 8

```

Here's what the code does:

1. Checks to see if batch line breaking should be used. This code is here only to illustrate the difference between batch line breaking and using the older function `ATSUBreakLine`.

2. Calls the function `ATSUBatchBreakLines`. You must supply the text layout object that is associated with the text you want to process. You must also supply a starting offset, the length of the text, and a line width. ATSUI returns the number of soft breaks that are calculated.
3. If batch line breaking isn't used, sets up variables for the starting and ending offsets of the text for which you want to calculate line breaks.
4. Calls the function `ATSUBreakLine` to calculate line breaks. The parameter `iUseAsSoftLineBreak` is set to `true` to indicate that ATSUI should automatically set the line break to the value returned by the `oLineBreak` parameter (`myCurrentEnd`). You must also provide as parameters the first character of the text range associated with the text layout object and the line width.
5. Calls the function `ATSUGetSoftLineBreaks` to obtain the number of soft line breaks calculated by ATSUI. This is a function you typically call twice. The first time, pass `NULL` for the `oBreaks` parameter to obtain the number of line breaks. Then, allocate memory for the line break array and call the function again to obtain the array, as shown in the next two steps.
6. Allocates the appropriate amount of memory for the line break array.
7. Calls the function `ATSUGetSoftLineBreaks` a second time, but this time passes an array of the appropriate size. On output, the array contains offsets from the beginning of the text buffer to each of the soft line breaks in the text range.
8. Frees the previously allocated memory.

Although it is possible for you to set soft line breaks instead of letting ATSUI do it for you (by passing `false` for the parameter `iUseAsSoftLineBreak`), you shouldn't do so unless it is absolutely necessary. See “[Flowing Text Around a Graphic](#)” (page 88) for an example of using the function `ATSUBreakLine` to set line breaks.

Tip: You get better performance from ATSUI when you set soft line breaks programmatically.

Measuring Text

The trick with measuring text is to choose the appropriate function to obtain text measurements. In most cases, an application needs to obtain the typographic bounds of a line of text after final layout. After-layout typographic bounds take into account rotation and other layout attributes that have been applied to the line. If these are the measurements you need, call the function `ATSUGetGlyphBounds`. See “[Flowing Text Around a Graphic](#)” (page 88) for an example of how the function `ATSUGetGlyphBounds` is used to get ascent and descent values. The obtained values are then used to set pen location prior to drawing a line of text.

In rare cases, an application may need to obtain the typographic bounds of a line of text prior to final layout. For example, before-justification/alignment typographic bounds can be used when you need to determine your own line breaks or the leading and line space to impose on a line. Before-justification/alignment typographic bounds ignore any previously-set line attributes such as line rotation, alignment, justification, ascent, and descent. If you need before-justification/alignment measurements, call the function `ATSUGetUnjustifiedBounds`.

For more information on typographic bounds, see “[Text Measurements](#)” (page 19). See *Inside Mac OS X: ATSUI Reference* for documentation on the functions `ATSUGetGlyphBounds` and `ATSUGetUnjustifiedBounds`.

Calculating Line Height

Your application needs to space lines appropriately when it draws text. When you calculate line spacing, you should use the ascent, descent, and leading values. (Line height = ascent + descent + leading) The line/layout descent value combines the font's descent and leading values, as both values refer to measurements below the baseline.

You can use either of these two functions to determine line height:

- `ATSUGetLineControl`. You can use this function to obtain line ascent and descent values in Mac OS X version 10.2 and later, even if the values were not explicitly set. The code fragment in Listing 3-5 demonstrates how to obtain line height using the function `ATSUGetLineControl`.
- `ATSUGetGlyphBounds`. You can use this function to obtain the trapezoid that represents the typographic bounds of a line after the final layout. You must use this function if your application runs in CarbonLib and Mac OS 8 and 9. The code fragment in Listing 3-6 (page 87) demonstrates how to obtain line height using the function `ATSUGetGlyphBounds`.

Note: Although the function `ATSUGetUnjustifiedBounds` (formerly named `ATSUMeasureText`) returns the typographical ascent and descent values, using this function to obtain ascent and descent values might degrade performance. You should not use `ATSUGetUnjustifiedBounds` to obtain these values.

A detailed explanation for each numbered line of code appears following the listing.

Listing 3-5 Calculating line height in Mac OS X version 10.2

```

ATSUTextLayout      myTextLayout;
ATSUTextMeasurement myAscent, myDescent;
ByteCount           actualSize;
UniCharArrayOffset  myStartingOffset;                                // 1

// Your code to set up the text layout object and associate it with with
// text

ATSUGetLineControl (myTextLayout,
                    myStartingOffset,
                    kATSULineAscentTag,
                    sizeof (ATSUTextMeasurement),
                    &myAscent,
                    &actualSize);                                // 2

ATSUGetLineControl (myTextLayout,
                    myStartingOffset,
                    kATSULineDescentTag,
                    sizeof (ATSUTextMeasurement),
                    &myDescent,
                    &actualSize);                                // 3

```

Here's what the code does:

1. Declares the variable `myStartingOffset`. The value of this variable should be the offset that corresponds to the beginning of the line you want to measure. You can obtain the starting offset by calling the function `ATSUGetSoftLineBreaks` to obtain all the soft line breaks set for the text you want to render. The offset for each soft line break also denotes the starting offset for a line. See “[Breaking Lines](#)” (page 83) for information on getting soft line breaks.
2. Calls the function `ATSUGetLineControl` to obtain the ascent for the line whose starting offset is specified by the `myStartingOffset` value. You must pass the `kATSULineAscentTag` attribute tag to specify that you want to obtain the line ascent. On output, the `myAscent` parameter contains the line ascent value, and the `actualSize` parameter points to the size in bytes of the attribute value.
3. Calls the function `ATSUGetLineControl` to obtain the descent for the line whose starting offset is specified by the `myStartingOffset` value. You must pass the `kATSULineDescentTag` attribute tag to specify that you want to obtain the line descent. On output, the `myDescent` parameter contains a value that is the line descent plus the leading, and the `actualSize` parameter points to the size in bytes of the attribute value.

Note: The attribute tags `kATSUAscentTag` and `kATSUDescentTag` can be used to obtain the ascent and descent values associated with an `ATSUStyle` object. In this case, you’d call the function `ATSUGetAttribute`. However, when you use these tags to obtain style attributes, the leading value is not included as part of the descent. To obtain the leading value for an `ATSUStyle` object, you must use the attribute tag `kATSULeadingTag`.

Listing 3-6 Calculating line height in Mac OS 8, Mac OS 9, and CarbonLib

```

ATSUTextLayout          myTextLayout;
ATSUTextMeasurement     myAscent, myDescent;
UniCharArrayOffset      myStartingOffset;           // 1
UniCharCount            myLineLength               // 2

ATSTrapezoid           theBounds;
ItemCount               numBounds;

// Your code to set up the text layout object and associate it with text

ATSUGetGlyphBounds (myTextLayout,
                    0,
                    0,
                    myStartingOffset,
                    myLineLength,
                    kATSUUseFractionalOrigins,
                    1,
                    &theBounds,
                    &numBounds);                  // 3
myAscent = - theBounds.upperLeft.y;                // 4
myDescent = theBounds.lowerLeft.y;                 // 5

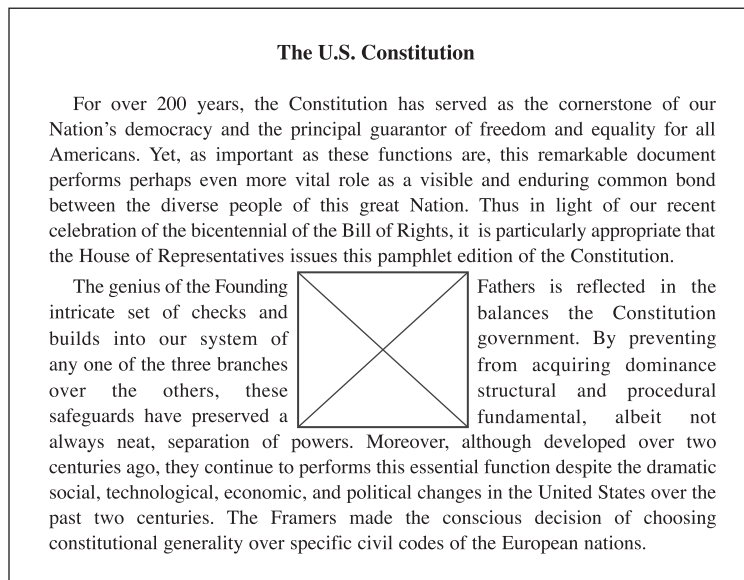
```

Here’s what the code does:

1. Declares the variable `myStartingOffset`. The value of this variable should be the offset that corresponds to the beginning of the line you want to measure. You can obtain the starting offset by calling the function `ATSUGetSoftLineBreaks` to obtain all the soft line breaks set for the text you want to render. The offset for each soft line break also denotes the starting offset for a line. See [“Breaking Lines”](#) (page 83) for information on getting soft line breaks.
2. Declares the variable `myLineLength`. The value of this variable should be the length of the line you want to measure. You can obtain the line length by subtracting the starting offset for the line you want to measure from the starting offset for the next line in the text layout.
3. Calls the function `ATSUGetGlyphBounds` to obtain the typographic bounds of a line of glyphs after the final layout. On output, `theBounds` points to an `ATSTrapezoid` structure that contains the line ascent and descent values. When you call this function on an entire line, only one trapezoid is returned.
4. Extracts the ascent value from the trapezoid returned by the function `ATSUGetGlyphBounds`.
5. Extracts the descent value from the trapezoid returned by the function `ATSUGetGlyphBounds`. The `theBounds.lowerLeft.y` value is the sum of the descent and the leading values.

Flowing Text Around a Graphic

This section shows you how to embed a graphic in text by flowing text around it, as shown in Figure 3-6. To flow text around a graphic, you use one text layout object per paragraph, then vary the width of the lines in the text layout object so the text flows around the graphic appropriately.

Figure 3-6 Text flowed around a graphic

The following steps outline what you must do for each paragraph of text for which you want to flow text around a graphic. The steps assume that you have already set up a text layout object for each paragraph of text and that you obtained the coordinates of the image around which you need to flow text.

1. Determine the current line breaking width. It should be the full line width, the line width for lines on the left side of the image, or the line width for lines on the right side of the image.
2. Set the line width as part of the line attributes for the text layout object.
3. Measure the text to get the ascent and descent of the laid-out line. You need the ascent and the descent to calculate the y-coordinate of the pen position.
4. Set the x and y coordinates of the pen position. If you are drawing a full-width line or a line that is on the left side of the image, you should use the left-side margin value for the x-coordinate.

If you are drawing a line that is on the right side of the image, you should use the value calculated for the x-coordinate of the right side of the image plus any whitespace you want between the image and the text.

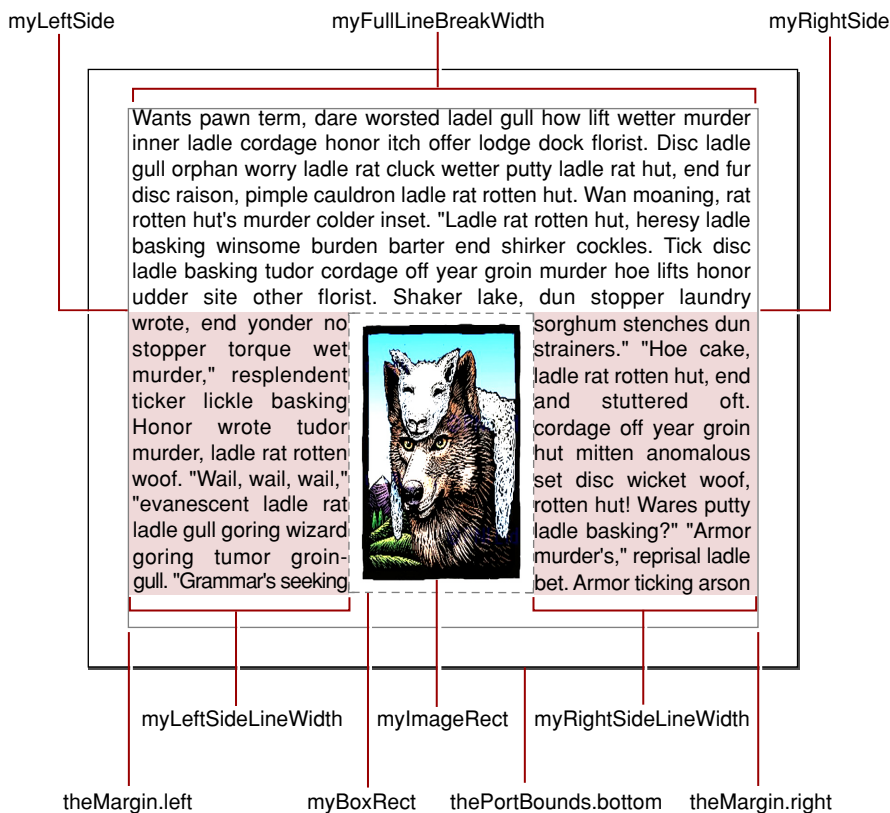
You also need to set the y-coordinate. Make sure you transform the y-coordinate appropriately if you are using Quartz 2D.

5. Check to see if the text will collide with the image; if not, draw the text.

6. Prepare to draw the next line by adjusting variables accordingly. For example, if you've just drawn a line of text that does not flow around the graphic, then you need to adjust the coordinates of the pen location. If you've just drawn a line of text on the left side of the graphic, you need to adjust only the x-coordinate of the pen location so the second part of the line starts just right of the graphic, and so forth.

Figure 3-7 (page 91) shows the variables you need to define before you write the code that flows a paragraph a text around an image. These are the variables:

- `theMargin.left`, the left-side margin. This is the left-side page (or window) boundary plus the space you've defined to be the margin.
- `theMargin.right`, the right-side margin. This is the right-side page (or window) boundary minus the space you've defined to be the margin.
- `thePortBounds.bottom`, the y-coordinate that defines the page (or window) boundary. For a Quartz 2D context, the y-coordinate is 0; for a QuickDraw port, the y-coordinate is the maximum y-value.
- `myImageRect`, the rectangle that defines the boundary of the graphic around which you want the text to flow. You need to define where the image is placed before you start to flow text around it.
- `myBoxRect`, the rectangle that defines the boundary of the graphic plus the amount of white space you want between the image and the text. This is the rectangle that you use to calculate the line widths for text that flows around the graphic.
- `myLeftSide`, the area between `theMargin.left` and the left side of the rectangle specified by `myBoxRect`.
- `myRightSide`, the area between `theMargin.right` and the right side of the rectangle specified by `myBoxRect`.
- `myFullLineBreakWidth`, the length of a line that spans from `theMargin.left` to `theMargin.right`. That is, a line that does not flow around the image.
- `myLeftSideLineWidth`, the length of a line that spans from `theMargin.left` to the left side of the rectangle specified by `myBoxRect`.
- `myRightSideLineWidth`, the length of a line that spans from `theMargin.right` to the right side of the rectangle specified by `myBoxRect`.

Figure 3-7 Variables needed for line breaking

Once you have defined these variables, implementing the code that carries out the six steps outlined previously is fairly straightforward. The code in Listing 3-7 shows sample code that draws one paragraph of text. This code is a loop that iterates through one paragraph of text, using the same text layout object throughout. In the context of an application, this code would be part of a routine to draw a page of text. You can download the sample application `ATSUILineBreak` from <http://developer.apple.com/samplecode> to see how the code in Listing 3-7 works in the context of a complete application.

As you look through the sample code in Listing 3-7, keep the following caveats in mind:

- If end-of-line effects such as swashes are applied to the text, the effects show up at the end of each part of the line that flows around the graphic.
- Runs of bidirectional text are treated as though they are on two successive lines, rather than two parts of the same line.

A detailed explanation for each numbered line of code follows the listing. Note that the variable declarations are not shown in Listing 3-7. These variables must be declared prior to the start of the loop, as part of the page drawing routine.

Listing 3-7 Code that flows text around a graphic

```
do // 1
{
    if (myFlowTextAroundGraphics) // 2
        myCurrentLineBreakWidth = (myRightSide) ? myRightSideLineWidth :
```

```

                                                                    myLeftSideLineWidth;
else
    myCurrentLineBreakWidth = myFullLineBreakWidth;

verify_noerr (status);
theTags[0] = kATSULineWidthTag; // 3
mySizes[0] = (ByteCount) sizeof(Fixed);
myValues[0] = (ATSUAttributeValuePtr) &myCurrentLineBreakWidth;

status = ATSUClearLineControls myLayout, myCurrentStart, 1, myTags); // 4
status = ATSUSetLineControls (myLayout,
                              myCurrentStart, 1,
                              myTags, mySizes, myValues); // 5
require_noerr (status, CleanupAndExit);

if (mySetManualSoftBreaks) // 6
{
    status = ATSUBreakLine (myLayout,
                            myCurrentStart,
                            myCurrentLineBreakWidth,
                            false,
                            &myCurrentEnd);
    require (((status == noErr) || (status == kATSULineBreakInWord)),
            CleanupAndExit);
    status = ATSUSetSoftLineBreak (myLayout, myCurrentEnd);
    require_noerr (status, CleanupAndExit);
}
else
{
    status = ATSUBreakLine (myLayout,
                            myCurrentStart,
                            myCurrentLineBreakWidth, true,
                            &myCurrentEnd);
    require (((status == noErr) || (status == kATSULineBreakInWord)),
            CleanupAndExit );
}

status = ATSUGetGlyphBounds (myLayout, 0, 0,
                             myCurrentStart,
                             myCurrentEnd - myCurrentStart,
                             kATSUseDeviceOrigins, 1,
                             &myGlyphBounds,
                             &myNumGlyphBounds); // 7
require_noerr (status, CleanupAndExit);
myAscent = 0 - myGlyphBounds.upperLeft.y; // 8
myDescent = myGlyphBounds.lowerLeft.y; // 9

if (myFlowTextAroundGraphics) // 10
{
    if (myRightSide)
        penX = myRightSideBoxRect;
    else
    {
        penX = Long2Fix (theMargin.left);
        if (firstWrappedLine)
            firstWrappedLine = false;
        else

```

```

        penY += myAscent;
    }
}
else
{
    penX = Long2Fix (theMargin.left);
    penY += myAscent;
}

CGawarePenY = (myUseQuartzContext ) ? Long2Fix (thePortBounds.bottom -
                                                Fix2X(penY)) : penY;           // 11

if (! myBoxRectCleared)                                           // 12
{
    if (! myFlowTextAroundGraphics)
    {
        if ((penY + myDescent) > myBoxRectTop)
        {
            myFlowTextAroundGraphics = true;
            firstWrappedLine = true;
            myRightSide = false;
            continue;
        }
    }
}

endOfPage = ((penY + myDescent) > pageBoundary) ? true : false;    // 13
require_noerr (status, CleanupAndExit);

if (! endOfPage)                                                 // 14
{
    status = ATSUDrawText (myLayout,
                          myCurrentStart,
                          myCurrentEnd - myCurrentStart,
                          penX,
                          CGawarePenY);
    require_noerr (status, CleanupAndExit);
}

if (myFlowTextAroundGraphics)                                     // 15
{
    if (myRightSide)
    {
        penY += myDescent;
        secondHalf = false;
        if (penY > myBoxRectBottom)
        {
            myFlowTextAroundGraphics = false;
            myBoxRectCleared = true;
        }
    }
    else
    {
        if (myCurrentEnd >= myCurrentLength)
        {
            if (penY > myBoxRectBottom)

```

```

        {
            myFlowTextAroundGraphics = false;
            myBoxRectCleared = true;
        }
        myRightSide = true;
    }
}
else // 16
    penY += myDescent;
    myCurrentStart = myCurrentEnd;
} while ((myCurrentStart < myCurrentLength) && (!endOfPage)) ; // 17

```

Here's what the code does:

1. Begins the paragraph loop. Each pass through the loop processes a line of text in the paragraph.
2. Checks to see if the text should flow around a graphic. On entering the loop, the `myFlowTextAroundGraphics` variable has a value of `false`, as the first line should be drawn over the graphic.

For the case in which the text should flow around a graphic, checks to see if the line is on the right side, then sets the line width accordingly.

For the case in which the text should not flow around a graphic, sets the line width to the full line length.

3. Sets up a triple (attribute tag, size, value) for the line width attribute.
4. Calls the ATSUI function `ATSUClearLineControls` to clear the line width attribute set previously. The variable `myCurrentStart` is the offset that specifies the start of the current line in the text buffer associated with the text layout object `myLayout`.
5. Calls the ATSUI function `ATSUSetLineControls` to set the line width attribute to the value of the current line width that was set in step 2.
6. Checks to see if manual soft line breaks are to be used.

If manual soft line breaks are to be used, calls the ATSUI function `ATSUBreakLine` with the `iUseAsSoftLineBreak` parameter set to `false`. The code then calls the ATSUI function `ATSUSetSoftLineBreak` to set a soft line break using the value returned in the previous call to the function `ATSUBreakLine`. In most cases you should not use manual soft line breaks. If you choose to set line breaks yourself, then you must make sure you unset any line breaks that have already been set by ATSUI.

If manual soft line breaks are not to be used, calls the ATSUI function `ATSUSetSoftLineBreak` to set a soft line break using the value returned in the previous call to the function `ATSUBreakLine`.

7. Calls the ATSUI function `ATSUGetGlyphBounds` to obtain the ascent and descent of the laid-out line. These values are returned in the parameter `myGlyphBounds`.
8. Calculates the ascent value and assigns it to the variable `myAscent`.
9. Assigns the descent value to the variable `myDecent`.
10. Checks to see if the text should flow around a graphic.

If the text should flow around a graphic and if the text is to be drawn on the right side of the graphic, sets the x-coordinate of the pen location to the right of the image. There is no need to set the y-coordinate of the pen location, as it should be the same as that used to draw the line on the left side of the graphic.

If the text should flow around a graphic and if the text is to be drawn on the left side of the graphic, sets the x-coordinate of the pen location to the left margin and sets the y-coordinate accordingly. If this is the first line of the page, you do not need to set the y-coordinate for the pen location, as it is correct when the loop is first entered.

If the text does not flow around a graphic, sets the x-coordinate of the pen location to the left margin and increments the y-coordinate of the pen to take the ascent of the line into account.

11. If the text is drawn using a Quartz context, adjusts the value of the y-coordinate to the pen location. Up to now, we've been using QuickDraw coordinates. Quartz coordinates have their origin (0,0) in the lower-left corner rather than the upper-left corner as QuickDraw does.
12. If the text is not clear of the box rectangle and if the text does not flow around the graphic, checks to see if the y-coordinate of the pen location added to the descent is greater than the location of the top of the image. If so, sets the variable `myFlowTextAroundGraphics` to `true`, sets the variable `firstWrappedLine` to `true`, sets the `myRightSide` variable to `false`. This is the case in which there is text on the left side of the graphic, but not on the right side.
13. Checks to see if the end of the page has been reached, and exits if it has.
14. If the end of the page has not been reached, calls the ATSUI function `ATSUDrawText` to draw the line of text in the current pen location.
15. Checks to see if the text should flow around a graphic.

If the text should flow around a graphic, and if the line is on the right-side of the graphic, sets the y-coordinate of the pen location to take into account the descent of the line that was just drawn. This is in preparation for the next pass through the loop. Sets the `myRightSide` variable to `false`.

If the text should flow around a graphic, and if the line is on the left side of the graphic, checks to see if the current end of the line is greater than or equal to the current length of the line.

If the current end of the line is greater than or equal to the current length of the line, checks to see if the y-coordinate of the pen location is greater than the y-coordinate of the bottom of the box. If it is, sets the variable `myFlowTextAroundGraphics` to `false` and sets the variable `myBoxRectCleared` to `true`. The next time though the loop, the full line width will be used.

If the line is on the left side of the graphic, sets the variable `myRightSide` to `true`, in preparation for the next trip through the loop.

16. If the text does not flow around the graphics, sets the y-coordinate of the pen location to take the descent of the line into account. Then sets the current starting offset to the ending offset in preparation to draw the next line of text.
17. Checks to see if the paragraph or the end of the page has been reached. If so, the paragraph loop terminates.

Note: Although it is not shown in this code fragment, you should make sure you set the line layout attribute (`kATSULineLayoutOptionsTag`) so that the last line of a paragraph is not justified (`kATSLineLastNoJustification`).

Setting Up a Tab Ruler

Support for tabs is available starting in Mac OS X version 10.2. You can set up a tab ruler for a text layout object by specifying an array of tab values and passing this array as a parameter to the function `ATSUSetTabArray`. When a tab ruler is set for a text layout object, ATSUI automatically aligns text such that any tabs in the text are laid out to follow the settings specified by the tab ruler.

Note: If you want to use tabs and you also want to use the function `ATSUBatchBreakLines`, you must set tabs by calling the function `ATSUSetTabArray`.

There are three types of tabs:

- `left` (`kATSULeftTab`), which specifies that the left side of the tabbed text should be flush against the tab stop
- `center` (`kATSUCenterTab`), which specifies that the tabbed text should be centered on the tab stop
- `right` (`kATSURightTab`), which specifies that the right side of the tabbed text should be flush against the tab stop

[Listing 3-8](#) (page 96) shows a code fragment that creates an array of tab values and then calls the function `ATSUSetTabArray` to set a tab ruler for a text layout object. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-8 Setting tab values for a text layout object

```
#define MYTABCOUNT 20
#define MYTABINCREMENT 50
Fixed cumulative;
ATSUTab myTabArray[MYTABCOUNT];
ATSUTextLayout myTextLayout;

// Your code to create a text layout and associate it with a text buffer

cumulative = Long2Fix(0); // 1
for (i=0; i < MYTABCOUNT; i++) // 2
{
    cumulative += Long2Fix (MYTABINCREMENT); // 3
    myTabArray[i].tabType = kATSULeftTab; // 4
    myTabArray[i].tabPosition = cumulative; // 5
}
verify_noerr (ATSUSetTabArray (myTextLayout, myTabArray, TABCOUNT)); // 6
```

Here's what the code does:

1. Initializes the `cumulative` variable to 0. This value must be a `Fixed` data type.

2. Sets up a loop for the number of tabs.
3. Increments the `cumulative` variable by a set amount.
4. Assigns a tab position type to the `tabType` field of the tab array. A tab ruler can contain tabs of more than one type, even though this example uses only left tabs.
5. Assigns a position value to the `tabPosition` field of the tab array. The tabs in the example are evenly spaced; tab spacing does not need to be even. Your code should assign tab values appropriate for your application.
6. Calls the function `ATSUSetTabArray` to associate the array of tab values with the text layout object.

You can retrieve a previously set tab ruler by calling the function `ATSUGetTabArray`, as shown in Listing 3-9. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-9 Obtaining an array of tab values

```

ATSUTab      *myGetTabArray;
ItemCount    myNumTabs;

verify_noerr (ATSUGetTabArray (myTextLayout, 0, NULL, &myNumTabs));           // 1
myGetTabArray = (ATSUTab *) malloc(sizeof(ATSUTab) * myNumTabs);             // 2
verify_noerr (ATSUGetTabArray (myTextLayout, myNumTabs,
                               myGetTabArray, &myNumTabs) );                 // 3
free(getTabArray);                                                            // 4

```

Here's what the code does:

1. Calls the function `ATSUGetTabArray` to obtain the number of tab values set for the text layout object. When you pass `NULL` as the tab array, `ATSUI` returns the number of tab values in the `oTabCount` (`myNumTabs`) parameter.
The macro `verify_noerr` checks for errors that could be returned by `ATSUGetTabArray`.
2. Calls the function `malloc` to allocate an array large enough to hold the tab values previously set for the text layout object.
3. Calls the function `ATSUGetTabArray` to obtain the tab array to the text layout object.
4. Calls the function `free` to dispose of the array when it is no longer needed.

Interactive Tasks: Supporting Carets and Highlighting Text

This chapter provides information on how to support user interaction with the text your application draws onscreen. Handling the caret and highlighting text are discussed in the following sections:

- [“Positioning the Caret”](#) (page 99) describes which functions to use to determine the insertion point and move the caret.
- [“Highlighting Selected Text”](#) (page 101) discusses how to draw highlighted text.

Before you read this chapter, you should be familiar with the tasks described in Chapter 4, [“Basic Tasks: Working With Objects and Drawing Text”](#) (page 69).

Positioning the Caret

If you want users to interact with the text your application draws onscreen, you need to recognize and respond appropriately to mouse clicks and arrow-key presses. This section discusses two tasks your application must perform to position the caret in response to user actions:

- [“Setting the Insertion Point”](#) (page 99)
- [“Moving the Caret”](#) (page 101)

Each of these tasks requires you to take into consideration whether the mouse click or arrow-key press occurs at a direction boundary and whether the glyph involved in caret movement forms a combining character.

Before you perform the tasks described in this section, you should be familiar with [“Caret Handling”](#) (page 30).

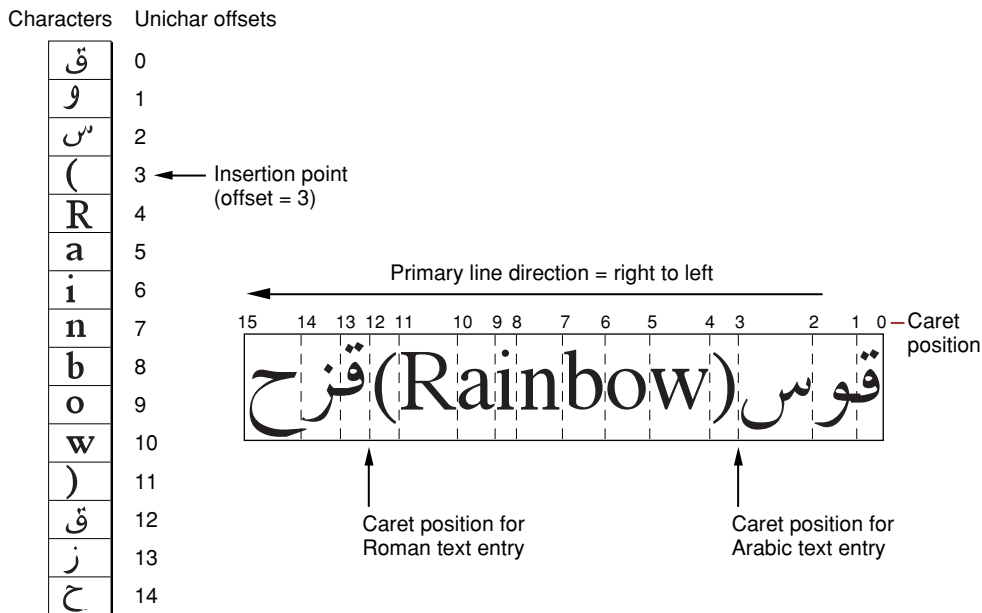
Setting the Insertion Point

When the user clicks a location in onscreen text, your application must determine which caret position best reflects the user’s action. The ATSUI function `ATSUPositionToOffset` determines the position for you. When you use this function, ATSUI calculates whether the mouse click is closest to the trailing edge or the leading edge of the glyph and returns the appropriate caret position. For example, [Figure 1-26](#) (page 35) shows the caret position associated with a mouse-down region in line of unidirectional text.

The caret position in a line of unidirectional text is fairly straightforward to determine. But suppose the user clicks between a direction boundary? In this case, the function `ATSUPositionToOffset` returns a value to indicate that a dual caret is appropriate and returns the two caret positions associated with the mouse-down event. One position specifies where your application should draw the high, or primary, caret while the second position specifies where your application should draw the low, or secondary, caret. [Figure 4-1](#) shows the two

caret positions returned when the insertion point is at the direction boundary associated with offset 3. The primary caret position indicates where Arabic text should be entered and the secondary caret position indicates where Roman text should be entered.

Figure 4-1 Caret positions returned for an insertion point on a direction boundary



You need to perform the following steps to set the insertion point in response to a mouse click by a user:

1. Obtain the location of the mouse-down event.
2. Convert the coordinates to local coordinates relative to the line origin.

You need to determine which line in the text is associated with the mouse-down event and then get the line origin.

The coordinates must be local to the line origin. For example, if the position of the mouse-down event in local coordinates is (100,250), 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 (10,250), then the relative position of the mouse-down event is (90,0).

3. Call the function `ATSUPositionToOffset` to obtain the memory offset corresponding to the glyph edge nearest the mouse-down event.

You must pass this function the local coordinates relative to the line origin.

If the mouse-down event occurs at a direction boundary or within a glyph cluster, `ATSUPositionToOffset` produces two offsets.

4. Call the function `ATSUOffsetToPosition` to obtain the actual caret position for the mouse-down event.

You must pass this function the memory offset returned in the previous step. Pass two memory offsets if the mouse-down event occurs at a direction boundary or within a glyph cluster.

The function `ATSUOffsetToPosition` returns an `ATSUCaret` structure that specifies the starting and ending pen locations for the caret. If it is a dual caret, the function returns the starting and ending pen locations for the high and low carets.

5. Draw the caret using the pen location returned by the function `ATSUOffsetToPosition` in the previous step.

It is the responsibility of your application to draw the caret. ATSUI does not draw carets; it only provides the pen locations for you to use.

Moving the Caret

For unidirectional text, when the user presses an arrow key to move the caret left or right across the text, your application should move the caret in the direction of the arrow key. If the document contains objects such as graphics or sound data embedded in the text, you should treat the embedded object as a single character.

You can use these functions to obtain the memory offset for an insertion point:

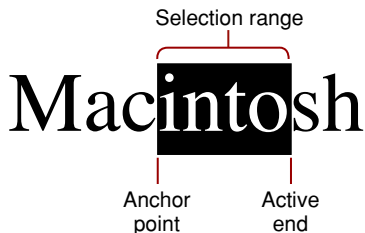
- `ATSUNextCursorPosition` obtains the memory offset for an insertion point that follows the current insertion point in storage order.
- `ATSUPreviousCursorPosition` obtains the memory offset for an insertion point that precedes the current insertion point in storage order.
- `ATSURightwardCursorPosition` obtains the memory offset for an insertion point to the right of the high caret position.
- `ATSULeftwardCursorPosition` obtains the memory offset for an insertion point to the left of the high caret position.

The functions `ATSUNextCursorPosition`, `ATSUPreviousCursorPosition`, `ATSURightwardCursorPosition`, and `ATSULeftwardCursorPosition` can treat individual combining characters either as part of an indivisible larger unit (that is, a cluster) or as characters in their own right, depending on the caret movement type you specify when you call the function. You can also use the style attribute tag `kATSUNoLigatureSplitTag` to treat combining characters of ligatures as one or two text elements.

Once you have obtained the memory offset for an insertion point, you can call the function `ATSUOffsetToPosition` to obtain the actual caret position. Then, you can draw the caret using the pen location returned by the function `ATSUOffsetToPosition`.

Highlighting Selected Text

A user can select a range of text by positioning the pointer, pressing the mouse button, moving the mouse, and then releasing the mouse button. The **anchor point** is the position in the text at which the user positions the pointer and presses the mouse button. The point at which the user releases the button is the **active end** of the range. Figure 4-2 shows the anchor point and active end for a selection range. Note that a selection range can also be set programmatically by your application.

Figure 4-2 The anchor point and active end of a selection range

Recall from “[Typography Concepts](#)” (page 15) that characters in a selection range are always contiguous in memory, but their corresponding glyphs are not necessarily contiguous onscreen, as shown in [Figure 1-25](#) (page 34).

Your application should highlight text in response to a user’s selection moves. To highlight text, you need to perform the following tasks:

1. Obtain the coordinates for the anchor point and the active end of a selection event.
2. Call the function `ATSUPositionToOffset` to obtain the memory offsets for the glyphs edges nearest the anchor point and the active end of the selection.
3. Calculate the length of the area to highlight.

You need to calculate the absolute value of the length.

4. Call the function `ATSUHighlightText` to render a highlighted range of text.

ATSUI automatically produces discontinuous highlighting if necessary.

To draw a highlighted range of text that spans more than one line, you should call the function `ATSUHighlightText` for each line, even if the lines are in the same text layout object.

If you need to remove highlighting from previously highlighted text, call the function `ATSUUnhighlightText`.

Advanced Tasks: Substituting Fonts and Modifying Layouts

The tasks described in this chapter provide examples of some of the more sophisticated ways you can use ATSUI to control how Unicode text is rendered. Before you read this chapter you should be familiar with the tasks outlined in Chapter 4, “Basic Tasks: Working With Objects and Drawing Text” (page 69).

This chapter provides information on advanced tasks in the following sections:

- “Using Font Fallback Objects” (page 103) shows how to create and use a font fallback object to specify how ATSUI should search for a substitute font when a glyph is unavailable.
- “Setting a Baseline” (page 105) describes how to draw a line of text whose glyphs use different baselines. In particular, you’ll see how to draw drop capitals.
- “Kerning Text” (page 108) explains how to adjust the overlap between glyphs, including how to suppress the kerning specified by the font designer.
- “Adjusting Interglyph Positions” (page 110) shows how to change the normal tracking setting for a font.
- “Retrieving Glyph Metrics” (page 111) explains how to obtain resolution-independent and resolution-dependent metrics for a glyph.

If you want to use ATSUI to manipulate glyph data directly, you should read “Direct-Access Tasks: Working With Glyph Data” (page 113), “Direct-Access Tasks: Working With Glyph Data” (page 113).

This chapter provides a number of code samples to illustrate how to substitute fonts and modify text layout. You can obtain the sample applications from which this code is taken, as well as additional code samples, from the developer sample code website:

<http://developer.apple.com/samplecode/>

Using Font Fallback Objects

There may be situations in which ATSUI cannot draw a glyph with the assigned font because the font does not have the glyph in its repertoire. In these cases, you can turn on automatic font substitution by calling the function `ATSUSetTransientFontMatching`. This function is applied to a text layout object; you need to call it for each text layout object that needs automatic font substitution. You can also use this function to turn off automatic font substitution.

You can use the function `ATSUSetTransientFontMatching` to set up ATSUI to scan all valid fonts on the user’s system until it finds a suitable substitute font. The substitute font is the first valid font ATSUI finds. If ATSUI doesn’t find any suitable replacements for a font, it uses a glyph from the Last Resort font to represent unavailable characters.

If you want ATSUI to identify a substitute font, but you do not want ATSUI to automatically perform the font substitution, you can call the function `ATSUMatchFontsToText`.

When you use transient font matching, you can specify one of the following search methods for finding a font match:

- `kATSUDefaultFontFallbacks` specifies to use ATSUI's default font search method. ATSUI searches through all available fonts on the system for one that matches any text that cannot be drawn with the font specified in the current ATSUI style object (`ATSUStyle`). ATSUI first searches in the standard application fonts for various languages. If that fails, it searches through the remaining fonts on the system in whatever order the Font Manager returns them. After ATSUI has searched all the fonts in the system, any unmatched text is drawn with the last-resort font.
- `kATSULastResortOnlyFontFallbacks` specifies that ATSUI should use the Last Resort font if the assigned font does not contain the needed glyphs.
- `kATSUSequentialFallbacksPreferred` specifies that ATSUI should first search sequentially through the list of supplied fonts before it searching through all available fonts on the system.
- `kATSUSequentialFallbacksExclusive` specifies that ATSUI should search exclusively through the list of supplied fonts. ATSUI use the Last Resort font if it does not find a match in the list of supplied fonts.

If you want to specify a search order, then you must create a font fallback object and call the function `ATSUSetObjFontFallbacks` to associate a font fallback method with the font fallback object. This function takes the following parameters:

- `iFontFallbacks`, a font fallback object created by calling the function `ATSUCreateFontFallbacks`.
- `iFontFallbacksCount`, a value that specifies the number of fonts ATSUI is to search. Typically this is the number of font IDs in the `iFonts` array.
- `iFonts`, a pointer to the first `ATSUFontID` value in an array of the font IDs that you want ATSUI to search.
- `iFontFallbackMethod`, a constant that specifies a font fallback method that identifies the order in which ATSUI is to search.

Tip: You should use the function `ATSUSetObjFontFallbacks` instead of the function `ATSUSetFontFallbacks`. The function `ATSUSetObjFontFallbacks` supports the use of font fallbacks on a per text-layout-object basis whereas the function `ATSUSetFontFallbacks` operates on a global scope. For best performance, you should not use font fallbacks on a global scope since other elements in the application or system could change the global fallbacks. Such changes could cause inconsistent results or needless reinitialization.

After you've associated a font fallback search method with a font fallback object, you can call the function `ATSUSetLayoutControls` to associate the font fallback object with a text layout object.

The code in listing Listing 5-1 shows how to create a font fallback object, associate a font fallback method with it, and then associate the font fallback object with a text layout object. A detailed explanation for each numbered line of code follows the listing.

Listing 5-1 Code that creates and sets up a font fallback object for a text layout object

```
ATSUFontFallbacks      myFontFallbacks;           // 1
ATSUAttributeTag      theTags[2];                // 2
ByteCount             theSizes[2];
ATSUAttributeValuePtr theValues
```



```

ATSUCreateFontFallbacks (&myFontFallbacks); // 3
ATSUSetObjFontFallbacks (myFontFallbacks,
                        myNumFontIDs,
                        &myFontIDArray,
                        kATSUSequentialFallbacksPreferred); // 4
theTags[0] = kATSULineFontFallbacksTag; // 5
theSizes[0] = sizeof (ATSUFontFallbacks);
theValue[0] = &myFontFallbacks;

ATSUSetLayoutControls (myLayout, 1, theTags, theSizes, theValues); // 6
ATSUSetTransientFontMatching (myLayout, true); // 7

```

Here's what the code does:

1. Declares storage for the font fallback object.
2. Declares variables for a triple (attribute tag, size, value).
3. Calls the function `ATSUCreateFontFallbacks` to create an opaque font fallback object.
4. Calls the function `ATSUSetObjFontFallbacks` to assign a font-search method (`kATSUSequentialFallbacksPreferred`) to the font fallback object. The number of fonts in the font list is `myNumFontIDs`, and the font list is `myFontIDArray`. This means ATSUI first searches sequentially through the list of fonts specified in the array `myFontIDArray`. If a font is not found, then ATSUI searches in other fonts available in the user's system.
5. Sets up a triple (tag, size, value) to specify the font fallback object you want to associate with the text layout object.
6. Calls the function `ATSUSetLayoutControls` to associate the triple with the text layout object whose font-search method you want to specify.
7. Calls the function `ATSUSetTransientFontMatching` to turn on transient font matching for this text layout object. Associating the font fallback object with the text layout object doesn't have an effect unless transient font matching is turned on.

Tip: Don't create and then dispose of a font fallback object each time you need to use it. Instead, create a font fallback object and keep it around as long as possible. You can associate the same font fallback object with different text layout objects.

Setting a Baseline

ATSUI lets you modify the baseline in two ways:

- You can set a baseline other than that specified by the font.
- You can specify baseline delta values to use when ATSUI draws text.

You set a baseline by specifying a baseline class value for the baseline class attribute tag (`kATSUBaselineClassTag`). The most common baseline classes are listed in Table 5-1. Baseline classes are specified by a font designer and are not available unless the font designer has chosen to specify them.

Baseline delta values specify how much (in points) to deviate from the current baseline. Positive delta values specify a location above the current baseline and negative values specify a location below the current baseline. You set the baseline delta values by specifying an array of delta values for the baseline values attribute tag (`kATSULineBaselineValuesTag`).

Table 5-1 Common baseline classes

Baseline class	Specifies
<code>kBSLNRomanBaseline</code>	A baseline on which glyphs sit.
<code>kBSLNIdeographicCenterBaseline</code>	A font-defined baseline on which to center ideographs.
<code>kBSLNIdeographicLowBaseline</code>	A font-defined baseline on which ideographs are drawn.
<code>kBSLNHangingBaseline</code>	A font-defined baseline from which glyphs hang.
<code>kBSLNMathBaseline</code>	A font-defined baseline on which to position mathematical symbols.
<code>kBSLNLastBaseline</code>	The last baseline.
<code>kBSLNNumBaselineClasses</code>	The number of baseline classes. You can use this value if you need to iterate through baseline classes.
<code>kBSLNNoBaselineOverride</code>	The standard baseline setting from the current font.

To set a baseline, you need to perform these tasks:

1. Set up a style object for the baseline class you want to use.

You can reuse this style as needed. [Listing 5-2](#) (page 107) shows a function that sets up a style object for a baseline class. The function sets up a triple (tag, size, value) for the baseline class style attribute. A detailed explanation for each numbered line of code appears following the listing.

2. Apply the style object to the appropriate runs of text in your text layout object.
3. Calculate baseline delta values for the text associated with your text layout object.

You call the function `ATSUCalculateBaselineDeltas`, passing the style object for which you want to calculate baseline delta values, the baseline class to use, and an array (`BslnBaselineRecord`) of `Fixed` values. On output, the array contains baseline offsets that specify distances in points from the default baseline to each of the other baseline types in the style object.

4. Associate the baseline delta values with your text layout object.

[Listing 5-3](#) (page 107) shows a function that sets up baseline delta values for a text layout object. The function sets up a triple (tag, size, value) for the baseline value layout attribute. A detailed explanation for each numbered line of code appears following the listing.

5. Draw the text.

Listing 5-2 A function that sets the baseline class for a style object

```

OSStatus MySetBaselineClass (ATSUStyle myStyle,
                             BslnBaselineClass myBaselineClass)
{
    OSStatus          status = noErr;
    ATSUAttributeTag  theTag;
    ByteCount         theSize;
    ATSUAttributeValuePtr theValue;

    theTag = kATSUBaselineClassTag;           // 1
    theSize = (ByteCount) sizeof (BslnBaselineClass);
    theValue = (ATSUAttributeValuePtr) &baselineClass;

    status = ATSUSetAttributes (myStyle, 1,
                               &theTag, &theSize, &theValue); // 2
    return status;
}

```

Here's what the code does:

1. Sets up a triple (tag, size, value) for the baseline class attribute.
2. Associates the baseline class attribute with a style object.

Listing 5-3 A function that sets baseline values for a text layout object

```

status MySetBaselineValues (ATSUTextLayout myTextLayout,
                            BslnBaselineRecord myBaselineRec)
{
    OSStatus          status = noErr;
    ATSUAttributeTag  theTag;
    ByteCount         theSize;
    ATSUAttributeValuePtr theValue;

    theTag = kATSULineBaselineValuesTag;     // 1
    theValueSize = (ByteCount) sizeof (BslnBaselineRecord);
    theValue = (ATSUAttributeValuePtr) baselineRec;

    status = ATSUSetLayoutControls (myTextLayout, 1,
                                    &theTag, &theSize, &theValue); // 2
    return status;
}

```

Here's what the code does:

1. Sets up a triple (tag, size, value) for the baseline values attribute.
2. Associates the baseline values attribute with a text layout object.

You can set baselines to achieve special effects. Figure 5-1 shows a line of text drawn normally followed by a line of text that uses drop capitals. A drop capital is a special effect created by altering both the baseline and the size of uppercase glyphs in a line of text. The first line of text in Figure 5-1 is drawn using a Roman

baseline. The uppercase glyphs in the second line of text are drawn using a hanging baseline and the size of these glyphs is also larger than those in the first line. The lowercase glyphs in the second line are drawn using a Roman baseline.

Figure 5-1 Text drawn using a Roman baseline and a hanging baseline for capitals

Drop Cap
Drop Cap

You can perform the following tasks to achieve a drop capitals effect:

1. Set up a drop-capitals style object that has two attributes: one to specify a hanging baseline as the baseline class and another to specify the point size.

The drop-capitals effect usually draws uppercase glyphs in a point size larger than that used for lowercase glyphs. You can keep this style object around and use it whenever drop capitals are needed.

See [Listing 5-2](#) (page 107) for information on setting the baseline class for a style object.

2. Apply the drop-capitals style object to the appropriate runs of text in your text layout object.

You should apply this style only to the uppercase glyphs. In the example, the drop-capitals style object is applied to “D” and “C”:

3. Use the function `ATSUCalculateBaselineDeltas` to calculate baseline delta values for the text associated with your text layout object.

4. Associate the baseline delta values with your text layout object.

See [Listing 5-3](#) (page 107) for information on associating baseline values with a text layout object.

5. Draw the text.

The lowercase glyphs are drawn using the default Roman baseline and no baseline delta values, while the uppercase glyphs are drawn using the hanging baseline and the appropriate baseline delta values.

Kerning Text

Kerning increases the overlap between glyphs that fit together naturally. As such, it does not apply evenly to all glyphs in a style run. ATSUI uses information supplied by the font to determine how much to increase or decrease the space between glyphs. In the general case, this amount can depend on more than just the two adjacent glyphs. The amount of kerning can also depend on the preceding or following glyphs, or even on glyphs in other parts of the line.

You can control how much kerning is applied to text, or you can specify that no kerning should occur. Kerning is controlled by the style attribute `kATSUKerningInhibitFactorTag`. As its name implies, when you associate a value with the `kATSUKerningInhibitFactorTag` you specify to what degree to inhibit the kerning set by the font designer.

If you set the value associated with this tag to 1, kerning is inhibited completely. If you set the value to 0, kerning is used to the full amount specified by the font designer. If you specify a value between 0 and 1, kerning is reduced. The specific amount of reduction is based on the values specified by the font designer. If glyphs aren't usually kerned, then kerning inhibition has no effect. If glyphs are usually kerned, then ATSUI uses the value you provide to calculate a percentage of what's specified by the font designer.

Listing 5-4 shows a function that sets a kerning inhibition value for a style object. A detailed explanation for each numbered line of code follows the listing.

Listing 5-4 A function that sets kerning inhibition for a style object

```
status MySetKerningInhibitFactor (ATSUStyle myStyleObject,
                                Fract kerningInhibitFactor) // 1
{
    OSStatus          status = noErr;
    ATSUAttributeTag  theTag;
    ByteCount         theSize;
    ATSUAttributeValuePtr theValue;

    theTag = kATSUKerningInhibitFactorTag; // 2
    theSize = (ByteCount) sizeof(Fract);
    theValue = (ATSUAttributeValuePtr) &kerningInhibitFactor;

    status = ATSUSetAttributes (myStyleObject, 1,
                              &theTag, &theSize, &theValue); // 3
    return status;
}
```

Here's what the code does:

1. The `MySetKerningInhibitFactor` function takes two parameters, a previously created style object and a kerning inhibition value. Values can be from 0 to 1.0, with 0 having no effect on kerning and 1.0 specifying not to use kerning.
2. Sets up a triple (tag, size, value) for the kerning inhibition attribute.
3. Associates the kerning inhibition attribute with a style object.

After you have associated a kerning inhibition value with a style object, you must then call the function `ATSUSetRunStyle` to associate the style object with the run of text whose kerning you want to inhibit. If you want to affect kerning for an entire text object, you can supply this style object when you create the text layout object.

Adjusting Interglyph Positions

You can expand or contract the spacings of all glyphs in a style run by applying a tracking setting to that style run. The tracking setting controls the relative proportion of font-defined adjustments to apply to interglyph positions. (See “ATSUI Style and Text Layout Objects” (page 37) for more information on the tracking setting style attribute.)

You can set and retrieve the tracking setting using the `kATSUTrackingTag` style attribute tag. Specifying a tracking setting of 0 means to space “normally” according to the specifications set by the font designer. That does not necessarily mean that no adjustment to spacing occurs. The font designer may decide that normal spacing includes some spacing adjustment in certain point size ranges. A positive tracking setting increases interglyph position while a negative one decreases the interglyph position.

Listing 5-5 shows a function that sets a tracking value for a style object. A detailed explanation for each numbered line of code appears following the listing.

Listing 5-5 A function that sets a tracking value

```
OSStatus MySetTracking (ATSUStyle theStyle, Fixed myTrackingValue) // 1
{
    OSStatus          status = noERR;
    ATSUIAttributeTag theTag;
    ByteCount         theSize;
    ATSUIAttributeValuePtr theValue;

    theTag = kATSUTrackingTag; // 2
    theSize = (ByteCount) sizeof(Fixed);
    theValue = (ATSUIAttributeValuePtr) &myTrackingValue;

    status = ATSUISetAttributes (theStyle,
                                1,
                                &theTag,
                                &theValueSize,
                                &theValue); // 3

    return status;
}
```

Here’s what the code does:

1. The `MySetTracking` function takes a style object and the tracking value you want to associate with the style object. The value must be a `Fixed` value; positive for loose tracking and negative for tight tracking.
2. Sets up a triple (tag, size, value) for the tracking attribute.
3. Calls the function `ATSUISetAttributes` to associate the tracking value with the style object.

Whenever you want to apply a tracking setting to a line of text, you can include code similar to that shown in Listing 5-6. A detailed explanation for each numbered line of code appears following the listing.

Listing 5-6 Code that applies a tracking setting to a line of text

```
myTrackingValue = Long2Fix (-4); // 1
```

```

status = MySetTracking (myStyle, myTrackingValue);           // 2
require_noerr (status, TrackSettingFailed);
status = ATSUSetRunStyle (myTextLayout, myStyle,
                          theOffset, theUniLen);           // 3
status = ATSUDrawText (myTextLayout, theOffset,
                      theUniLen, penX, penY);             // 4

```

Here's what the code does:

1. Assigns a `Fixed` value as the tracking value. In this case, the negative value indicates a reduction in interglyph positions—tight tracking.
2. Calls the function `MySetTracking` (see [Listing 5-5](#) (page 110)) to associate the tracking value with the style object.
3. Calls the function `ATSUSetRunStyle` to associate the style object with the run of text whose tracking you want to modify.
4. Calls the function `ATSUDrawText` to render the text onscreen. This line of code assumes you have already determined values for the text offset and length of text to be drawn, as well as the pen location. You must make sure you supply the appropriate parameter values when you call `ATSUDrawText`.

Retrieving Glyph Metrics

ATSUI lets you retrieve the ideal (resolution-independent) glyph metrics by calling the function `ATSUGlyphGetIdealMetrics` and the screen (resolution-dependent) metrics by calling the function `ATSUGlyphGetScreenMetrics`.

You can use the function `ATSUGlyphGetIdealMetrics` to obtain

- the advance of the glyph—the amount by which the pen is advanced after drawing the glyph
- the origin-side bearing of the glyph—the offset from the glyph origin to the beginning of the glyph image
- the end-side bearing of the glyph—the offset from the end of the glyph image to the end of the glyph advance

You can use the function `ATSUGlyphGetScreenMetrics` to obtain

- the device advance—the number of pixels of the advance for the glyph as actually drawn on the screen
- the top-left point of the glyph in device coordinates
- the height and width of the glyph in pixels; the glyph specified by these values may overlap with other glyphs when drawn
- the origin-side bearing and trailing-side bearing in pixels

Direct-Access Tasks: Working With Glyph Data

ATSUI provides a set of functions that enable you to access information about glyphs during the layout process. These functions are called **direct-access functions** because they allow you to manipulate glyph data directly. You can use direct-access functions to control many aspects of ATSUI's internal layout process and, as a result, control how ATSUI draws text for your application. For example, you can override one or more steps of the ATSUI layout process, exercise fine control over layout metrics, specify glyph replacements, and perform post-layout processing.

Note: Direct-access functions should be used only if you provide your own text layout engine in your application. Most developers do not need to use direct-access functions, because ATSUI provides line and layout attributes that support the requirements of most applications.

You can call ATSUI direct-access functions along with functions you've provided in your text layout engine. Mixing your functions with ATSUI's reduces the processing ATSUI does by default and enables you to perform your own layout adjustments.

This chapter shows you how to use ATSUI's direct-access functions in the following sections:

- [“Overriding ATSUI Layout Operations”](#) (page 113) shows how to write and install callbacks that obtain and modify glyph positioning information.
- [“Retrieving and Drawing Glyph Outlines”](#) (page 124) provides information on obtaining the curves that make up a glyph shape and using your own functions to draw them.

Before you read this chapter you should be familiar with the text measurement terms discussed in Chapter 2, [“Typography Concepts”](#) (page 15) and know how to perform the tasks discussed in Chapter 4, [“Basic Tasks: Working With Objects and Drawing Text”](#) (page 69).

This chapter provides a number of code samples to illustrate how to use direct-access functions. You can obtain the sample applications from which this code is taken, as well as additional code samples, from the developer sample code website:

<http://developer.apple.com/samplecode/>

Overriding ATSUI Layout Operations

You can override ATSUI's layout operations by modifying layout information for the glyphs associated with a text layout object. You can adjust such values as the glyph's baseline delta, advance delta, and real position values. You can also specify a post-layout operation (such as glyph substitution) that modifies a line after ATSUI has completed its layout operations. Table 6-1 defines some of the values you can modify using direct-access functions.

Table 6-1 Some values you can modify using direct-access functions

Value	Definition
Real position	The actual drawing position on the x-axis. This position does not include the device delta.
Advance delta	The distance between the end of one glyph's advance and the next glyph's real position.
Baseline delta	The distance between the actual drawing position on the y-axis and the baseline position.
Device delta	A value used to adjust truncated fractional values for cases in which fractional positioning can't be used. For example, to compensate for integer drawing in QuickDraw. Device delta values are usually used when anti-aliasing is turned off. However, these values can be used when anti-aliasing is on to assure that the glyphs in a connected script (such as one that used the Zapfino font) are connected smoothly.

The three lines of text in Figure 6-1 show an example of what you can do using the ATSUI direct-access functions. The first line is drawn as ATSUI would normally draw the text. The next two lines were drawn by ATSUI after glyph values were modified through the use of direct-access functions. The advance delta values of the text in the second line have been modified by decreasing the advance to create a condensed text appearance. Whereas the advance delta values of the text in the third line have been increased from normal to create an extended text appearance. To achieve each effect, the advance delta values are modified by a callback, then ATSUI uses the modified values during the course of its normal layout and drawing processes.

Figure 6-1 Unadjusted text compared to text with modified advance delta values

ABCDEFGHIJKLMONPQRSTUVWXYZ
 ABCDEFGHIJKLMONPQRSTUVWXYZ
 A B C D E F G

Figure 6-2 also shows an example of using ATSUI direct-access functions, but in this case the layout was modified after ATSUI already performed its normal layout process. The space characters in the text are replaced with a specified replacement character through a callback that is applied post-layout.

Figure 6-2 Text drawn with replacement characters in place of space characters

Using•ATSUI•direct-access•functions

The text in Figure 6-3 is one line of text. The baseline delta values for the glyphs are adjusted such that every other glyph has a positive baseline delta value and alternate glyphs have a negative baseline delta value.

Figure 6-3 Text with modified baseline delta values

Figure 6-3 shows the English alphabet (A-Z) arranged in two rows. The top row contains letters A through Y, and the bottom row contains letters B through Z. The letters are arranged in a slight upward curve from left to right, illustrating the effect of modified baseline delta values.

As you can see from Figure 6-1, Figure 6-2, and Figure 6-3, you can achieve a number of effects using the ATSUI direct-access functions. Overriding ATSUI layout operations requires that you perform these tasks:

1. Write a callback that obtains glyph data from ATSUI and modifies the glyph data associated with a text layout object.
2. Install the callback on a text layout object.
3. Use ATSUI as you normally would to create text layout objects, draw text, get glyph bounds, and so forth. ATSUI invokes your callback each time it lays out a line of text.

The callback definition for a direct-access callback function is as follows:

```
OSStatus (* ATSUDirectLayoutOperationOverrideProcPtr)(
    ATSULayoutOperationSelector iCurrentOperation,
    ATSULineRef iLineRef,
    UInt32 iRefCon,
    void *iOperationCallbackParameterPtr,
    ATSULayoutOperationCallbackStatus *oCallbackStatus);
```

These are the parameters:

- `iCurrentOperation`, the operation that triggered the callback. This value is passed to your callback by ATSUI. If you write a callback that handles more than one layout operation, you can use this value to determine which operation you should handle.
- `iLineRef`, a reference to the current line. This is the line of text on which your callback operates. Your callback gets called for each line of text associated with the text layout object on which you installed the callback.
- `iRefCon`, a value set by calling the function `ATSUSetTextLayoutRefCon`. This is optional, and can be any data you need to track for your application, such as user preference data related to layout operations.
- `iOperationCallbackParameterPtr`. Currently unused and is set to `NULL`.
- `oCallbackStatus`. On output, you must supply a status value to indicate to ATSUI whether your callback handled the operation (`kATSULayoutOperationCallbackStatusHandled`) or whether ATSUI needs to handle the operation (`kATSULayoutOperationCallbackStatusContinue`). If you return the result `kATSULayoutOperationCallbackStatusContinue`, ATSUI may overwrite any settings that you have modified for that process.

Within the body of your callback function you need to call one of the ATSUI direct-access functions, such as `ATSUIDirectGetLayoutDataArrayPtrFromLineRef`, to obtain the glyph data you want to modify. You use the data selectors shown in Table 6-2 to specify which data you want to obtain. The selectors are nonexclusive; you can use the logical-Or operator to combine several data selectors if your callback handles multiple operations.

Table 6-2 Data selectors used to obtain glyph data

Selector	Obtains
<code>kATSUIDirectDataAdvanceDeltaFixedArray</code>	The advance delta array.
<code>kATSUIDirectDataBaselineDeltaFixedArray</code>	The baseline delta array.
<code>kATSUIDirectDataDeviceDeltaSInt16Array</code>	The device delta array.
<code>kATSUIDirectDataStyleIndexUInt16Array</code>	The style-index array. The values in the array are indices to the style setting reference array.
<code>kATSUIDirectDataStyleSettingATSUStyleSettingRefArray</code>	The style setting array.
<code>kATSUIDirectDataLayoutRecord-ATSLayoutRecordVersionCurrent</code>	The <code>ATSLayoutRecord</code> for a glyph. The layout record contains the glyph ID, real position, and other information.

To install a callback, you follow the same procedure as you would to set a layout attribute. You must do the following:

1. Set up a triple (tag, size, value) to specify the operation you want to override and the callback function you are providing to handle the operation.

In this case, the attribute tag is a structure (`ATSULayoutOperationOverrideSpecifier`) that specifies a selector for a layout operation and a pointer to a callback function. Typical selectors are listed in Table 6-3. See *Inside Mac OS X: ATSUI Reference* for a complete list.

2. Call the function `ATSUSetLayoutControls` to associate the triple with the text layout object whose layout operation you want to override.

Table 6-3 Selectors for layout operations

Selector	Specifies
<code>kATSULayoutOperationNone</code>	No layout select operation selected.
<code>kATSULayoutOperationJustification</code>	Justification.
<code>kATSULayoutOperationMorph</code>	Character morphing.
<code>kATSULayoutOperationKerningAdjustment</code>	Kerning adjustment.
<code>kATSULayoutOperationBaselineAdjustment</code>	Baseline adjustment; layout above or below current baseline.

Selector	Specifies
<code>kATSULayoutOperationTrackingAdjustment</code>	Tracking adjustment.
<code>kATSULayoutOperationPostLayoutAdjustment</code>	Applies a layout operation after ATSUI has finished its layout operations.

After you've installed the callback, you use ATSUI as you normally would to draw text. ATSUI triggers your callback each time the layout operation you specified is invoked.

You'll see specific examples of how to write and install callbacks that manipulate the final layout in the following sections:

- [“Extending the Space Between Glyphs”](#) (page 117). This example shows how to obtain the array of advance delta values for a line of text, and then to modify advance values to extend the space between the glyphs.
- [“Positioning Glyphs Along a Curve”](#) (page 120). This example shows how to retrieve ATS layout records, advance delta arrays, and baseline delta arrays, and then to use real position information to calculate advance and baseline delta values that achieve the desired layout.

Before you override any of ATSUI's layout operations, you should read the guidelines outlined in the following section.

Guidelines for Overriding Layout Operations

Follow these guidelines when you override ATSUI layout operations:

- You should modify advance delta values during ATSUI's layout process and not as a post layout operation. ATSUI uses advance delta values when it calculates the real position. The real position is available only post layout. So adjusting advance delta values post layout won't have an effect.
- You should modify the real position only at post layout.
- Ideally, if you modify the real position, you should update the advance delta values to reflect the real-position modifications.
- You should modify baseline delta values as a post-layout operation (`kATSULayoutOperationPostLayoutAdjustment`) or in the baseline operation (`kATSULayoutOperationBaselineAdjustment`). If you modify these values during any other operation, ATSUI's baseline operation can overwrite the values you modify.

Extending the Space Between Glyphs

You can extend the space between glyphs by providing values that adjust the positions of the glyph. To do so, you need to

- inform ATSUI that you want to override the justification layout operation
- obtain the advance delta array for the glyphs associated with the text layout object whose layout you want to modify
- adjust advance delta values for each glyph whose layout you want to modify

Depending on the advance delta values you provide, you'll get an effect similar to the bottom line shown in [Figure 6-1](#) (page 114). All glyphs except the first glyph in this line are drawn using the same advance delta value to achieve a uniform spacing. The advance delta values do not need to be the same for each glyph.

The following sections show how to extend the space between glyphs:

- [“Writing a Callback to Modify Advance Delta Values”](#) (page 118)
- [“Installing a Callback for a Justification Override Operation”](#) (page 119)

Writing a Callback to Modify Advance Delta Values

The callback shown in [Listing 6-1](#) (`MyStretchGlyphCallback`) performs one task—it modifies advance delta values. The effect is to extend the space between glyphs. A detailed explanation for each numbered line of code appears following the listing.

Listing 6-1 A callback that modifies advance delta values

```
static
OSStatus MyStretchGlyphCallback (
    ATSULayoutOperationSelector    iCurrentOperation,
    ATSULineRef                    iLineRef,
    UInt32                          iRefCon,
    void                            *iOperationExtraParameter,
    ATSULayoutOperationCallbackStatus *oCallbackStatus )    // 1
{
    OSStatus        status;
    Fixed           *myDeltaXArray;                          // 2
    ItemCount       myRecordArrayCount;
    ItemCount       myItems;
    Fixed           myStretchFactor;

    status = ATSUDirectGetLayoutDataArrayPtrFromLineRef ( iLineRef,
        kATSUDirectDataAdvanceDeltaFixedArray,
        true,
        (void **) &myDeltaXArray,
        &myRecordArrayCount);                               // 3
    require_noerr (status, StretchGlyphCallback_err);

    myStretchFactor = (Fixed) ((gFontSize*2) << 16);       // 4
    for (myItems = 1; myItems < myRecordArrayCount; myItems++ ) // 5
    {
        myDeltaXArray[myItems] += myStretchFactor;
    };
    status = ATSUDirectReleaseLayoutDataArrayPtr ( iLineRef,
        kATSUDirectDataAdvanceDeltaFixedArray,
        (void **) &myDeltaXArray );                       // 6

StretchGlyphCallback_err:
    *oCallbackStatus = kATSULayoutOperationCallbackStatusHandled; // 7
    return status;
}
```

Here's what the code does:

1. Provides the parameters specified by the callback definition. As you'll see, this callback uses only two of the parameters: `iLineRef` and `oCallbackStatus`.
2. Declares an array for the advance delta values.
3. Calls the function `ATSUDirectGetLayoutDataArrayPtrFromLineRef` to obtain the advance delta values for the line specified by `iLineRef`. This function returns the data pointer specified by the `iDataSelector` parameter (in this case, `kATSUDirectDataAdvanceDeltaFixedArray`) for the line specified by `iLineRef`.

You should pass `true` for the `iCreate` parameter. If the requested array isn't referenced by `iLineRef`, ATSUI creates and returns a zero-filled array. If the requested array has already been created, ATSUI returns the array with the current advance delta values. On output, `myRecordArrayCount` specifies the number of items in the array.
4. Sets a stretch-factor value, based on the current font size, to be used for the advance delta.
5. Iterates through the advance delta array, assigning a value to each element in the array.
6. Releases the data pointer (`myDeltaXArray`) obtained by calling the function `ATSUDirectGetLayoutDataArrayPtrFromLineRef`. You must release this data pointer by calling the function `ATSUDirectReleaseLayoutDataArrayPtr`. Releasing the array signals ATSUI that you are done with the data and that ATSUI can merge values you set with those set in the font.
7. Returns the status `kATSULayoutOperationCallbackStatusHandled` to indicate the layout operation is handled successfully.

Installing a Callback for a Justification Override Operation

The `MyInstallStretchGlyphCallback` function in “A function that installs a justification override callback” sets up ATSUI to call `MyStretchGlyphCallback` each time a justification operation needs to be performed on the text associated with the specified text layout object. A detailed explanation for each numbered line of code appears following the listing.

Listing 6-2 A function that installs a justification override callback

```
OSStatus MyInstallStretchGlyphCallback (ATSUTextLayout myTextLayout)
{
    OSStatus          status = noErr;
    ATSULayoutOperationOverrideSpecifier  myOverrideSpec;           // 1
    ATSUAttributeTag  theTag;
    ByteCount         theSize;
    ATSUAttributeValuePtr  theValue;

    myOverrideSpec.operationSelector =
        kATSULayoutOperationJustification;                       // 2
    myOverrideSpec.overrideUPP = MyStretchGlyphCallback;

    theTag = kATSULayoutOperationOverrideTag;                    // 3
    theSize = sizeof (ATSULayoutOperationOverrideSpecifier);
    theValue = &myOverrideSpec;

    status = ATSUSetLayoutControls (myTextLayout, 1,
                                   &theTag, &theSize, &theValue); // 4
}
```

```

    require_noerr (status, InstallStrechGlyphCallback_err );

InstallStrechGlyphCallback_err:
    return status;
}

```

Here's what the code does:

1. Declares an override specification. This structure contains a selector for a layout operation and a universal procedure pointer to the callback you supply.
2. Assigns the justification selector as the operation selector and the `MyStretchGlyphCallback` callback as the callback to handle justification.
3. Sets up a triple (tag, size, value) for the layout attribute. In this case, the layout attribute is the override specification.
4. Calls the function `ATSUSetLayoutControls` to associate the override specification with the text layout object.

Positioning Glyphs Along a Curve

The text in Figure 6-4 shows glyphs whose baseline delta values are adjusted to track a sine curve. To achieve the smoothest look possible, the baseline delta values are calculated using the real position of the glyphs. In addition, the advance width for each glyph is adjusted.

To position glyphs along a curve, you need to do the following:

- Inform ATSUI that you want to provide a post-layout operation.
- Obtain the ATS layout record, advance delta array, and the baseline delta array for the glyphs associated with the text layout object whose layout you want to modify. The ATS layout record contains the real position of the glyph.
- Adjust the advance delta and baseline delta values for each glyph such that the values track the specified curve.

Note: When you position text along a curve, you must obtain and lay out the glyphs in small groups. For arbitrary Unicode text, you should consider international issues when you break the glyphs into small groups. For example, breaking a cursive script, such as Arabic, in the middle of a ligature or between two cursively connected glyphs gives the wrong appearance in the drawn output.

The next sections discuss the specific tasks you need to perform to position glyphs along a curve:

- [“Writing the Callback to Modify Baseline Delta Values”](#) (page 121)
- [“Installing a Callback for a Post-Layout Operation”](#) (page 123)

Figure 6-4 Text positioned along a sine curve

Writing the Callback to Modify Baseline Delta Values

The callback (`MySineCurveGlyphCallback`) shown in Listing 6-3 modifies advance and delta values so that glyphs are positioned along a sine curve. A detailed explanation for each numbered line of code appears following the listing.

Listing 6-3 A callback that positions glyphs along a sine curve

```
static
OSStatus MySineCurveGlyphCallback(
    ATSULayoutOperationSelector    iCurrentOperation,
    ATSULineRef                    iLineRef,
    UInt32                          iRefCon,
    void                            *iOperationExtraParameter,
    ATSULayoutOperationCallbackStatus *oCallbackStatus )
{
    OSStatus          status = noErr;
    Fixed             *deltaYArray;
    Fixed             *deltaXArray;
    Fixed             positionDifference;
    ATSLayoutRecord *layoutRecordArray;
    ItemCount         recordArrayCount;
    ItemCount         i;
    Fixed             scaleFactor = 0;
    float             amplitude;

    status = ATSUDirectGetLayoutDataArrayPtrFromLineRef (iLineRef,
        kATSUDirectDataLayoutRecordATSLayoutRecordCurrent,
        false,
        (void **) &layoutRecordArray,
        &recordArrayCount ); // 1
    require_noerr (status, GlyphWaveCallback_err);

    status = ATSUDirectGetLayoutDataArrayPtrFromLineRef (iLineRef,
        kATSUDirectDataAdvanceDeltaFixedArray,
        true,
        (void **) &deltaXArray,
        &recordArrayCount ); // 2
    require_noerr (status, GlyphWaveCallbackDeltaXArray_err);

    status = ATSUDirectGetLayoutDataArrayPtrFromLineRef (iLineRef,
        kATSUDirectDataBaselineDeltaFixedArray,
        true,
        (void **) &deltaYArray,
        &recordArrayCount ); // 3
    require_noerr (status, SineCurveGlyphCallbackDeltaYArray_err);

    for ( i = 1; i < recordArrayCount; i++ ) // 4
```

```

    {
        positionDifference = (layoutRecordArray[i].realPos -
                             layoutRecordArray[i-1].realPos);
        if (positionDifference > scaleFactor)
            scaleFactor = positionDifference;
    }

    amplitude = FixedToFloat( scaleFactor ); // 5

    for ( i = 1; i < recordArrayCount - 1; i++ ) // 6
    {
        positionDifference = scaleFactor - (layoutRecordArray[i].realPos -
                                             layoutRecordArray[i-1].realPos); // 7

        layoutRecordArray[i].realPos += positionDifference; // 8
        deltaXArray[i-1] += positionDifference; // 9

        deltaYArray[i] = FloatToFixed ( sinf ( i ) * amplitude ); // 10
    };

    ATSUDirectReleaseLayoutDataArrayPtr (iLineRef,
                                          kATSUDirectDataBaselineDeltaFixedArray,
                                          (void **) &deltaYArray ); // 11

SineCurveGlyphCallbackDeltaYArray_err:

    ATSUDirectReleaseLayoutDataArrayPtr (iLineRef,
                                          kATSUDirectDataAdvanceDeltaFixedArray,
                                          (void **) &deltaXArray ); // 12

SineCurveGlyphCallbackDeltaXArray_err:

    ATSUDirectReleaseLayoutDataArrayPtr (iLineRef,
                                          kATSUDirectDataLayoutRecordATSLayoutRecordCurrent,
                                          (void **) &layoutRecordArray ); // 13

SineCurveGlyphCallback_err:

    *oCallbackStatus = kATSULayoutOperationCallbackStatusHandled; // 14

    return status;

}

```

Here's what the code does:

1. Calls the function `ATSUDirectGetLayoutDataArrayPtrFromLineRef` with the layout record data selector to obtain the ATS layout records for each glyph on the line. A layout record contains the glyph real position. Calculating a sine curve based on the real positions of the glyphs results in a much smoother appearance.
2. Calls the function `ATSUDirectGetLayoutDataArrayPtrFromLineRef` with the advance delta data selector to obtain the advance delta array for the glyphs on the line.
3. Calls the function `ATSUDirectGetLayoutDataArrayPtrFromLineRef` with the baseline delta data selector to obtain the baseline delta array for the glyphs on the line.

4. Iterates through the layout records to find the largest positional difference in the line. This difference will be used to set a scaling factor.
5. Sets the amplitude for the sine curve calculation to the scale factor.
6. Iterates through the glyph information to set the advance and baseline delta factors, taking into account the real position of the glyph.
7. Calculates a positional difference. This will be used to impose a fixed width on each glyph that takes the glyph's point size into account.
8. Adds the positional difference to the real position value for the layout record array.
9. Adds the positional difference to the advance delta value.
10. Calculates the baseline delta using the previously calculated amplitude value and calling the function `sinf`.
11. Releases the baseline delta array by calling the function `ATSUDirectReleaseLayoutDataArrayPtr` with the baseline delta data selector.
12. Releases the advance delta array by calling the function `ATSUDirectReleaseLayoutDataArrayPtr` with the advance delta data selector.
13. Releases the layout record array by calling the function `ATSUDirectReleaseLayoutDataArrayPtr` with the ATS layout record data selector.
14. Returns a status value to ATSUI to indicate the layout operation is handled successfully.

Installing a Callback for a Post-Layout Operation

The `MyInstallSineCurveGlyphCallback` function in Listing 6-4 sets up ATSUI to call `MySineCurveGlyphCallback`. ATSUI triggers the callback for each line of text associated with the specified text layout object. Because the callback does post-layout processing, ATSUI invokes the callback at the end of its layout operations for the line. A detailed explanation for each numbered line of code appears following the listing.

Listing 6-4 Installing a callback for a post-layout override operation

```
OSStatus MyInstallSineCurveGlyphCallback (ATSUTextLayout textLayout )
{
    OSStatus                status = noErr;
    ATSULayoutOperationOverrideSpecifier myOverrideSpec;           // 1
    ATSUAttributeTag        theTag;
    ByteCount               theSize;
    ATSUAttributeValuePtr   thePtr;

    myOverrideSpec.operationSelector =
        kATSULayoutOperationPostLayoutAdjustment;               // 2
    myOverrideSpec.overrideUPP = MySineCurveGlyphCallback;

    attrTag = kATSULayoutOperationOverrideTag;                   // 3
    attrSize = sizeof (ATSULayoutOperationOverrideSpecifier);
    attrPtr = &myOverrideSpec;
}
```

```

    status = ATSUSetLayoutControls (textLayout, 1,
                                   &theTag, &theSize, &thePtr );           // 4
    require_noerr (status, InstallSineCurveGlyphCallback_err );

InstallSineCurveGlyphCallback_err:

    return status;
}

```

Here's what the code does:

1. Declares an override specification. This structure contains a selector for a layout operation and a universal procedure pointer to the callback you supply.
2. Assigns the post-layout selector as the operation selector and the `MySineCurveGlyphCallback` callback as the callback to handle justification.
3. Sets up a triple (tag, size, value) for the layout attribute. In this case, the layout attribute is the override specification.
4. Calls the function `ATSUSetLayoutControls` to associate the override specification with the text layout object.

Retrieving and Drawing Glyph Outlines

ATSUI provides several direct-access functions that retrieve **glyph outlines**—the curves that make up the shape of the glyph. You should obtain glyph outlines only when you want to handle drawing the glyph instead of letting ATSUI do it. You can make modifications to the glyph data before you draw the glyph.

Using direct-access functions, you can do the following:

- Determine the native curve type of a font. TrueType fonts use quadratic curves while Type 1 (PostScript) fonts use cubic curves.
- Retrieve a cubic or quadratic glyph path—that is, the segments that make up the shape of a glyph.

Determining the native curve type of a font is easy, just call the function `ATSUGetNativeCurveType`.

Obtaining cubic or quadratic paths for a glyph requires you to use the functions `ATSUGlyphGetCubicPaths` or `ATSUGlyphGetQuadraticPaths`, respectively. However, you can call the function `ATSUGlyphGetQuadraticPaths` for fonts whose native curve type is cubic, and you can call `ATSUGlyphGetCubicPaths` for a font whose native curve type is quadratic. In each case, the font's curves are converted to the format specified by the function.

The curves returned by the functions are those that have been modified by hints present in the font. If you need unhinted outlines, you should use a very large point size (for example, 1000 points) and scale down the result. Alternatively, you can set the `ATSUStyleRenderingOptions` of the style object (`ATSUStyle`) to 0.

The coordinates returned for the curves and lines use the QuickDraw coordinate system. The (0,0) coordinate in QuickDraw is located in the upper-left corner. Quartz 2D has its (0,0) coordinate in the lower-left corner. If you are drawing into a Quartz context, you need to transform the QuickDraw coordinates accordingly.

If you want to handle drawing glyphs that use quadratic curves, you call the function `ATSUGlyphGetQuadraticPaths`. This function obtains the glyph segments for a glyph and then calls your callback functions for drawing the glyph. You must supply these universal procedure pointers (UPPs) to the function `ATSUGlyphGetQuadraticPaths[]`. This function obtains the glyph segments for a glyph and then calls your callback functions for drawing the glyph. You must supply these universal procedure pointers (UPPs) to the function `[ATSUGlyphGetQuadraticPaths`:

- `ATSQuadraticNewPathUPP`, a pointer to your callback to handle the new-path operation
- `ATSQuadraticLineUPP`, a pointer to your callback to handle the line-to operation
- `ATSQuadraticCurveUPP`, a pointer to your callback to handle the curve-to operation
- `ATSQuadraticClosePathUPP`, a pointer to your callback to handle the close-path operation

Similarly, if you want to handle drawing glyphs that use cubic curves, you call the function `ATSUGlyphGetCubicPaths`. This function obtains the glyph segments for a glyph and then calls your callback functions for drawing the glyph. You must supply these UPPs to the function `ATSUGlyphGetCubicPaths`:

- `ATSCubicMoveToUPP`, a pointer to your callback to handle the move-to operation
- `ATSCubicLineToUPP`, a pointer to your callback to handle the line-to operation
- `ATSCubicCurveToUPP`, a pointer to your callback to handle the curve-to operation
- `ATSCubicClosePathUPP`, a pointer to your callback to handle the close-path operation

Note that for cubic paths, the starting position for each curve or line is implicit from the current pen position. The start of a path is also implicit and is signaled by the move to establish the initial pen position.

Listing 6-5 shows code that sets up the four quadratic curve callbacks that handle glyph drawing for glyphs whose curve type is quadratic. Listing 6-6 (page 128) shows a function (`MyDrawQuadratics`) that creates universal procedure pointers to the callbacks and uses the function `ATSUGlyphGetQuadraticPaths`. A detailed explanation for each numbered line of code in a listing appears following each listing. The code for using cubic curve callbacks to handle glyph drawing is similar to that shown in Listing 6-5 and Listing 6-6, except that you would use the function `ATSUGlyphGetCubicPaths` and supply to it your cubic callbacks.

Listing 6-5 Setting up the quadratic curve callbacks

```
OSStatus MyQuadraticLineProc (const Float32Point *pt1,
                             const Float32Point *pt2,
                             void *callBackDataPtr) // 1
{
    float x1 = ((MyCallbackData *)callBackDataPtr)->origin.x + pt1->x; // 2
    float y1 = ((MyCallbackData *)callBackDataPtr)->origin.y + pt1->y;
    float x2 = ((MyCallbackData *)callBackDataPtr)->origin.x + pt2->x;
    float y2 = ((MyCallbackData *)callBackDataPtr)->origin.y + pt2->y;

    y1 = ((MyCallbackData *)callBackDataPtr)->windowHeight - y1; // 3
    y2 = ((MyCallbackData *)callBackDataPtr)->windowHeight - y2;
    if ( ((MyCurveCallbackData *)callBackDataPtr)->first ) // 4
```

```

        {
            CGContextMoveToPoint (gContext, x1, y1);
            ((MyCurveCallbackData *)callBackDataPtr)->first = false;
        }
        CGContextAddLineToPoint (context, x2, y2); // 5

    return noErr; // 6
}

OSStatus MyQuadraticCurveProc (const Float32Point *pt1,
                               const Float32Point *controlPt,
                               const Float32Point *pt2,
                               void *callBackDataPtr) // 7
{
    float x1 = ((MyCallbackData *)callBackDataPtr)->origin.x + pt1->x; // 8
    float y1 = ((MyCallbackData *)callBackDataPtr)->origin.y + pt1->y;
    float x2 = ((MyCallbackData *)callBackDataPtr)->origin.x + pt2->x;
    float y2 = ((MyCallbackData *)callBackDataPtr)->origin.y + pt2->y;
    float cpx = ((MyCallbackData *)callBackDataPtr)->origin.x +
                controlPt->x;
    float cpy = ((MyCallbackData *)callBackDataPtr)->origin.y +
                controlPt->y;

    y1 = ((MyCallbackData *)callBackDataPtr)->windowHeight - y1; // 9
    y2 = ((MyCallbackData *)callBackDataPtr)->windowHeight - y2;
    cpy = ((MyCallbackData *)callBackDataPtr)->windowHeight - cpy;
    if ( ((MyCurveCallbackData *)callBackDataPtr)->first ) // 10
    {
        CGContextMoveToPoint(gContext, x1, y1);
        ((MyCurveCallbackData *)callBackDataPtr)->first = false;
    }

    CGContextAddQuadCurveToPoint (context, cpx, cpy, x2, y2); // 11

    return noErr; // 12
}

OSStatus MyQuadraticNewPathProc (void * callBackDataPtr) // 13
{
    ((MyCurveCallbackData *)callBackDataPtr)->first = true; // 14

    return noErr; // 15
}

OSStatus MyQuadraticClosePathProc (void * callBackDataPtr)
{
    ((MyCurveCallbackData *)callBackDataPtr)->first = true; // 16
    return noErr; // 17
}

```

Here's what the code does:

1. Sets up the parameters that are passed to your callback for drawing a line. ATSUI passes two points that define a line and a pointer to any data your callback needs. You pass the pointer to your callback data to ATSUI when you call the function `ATSUGlyphGetQuadraticPaths`. See [Listing 6-6](#) (page 128).

2. Modifies the coordinate values of the points passed in by ATSUI. The four lines of code here add values supplied by the callback data pointer. These values are spacing adjustments that take into account the window height. When you write your callback, you would modify the values appropriately for your application.
3. Transforms the y-coordinate values from QuickDraw coordinates to Quartz coordinates. If you are using a Quartz context, you must perform this transformation because ATSUI always passes coordinates as QuickDraw coordinates.
4. Checks to see if this is the first point in the curve. If it is, calls the Quartz 2D function `CGContextMoveToPoint` to begin drawing at the specified coordinates.
5. Calls the Quartz 2D function `CGContextAddLineToPoint` to draw a straight line segment from the current point to the specified point.
6. Returns a result code that indicates whether or not your callback executed successfully. If your callback returns any value other than 0, the function `ATSGlyphGetQuadraticPaths` stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.
7. Sets up the parameters that are passed to your callback for drawing a curve. ATSUI passes the two end points and a control point that define the curve along with a pointer to any data your callback needs. You pass the pointer to your callback data to ATSUI when you call the function `ATSUGlyphGetQuadraticPaths`. See [Listing 6-6](#) (page 128).
8. Modifies the coordinate values of the end points and control point passed in by ATSUI. The six lines of code here add origin values supplied by the callback data pointer. When you write your callback, you would modify the values of the end points and control point appropriately for your application.
9. Transforms the y-coordinate values from QuickDraw coordinates to Quartz coordinates. If you are using a Quartz context, you must perform this transformation because ATSUI always passes coordinates as QuickDraw coordinates.
10. Checks to see if this is the first point in the curve. If it is, calls the Quartz 2D function `CGContextMoveToPoint` to begin drawing at the specified location.
11. Calls the Quartz 2D function `CGContextAddQuadCurveToPoint` to draw a quadratic Bézier curve from the current point, using the control point and end point you specify.
12. Returns a result code that indicates whether or not your callback executed successfully. If your callback returns any value other than 0, the function `ATSGlyphGetQuadraticPaths` stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.
13. Sets up the parameter that is passed to your callback for establishing a new path. ATSUI passes a pointer to any data your callback needs. You pass the pointer to your callback data to ATSUI when you call the function `ATSUGlyphGetQuadraticPaths`. See [Listing 6-6](#) (page 128).
14. Sets the flag that indicates the beginning of a curve segment to `true`. This flag is used by the functions `MyQuadraticLineProc` and `MyQuadraticCurveProc`.
15. Returns a result code that indicates whether or not your callback executed successfully. If your callback returns any value other than 0, the function `ATSGlyphGetQuadraticPaths` stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.
16. Sets the flag that indicates the beginning of a curve segment to `true`. This flag is used by the functions `MyQuadraticLineProc` and `MyQuadraticCurveProc`.

17. Returns a result code that indicates whether or not your callback executed successfully. If your callback returns any value other than 0, the function `ATSGlyphGetQuadraticPaths` stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.

After you have written callbacks to handle drawing operations for a quadratic curve (as shown in Listing 6-5), you need to write a function that creates universal procedure pointers (UPPs) for each callback and then pass the UPPs to the function `ATSUGlyphGetQuadraticPaths`. Listing 6-6 shows a function that sets up UPPs, obtains glyph data, and calls the function `ATSUGlyphGetQuadraticPaths` to draw each glyph in a text run. A detailed explanation for each numbered line of code appears following the listing.

Listing 6-6 A function that draws glyph outlines using quadratic curve data

```
void MyDrawQuadratics (ATSUTextLayout iLayout,
                      ATSUStyle iStyle,
                      UniCharArrayOffset start,
                      UniCharCount length,
                      Fixed penX,
                      Fixed penY,
                      float windowHeight) // 1
{
    ATSLayoutRecord *layoutRecords;
    ItemCount numRecords;
    Fixed *deltaYs;
    ItemCount numDeltaYs;
    ATSQuadraticNewPathUPP newPathProc;
    ATSQuadraticLineUPP lineProc;
    ATSQuadraticCurveUPP curveProc;
    ATSQuadraticClosePathUPP closePathProc;
    MyCallbackData data;
    OSStatus status;
    int i;

    newPathProc = NewATSQuadraticNewPathUPP (MyQuadraticNewPathProc); // 2
    lineProc = NewATSQuadraticLineUPP (MyQuadraticLineProc); // 3
    curveProc = NewATSQuadraticCurveUPP (MyQuadraticCurveProc); // 4
    closePathProc = NewATSQuadraticClosePathUPP (MyQuadraticClosePathProc); // 5

    ATSUDirectGetLayoutDataArrayPtrFromTextLayout (iLayout,
                                                    start,
                                                    kATSUDirectDataLayoutRecordATSLayoutRecordCurrent,
                                                    (void *) &layoutRecords,
                                                    &numRecords); // 6
    ATSUDirectGetLayoutDataArrayPtrFromTextLayout (iLayout,
                                                    start,
                                                    kATSUDirectDataBaselineDeltaFixedArray,
                                                    (void *) &deltaYs,
                                                    &numDeltaYs); // 7
    CGContextBeginPath (gContext); // 8

    data.windowHeight = windowHeight; // 9
    for (i=0; i < numRecords; i++) // 10
    {
        data.origin.x = Fix2X(penX) +
                      Fix2X(layoutRecords[i].realPos); // 11
        if (deltaYs == NULL) // 12
```



```

        data.origin.y = Fix2X(penY);
    else
        data.origin.y = Fix2X(penY) - Fix2X(deltaYs[i]);
    data.first = true; // 13
    if (layoutRecords[i].glyphID != kATSDeletedGlyphcode) // 14
    {
        ATSUGlyphGetQuadraticPaths (iStyle,
            layoutRecords[i].glyphID,
            newPathProc,
            lineProc,
            curveProc,
            closPathProc,
            &data,
            &status);
    }
}

CGContextClosePath(gContext); // 15
CGContextDrawPath(gContext, kCGPathStroke); // 16

if (deltaYs != NULL) // 17
    ATSUDirectReleaseLayoutDataArrayPtr (NULL,
        kATSUDirectDataBaselineDeltaFixedArray,
        (void *) &deltaYs);
ATSUDirectReleaseLayoutDataArrayPtr (NULL,
    kATSUDirectDataLayoutRecordATSLayoutRecordCurrent,
    (void *) &layoutRecords); // 18

DisposeATSQuadraticNewPathUPP (newPathProc); // 19
DisposeATSQuadraticLineUPP (lineProc);
DisposeATSQuadraticCurveUPP (curveProc);
DisposeATSQuadraticClosePathUPP (closePathProc);
}

```

Here's what the code does:

1. Sets up the parameters needed by this function:
 - a valid text layout
 - the style object associated with the text layout
 - the starting offset for the text run that will be processed by this function
 - the length of the text run that will be processed by this function
 - the x-coordinate for the pen's starting location
 - the y-coordinate for the pen's starting location
 - the height of the window into which the text will be drawn
2. Calls the function `NewATSQuadraticNewPathUPP` to create a UPP for the `MyQuadraticNewPathProc` callback created in [Listing 6-5](#) (page 125).
3. Calls the function `NewATSQuadraticLineUPP` to create a UPP for the `MyQuadraticLineProc` callback created in [Listing 6-5](#) (page 125).

4. Calls the function `NewATSQuadraticCurveUPP` to create a UPP for the `MyQuadraticCurveProc` callback created in [Listing 6-5](#) (page 125).
5. Calls the function `NewATSQuadraticClosePathUPP` to create a UPP for the `MyQuadraticClosePathProc` callback created in [Listing 6-5](#) (page 125).
6. Calls the function `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` to obtain the glyph IDs and the real position for the glyphs associated with the text layout.
7. Calls the function `ATSUDirectGetLayoutDataArrayPtrFromTextLayout` to obtain the baseline delta values (if any) for the glyphs associated with the text layout.
8. Calls the Quartz 2D function `CGContextBeginPath` to begin a path for the glyph outlines. A Quartz graphics context can have only a single path in use at any time. If the context already contains a path when you call `CGContextBeginPath`, Quartz replaces the previous current path with the new path, discarding the old path and any data associated with it.
9. Assigns the window height to the data structure that will be passed to the callbacks. This is used to transform the y-coordinate from QuickDraw coordinate space to Quartz coordinate space.
10. Begins a loop over all the glyphs in the text run.
11. Assigns an adjusted x-coordinate to the data structure that will be passed to the callbacks. The x-coordinate is the position for the beginning of the line added to the real position.
12. Checks to see if the baseline delta array is `NULL`. If the array is `NULL`, the y-coordinate is the vertical position of the line; otherwise, the y-coordinate is the vertical position of the line added to the baseline delta value.
13. Sets the flag that indicates the start of a curve segment to `true`. This flag is used by the callback functions `MyQuadraticLineProc` and `MyQuadraticCurveProc`. See [Listing 6-5](#) (page 125).
14. Checks whether the glyph ID is not that of a deleted glyph. Deleted glyphs should not be drawn. If the glyph should be drawn, calls the function `ATSUGlyphGetQuadraticPaths` to draw the glyphs using the callback functions specified by the UPPs passed to `ATSUGlyphGetQuadraticPaths`. This function also takes as parameters the style object associated with the text run, the glyph ID, a pointer to the data structure that will be passed to the callbacks, and a pointer to a status value. The status value is used to indicate the status of your callback functions. When a callback function returns any value other than 0, the `ATSUGlyphsGetQuadraticPaths` function stops parsing the path outline and returns the result `kATSOutlineParseAbortedErr`.
15. Calls the Quartz 2D function `CGContextClosePath` to close and terminate the current path.
16. Calls the Quartz 2D function `CGContextDrawPath` to paint a line along the current path.
17. Checks to see if the baseline delta array is `NULL`. If it is not, calls the function `ATSUDirectReleaseLayoutDataArrayPtr` to release the baseline delta array.
18. Calls the function `ATSUDirectReleaseLayoutDataArrayPtr` to release the layout record array.
19. Calls the appropriate ATSUI functions to dispose of the four UPPs created previously.

ATSUI Implementation of the Unicode Specification

This appendix provides additional details about the implementation of the Unicode specification in ATSUI. ATSUI provides full layout support for Unicode 3.2 and supports text rendering for all the features required by scripts included with version 2.1 of the Unicode standard or later. It does not provide other Unicode-related text processing services such as date and time formatting, collation, or string matching. The ability of ATSUI to render Unicode text is limited only by the available fonts the user has installed.

The correct handling of many Unicode characters requires that the current font supports those characters properly. For example, correct ligature formation requires that the font supports those features using Apple Advanced Typography (AAT) tables. If there is more than one equivalent combining character sequence for a given glyph, the font is responsible for mapping all such sequences to the correct glyph. For example, ATSUI does not automatically support conjoining Jamo in a Korean font that specifies precomposed glyphs only. For more details on the required AAT font tables and tools for creating them, see the description of Apple Advanced Typography at <http://developer.apple.com/fonts/>.

Unsupported Control Characters

ATSUI version 1.1 and later does not support the following control characters:

- U+00AD (soft hyphen)
- U+206A (inhibit symmetric swapping)
- U+206B (activate symmetric swapping)
- U+206C (inhibit Arabic form shaping)
- U+206D (activate Arabic form shaping)
- U+206E (national digit shapes)
- U+206F (normal digit shapes)
- Use of U+005F (spacing underscore or low line) to underline other characters

You can, however, achieve similar effects achieved by these control characters by setting certain style attributes. (See “[ATSUI Style and Text Layout Objects](#)” (page 37) for information on style objects and style attributes.) In addition, ATSUI currently treats the following characters as hard line breaks:

- U+000A (line feed)
- U+000C (form feed)
- U+000D (carriage return)
- U+2028 (line separator)
- U+2029 (paragraph separator)

ATSUI fully renders nonspacing marks, though correct font tables are required to render and process nonspacing marks correctly. To locate text element boundaries, ATSUI defines a cluster as a run consisting of a base character plus zero or more nonbase characters, where a base character is defined as one whose combining class is 0 and whose glyph is not deleted. Whether or not a set of characters is a cluster is also dependent upon the behavior of the specific font you are using with those characters.

ATSUI uses the Unicode Utilities function `UCFindTextBreak` to determine the boundaries of text elements such as character clusters, words, and lines. For example, character clusters as determined by `UCFindTextBreak` include the following:

- a run consisting of a base character plus zero or more combining characters
- a sequence of conjoining Jamo that would normally be displayed as a single composed Hangul character

However, ATSUI extends the `UCFindTextBreak` function's notion of text boundaries so that they may also be affected by rendering behavior, and may thus be dependent on the behavior of the specific font that is being used to render the text.

ATSUI fully supports the Unicode bidirectional algorithm, including the bidirectional ordering codes. Correct bidirectional processing requires that the font have the correct glyph properties set (for example, mirrored punctuation). Other characters that require font support for correct processing include invisible characters such as U+FEFF (zero-width no-break space). There are some characters that ATSUI maps to either a zero-width glyph or a nonmarking return.

Surrogates

Unicode code points, or scalar values, range from 0 to U+10FFFF (excluding the surrogate range 0xD800 to 0xDFFF and certain disallowed values such as 0xFFFF). There are three encoding forms for Unicode: UTF-32, UTF-16, and UTF-8. UTF-32 uses 32-bit code units, and it can represent Unicode scalar values directly. UTF-16 uses 16-bit code units, and it is the encoding form used by ATSUI. Unicode scalar values in the range 0 to U+FFFF, the Basic Multilingual Plane (BMP), are represented in UTF-16 by a single code unit with the same numeric value.

Unicode values in the range U+10000 to U+10FFFF are represented in UTF-16 by a pair of UTF-16 code units: A high surrogate in the range 0xD800 to 0xDBFF followed by a low surrogate in the range 0xDC00 to 0xDFFF (the mapping between Unicode scalar values above U+FFFF and surrogate pairs is described in the Unicode Standard).

A surrogate pair should generally be treated as a single character for such editing operations as text insertion, deletion, selection, hit-testing, and cursor movement. In other areas such as ligatures or accented letters, you may treat surrogates as a single or multiple entities.

Until version 3.1 of the Unicode Standard, there were no characters encoded outside the BMP—that is, no characters with scalar values above U+FFFF, and thus no characters that required surrogate pairs in UTF-16. However, Unicode 3.1 encodes a number of characters that require surrogates, including many CJK ideographs that are in the Hiragino Japanese fonts included with Mac OS X. ATSUI and the Unicode Utilities provide support for handling such non-BMP characters as surrogate pairs in UTF-16.

Character Properties

The character properties used by ATSUI support the standard characters in Unicode 3.2 plus the characters that Apple has defined in the corporate private use zone. ATSUI requires that white space and symmetric swapping-related properties be correctly set in the font. Characters not defined in the standard are assumed to be direction-neutral.

Font Features

This appendix describes font feature types and the selectors available for each feature type. In many cases, it also provides illustrations that show the effect of enabling a font feature or set of features. Before you read this appendix, you should be familiar with the font feature concepts discussed in detail in Chapter 3, “ATSUI Style and Text Layout Objects” (page 37).

Font features are available for a font only if the font designer chooses to include them. Many of the font features described here are rarely available. You should check with the font provider to see what features, if any, are available for a specific font.

The constants that represent font feature types and selectors are declared in the header file `SFNTLayoutTypes.h`. When you use ATSUI to access and set font features, you must use the constants defined in this header file, which are described in this appendix. Font designers can define feature types at any time. For the most up-to-date list of font feature types and selectors you should check Apple’s font feature registry website:

<http://developer.apple.com/fonts/Registry/index.html>

All-Typographic Features Feature Type

The all-typographic features feature type (`kAllTypographicFeaturesType`) enables or disables all available typographic features at once. Table B-1 lists the selectors for this feature.

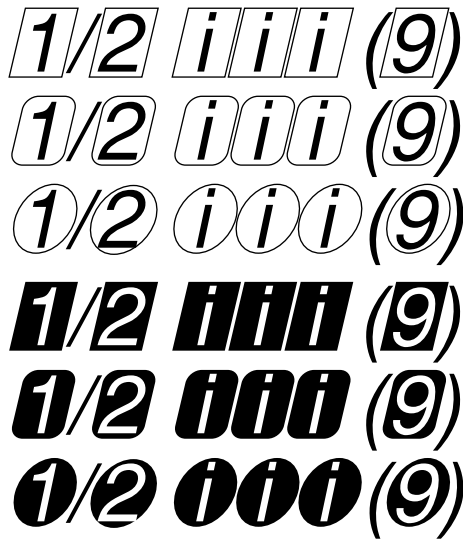
Table B-1 Selectors for the all-typographic features feature type

Feature selector	Description
<code>kAllTypographicFeaturesOnSelector</code>	Enables all typographic features. This is the default setting.
<code>kAllTypographicFeaturesOffSelector</code>	Disables all typographic features.

Annotation Feature Type

The annotation feature type (`kAnnotationType`) specifies annotations (or adornments) to basic letter shapes. For instance, most Japanese fonts include versions of numbers that are circled, are enclosed by parentheses, have periods after them, and so on. Figure B-1 shows some glyphs that are drawn with annotations.

Figure B-1 Glyphs drawn with annotations



Annotation is a noncontextual, exclusive feature type. Table B-2 lists the selectors for this feature.

Table B-2 Selectors for the annotation feature

Feature selector	Description
<code>kNoAnnotationSelector</code>	Specifies that characters should appear without annotation. This is the default setting.
<code>kBoxAnnotationSelector</code>	Specifies to use the forms of characters surrounded by a box cartouche. See line 1 in Figure B-1.
<code>kRoundedBox-AnnotationSelector</code>	Specifies to use the forms of characters surrounded by a box cartouche with rounded corners. See line 2 in Figure B-1.
<code>kCircleAnnotationSelector</code>	Specifies to use the forms of characters surrounded by a circle. For example, see Unicode characters U+3260 through U+326F. See line 3 in Figure B-1.
<code>kInvertedBox-AnnotationSelector</code>	Specifies to use the forms of characters surrounded by a box cartouche, but with white and black reversed. See line 4 in Figure B-1.
<code>kInvertedRoundedBox-AnnotationSelector</code>	Specifies to use the forms of characters surrounded by a box cartouche with rounded corners, but with white and black reversed. See line 5 in Figure B-1.
<code>kInvertedCircle-AnnotationSelector</code>	Specifies to use the forms of characters surrounded by a circle, but with white and black reversed. For example, see Unicode characters U+2776 through U+277F. See line 6 in Figure B-1.
<code>kParenthesis-AnnotationSelector</code>	Specifies to use the forms of characters surrounded by parentheses. For example, see Unicode characters U+2474 through U+2487.

Feature selector	Description
<code>kPeriodAnnotationSelector</code>	Specifies to use the forms of characters followed by a period. For example, see Unicode characters U+2488 through U+249B.
<code>kRomanNumeralAnnotationSelector</code>	Specifies to display the given characters in their Roman numeral form.
<code>kDiamondAnnotationSelector</code>	Specifies to display the text surrounded by a diamond.

Cursive Connection Feature Type

The cursive connection feature type (`kCursiveConnectionType`) is used for cursively connected scripts to specify whether or not cursive connections are to be used between glyphs. This feature type is required for Arabic, but may be supported by other scripts as well. Figure B-2 shows an example of cursive connection in a Roman font.

Figure B-2 Cursive connection in a Roman font

A B C s t u 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 noncontextual manner, or enable completely contextual, cursively connected letter forms (as in Arabic). Table B-3 lists the feature selectors for cursive connection. This is a contextual, exclusive feature type.

Table B-3 Selectors for the cursive connection feature type

Feature selector	Description
<code>kUnconnectedSelector</code>	Disables cursive connection. Selecting this for some scripts results in incorrect linguistic appearance.
<code>kPartiallyConnectedSelector</code>	Specifies predrawn letter forms that connect in a noncontextual manner.
<code>kCursiveSelector</code>	Specifies full contextual connection of letter forms. For Arabic fonts, this selector is set by default.

Character Alternatives Feature Type

The character alternatives feature type (`kCharacterAlternativesType`) specifies any font-specific set of alternate glyph forms. 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. Figure B-3 shows the character “g,” first drawn using the default glyph for the font, then drawn using an alternate glyph form provided by the font.

Figure B-3 The character “g” shown as two glyph forms



The character alternative feature type is a noncontextual, exclusive feature type. Table B-4 lists the only defined selector for this feature.

Table B-4 Selector for the character alternatives feature

Feature selector	Description
<code>kNoAlternativesSelector</code>	Specifies that no character alternatives are to be used. This is the default setting.

Character Shape Feature Type

The character shape feature type (`kCharacterShapeType`) is used when a single font contains different appearances for the same character shape, and these shapes are not traditionally treated as swashes. It is needed for languages such as Chinese that have both traditional and simplified character sets, as shown in Figure B-4. In fact, Chinese usually has several alternatives for traditional characters sets.

Figure B-4 Traditional and simplified versions of a Chinese character



Table B-5 lists the selectors for the character shape feature type. This is a noncontextual, exclusive feature type.

Table B-5 Selectors for the character shape feature

Feature selector	Description
<code>kTraditionalCharactersSelector</code>	Specifies to use traditional forms for characters.
<code>kSimplifiedCharactersSelector</code>	Specifies to use simplified forms for characters.
<code>kJIS1978CharactersSelector</code>	Specifies to use character shapes for Japanese characters as defined by the JIS (Japanese Industrial Standard) C 6226-1978 document.

Feature selector	Description
<code>kJIS1983CharactersSelector</code>	Specifies to use character shapes for Japanese characters as defined by the JIS (Japanese Industrial Standard) C 0208-1983 document.
<code>kJIS1990CharactersSelector</code>	Specifies to use character shapes for Japanese characters as defined by the JIS (Japanese Industrial Standard) C 0208-1990 document.
<code>kTraditionalAltOneSelector</code>	Specifies to use alternate set 1 of traditional forms for characters.
<code>kTraditionalAltTwoSelector</code>	Specifies to use alternate set 2 of traditional forms for characters.
<code>kTraditionalAltThreeSelector</code>	Specifies to use alternate set 3 of traditional forms for characters.
<code>kTraditionalAltFourSelector</code>	Specifies to use alternate set 4 of traditional forms for characters.
<code>kTraditionalAltFiveSelector</code>	Specifies to use alternate set 5 of traditional forms for characters.
<code>kExpertCharactersSelector</code>	Specifies to use expert forms of ideographs, such as those defined in the Fujitsu FMR character set.

CJK Italic Roman Feature Type

The CJK (Chinese, Japanese, Korean) italic Roman feature type (`kItalicCJKRomanType`) specifies whether or not to use italic Roman characters in a CJK font. Table B-6 lists the selectors for this feature.

Table B-6 Selectors for the CJK italic Roman feature type

Feature selector	Description
<code>kCJKItalicRomanOffSelector</code>	Specifies not to use italic Roman characters. This is the default setting.
<code>kCJKItalicRomanOnSelector</code>	Specifies to use italic Roman characters.
<code>kNoCJKItalicRomanSelector</code>	Specifies not to use italic Roman characters. This is deprecated; use <code>kCJKItalicRomanOffSelector</code> instead.
<code>kCJKItalicRomanSelector</code>	Specifies to use italic Roman characters. This is deprecated; use <code>kCJKItalicRomanOnSelector</code> instead.

CJK Roman Spacing Feature Type

The CJK (Chinese, Japanese, Korean) Roman spacing feature type (`kCJKRomanSpacingType`) specifies the spacing to use for Roman characters in a CJK font. Table B-7 lists the selectors for this feature.

Table B-7 Selectors for the CJK Roman spacing feature

Feature selector	Description
kHalfWidthCJKRomanSelector	Specifies to use half-width spacing.
kProportionalCJKRomanSelector	Specifies to use proportional-width spacing.
kFullWidthCJKRomanSelector	Specifies to use full-width spacing.

Design Complexity Feature Type

The design complexity feature type (`kDesignComplexityType`) controls the overall appearance of a font. It can be used to allow a single font to contain plain glyphs, italic glyphs, calligraphic chancery glyphs, and so forth. 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 such selectors as those shown in Table B-8. This is a noncontextual, exclusive feature type. Figure B-5 shows four levels of design complexity for a font.

Figure B-5 Four levels of design complexity

Four levels of design complexity
Four levels of design complexity
Four levels of design complexity
Four levels of design complexity

Table B-8 Selectors for the design complexity feature

Feature selector	Description
kDesignLevel1Selector	Specifies the basic glyph set. This should be available for any font that utilizes this feature type. This is the default setting.
kDesignLevel2Selector	Specifies an alternate glyph set, more complex than level 1.
kDesignLevel3Selector	Specifies an alternate glyph set, more complex than level 2.
kDesignLevel4Selector	Specifies an alternate glyph set, more complex than level 3.
kDesignLevel5Selector	Specifies an alternate glyph set, more complex than level 4.

Diacritical Marks Feature Type

The diacritical marks feature type (`kDiacriticsType`) controls how diacritical marks (that is, accent marks or applied vowels) appear in text. 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 B-6 shows an example of Hebrew text drawn with and without its diacritical marks.

Figure B-6 Hebrew text with diacritical marks shown (upper) and hidden (lower)

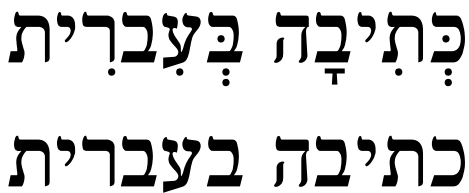
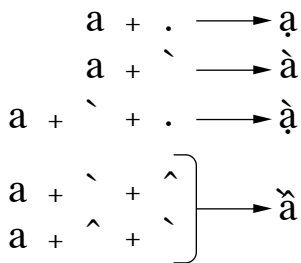


Figure B-7 shows an example of text drawn with and without its accents.

Figure B-7 Accented forms



If the font supports the diacritical marks feature type, you can specify that ATSUI should show, hide, or decompose diacritical marks, using the feature selectors shown in Table B-9. This is a contextual, exclusive feature type.

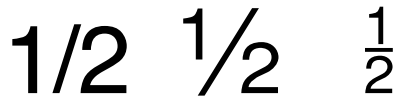
Table B-9 Selectors for the diacritical marks feature

Feature selector	Description
<code>kShowDiacriticsSelector</code>	Specifies to display diacritical marks normally; that is, attached to their base forms (glyphs) in the appropriate place. This is the default setting.
<code>kHideDiacriticsSelector</code>	Specifies not to display diacritical marks.
<code>kDecomposeDiacriticsSelector</code>	Specifies not to display marked glyphs as unmarked, followed by the accent ligatures as standalone glyphs.

Fractions Feature Type

The fractions feature type (`kFractionsType`) controls the selection and generation of fractions. For a font that supports the fractions feature type, you can select between two different types of automatic fraction generation—vertical and diagonal. Figure B-8 shows a fraction, drawn first without automatic fraction formation, then with the diagonal fractions selector, and finally with the vertical fractions selector.

Figure B-8 Fractions drawn with three different fractions feature selectors



1/2 1/2 ½

The feature selectors for the fractions feature type is shown in Table B-10. This feature is a contextual, exclusive feature type.

Table B-10 Selectors for the fractions feature

Feature selector	Description
<code>kNoFractionsSelector</code>	Specifies that fractions should not be formed automatically.
<code>kVerticalFractionsSelector</code>	Specifies replacement of slash-separated numeric sequences with pre-drawn fraction glyphs, if present in the font.
<code>kDiagonalFractionsSelector</code>	Specifies replacement of slash-separated numeric sequences with pre-drawn fraction glyphs, but fractions will be synthesized using superiors and inferiors (or special-purpose number and denominator forms) if present in the font.

Note: To use the automatic fraction-generation capability, make sure that the slash separating the numerator and denominator is the fraction slash (U+2044), not the normal slash character. Automatic fraction generation does not occur unless the slash is a fraction slash.

Ideographic Spacing Feature Type

The ideographic spacing feature type (`kIdeographicSpacingType`) specifies whether to use full-width or proportional-width text in the representation of Japanese kanji, Chinese hanzi, and Korean hanja (that is, ideographic characters). Table B-11 lists the selectors for this feature. This is a noncontextual, exclusive feature type.

Table B-11 Selectors for the ideographic spacing feature

Feature selector	Description
<code>kFullWidthIdeographicSelector</code>	Specifies to use full-width spacing for ideographs. This is the default setting.

Feature selector	Description
<code>kProportionalIdeographsSelector</code>	Specifies to use proportional spacing for ideographs.
<code>kHalfWidthIdeographsSelector</code>	Specifies to use half-width spacing for ideographs.

Kana Spacing Feature Type

The kana spacing feature type (`kKanaSpacingType`) specifies the widths to use for Japanese hiragana and katakana characters. Table B-12 lists the selectors for this feature. This is a noncontextual, exclusive feature type.

Table B-12 Selectors for the kana spacing feature

Feature selector	Description
<code>kFullWidthKanaSelector</code>	Specifies to use the full-width forms of kana. This is the default setting.
<code>kProportionalKanaSelector</code>	Specifies to use the proportional forms of kana.

Letter Case Feature Type

The letter case feature type (`kLetterCaseType`) is used to specify changes to letter case in scripts where case has meaning. This feature changes only the appearance of the letter; the string typed by the user remains invariant. If the string is typed using lowercase letters, the string remains lowercase.

Figure B-9 shows text in which different letter case feature selectors have been applied. The first line of text is drawn with no case conversion. The second line is drawn with the all caps feature enabled. The third line is drawn with the initial caps and small caps feature enabled.

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 uses them.

Figure B-9 Text drawn with different letter case feature selectors

Uppercase and lowercase

UPPERCASE AND LOWERCASE

UPPERCASE AND LOWERCASE

If the font supports the letter case feature type, you can select features that specify case changes such as those shown in Table B-13. This is an exclusive feature type.

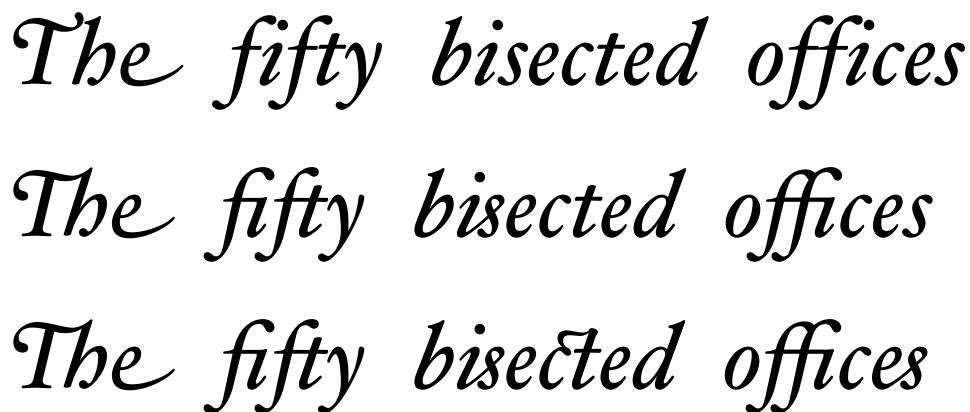
Table B-13 Selectors for the letter case feature

Feature selector	Description
<code>kUpperAndLowerCaseSelector</code>	Specifies no case conversion. This leaves letters in whichever case the user specifies. This is the default setting.
<code>kAllCapsSelector</code>	Converts all letters to uppercase. This feature is noncontextual.
<code>kAllLowerCaseSelector</code>	Converts all letters to lowercase. This feature is noncontextual.
<code>kInitialCapsSelector</code>	Converts the first letter of a word to uppercase and the remaining letters to lowercase. This feature is contextual.
<code>kInitialCapsAndSmall-CapsSelector</code>	Converts the first letter of a word to uppercase and the remaining letters to small caps. This feature is contextual.

Ligatures Feature Type

The ligatures feature type (`kLigaturesType`) specifies the use of linguistically required ligatures and a variety of optional ligatures. Figure B-10 shows several levels of ligature formation specified through ligature feature selectors. In the top line, the ligatures feature type is set to have required ligatures enabled. In the middle line, the value is set to have common ligatures enabled. In the bottom line, the value is set to have rare ligatures enabled.

Figure B-10 Levels of ligature formation controlled with ligature feature selectors



The figure displays three lines of cursive text: "The fifty bisected offices". The top line shows required ligatures enabled, the middle line shows common ligatures enabled, and the bottom line shows rare ligatures enabled.

Figure B-11 shows the results of selection and deselection of diphthong ligatures. In the top line, the ligatures feature type (`kLigaturesType`) is set to have diphthong ligatures enabled; in the bottom line, the ligatures feature type is set to have diphthong ligatures disabled.

Figure B-11 Use of diphthong ligatures

orthopædic encyclopædia
orthopaedic encyclopaedia

If the font supports the ligatures feature type, you can select features related to ligature formation, including those shown in Table B-14. This is a contextual, nonexclusive feature type.

Table B-14 Selectors for the ligatures feature type

Feature selector	Description
kRequiredLigatures-OnSelector	Allows the use of linguistically required ligatures (such as occur in Arabic or Hindi). This is the default setting.
kRequiredLigatures-OffSelector	Prevents the use of linguistically required ligatures (such as occur in Arabic or Hindi).
kCommonOnSelector	Allows the use of ligatures that are common, or that usually appear in well-set text, such as the “fi” and “fl” ligatures in English.
kCommonOffSelector	Prevents the use of ligatures that are common, or that usually appear in well-set text, such as the “fi” and “fl” ligatures in English.
kRareOnSelector	Allows the use of ligatures that are used less than those in the Common category, such as “ct” or “ss” ligatures.
kRareOffSelector	Prevents the use of ligatures that are used less than those in the Common category, such as “ct” or “ss” ligatures.
kLogosOnSelector	Allows the use of ligatures that represent logos; typically used for trademarks or other special display text. For example, typing the word “Apple” to display the Apple logo.
kLogosOffSelector	Prevents the use of ligatures that represent logos; typically used for trademarks or other special display text. For example, typing the word “Apple” to display the Apple logo.
kRebusPicturesOnSelector	Allows the use of pictures that represent words or syllables.
kRebusPicturesOffSelector	Prevents the use of pictures that represent words or syllables.
kDiphthongLigatures-OnSelector	Specifies to replace diphthong sequences, such as “AE” and “oe” with their equivalent ligatures.
kDiphthongLigatures-OffSelector	Specifies not to replace diphthong sequences, such as “AE” and “oe” with their equivalent ligatures.

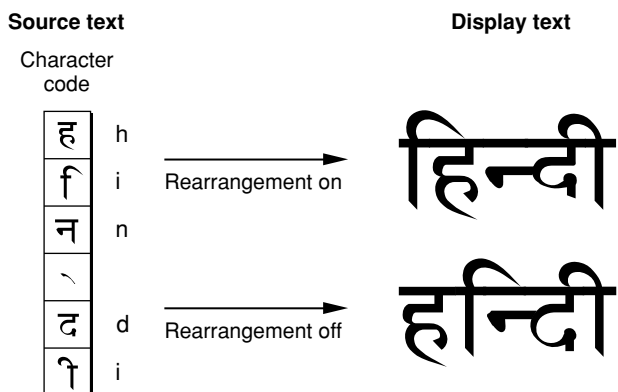
Feature selector	Description
kSquaredLigaturesOnSelector	Allows the use of ligatures in which the component letters are arranged in a lattice, such that the ligature fits into the space of a single letter. For examples, see Unicode characters U+3300 through U+3357 and U+337B through U+337F.
kSquaredLigaturesOffSelector	Prevents the use of ligatures in which 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	Allows the use of ligatures that are similar to squared ligatures, but abbreviated in form.
kAbbrevSquaredLigaturesOffSelector	Prevents the use of ligatures that are similar to squared ligatures, but abbreviated in form.
kSymbolLigaturesOnSelector	Allows the use of symbol ligatures.
kSymbolLigaturesOffSelector	Prevents the use of symbol ligatures.

Linguistic Rearrangement Feature Type

The linguistic rearrangement feature type (`kLinguisticRearrangementType`) specifies whether or not linguistic rearrangement of glyphs (Indic-style) is to be used. This feature is on by default for fonts that represent South Asian scripts. Linguistic rearrangement is different than the notion of linguistic reordering, which happens when text from predominantly left-to-right scripts (such as Latin) is mixed with text from predominantly right-to-left scripts (such as Hebrew).

Figure B-12 shows two examples of the display of the word “hindi”; first with linguistic rearrangement on and then with it off. In both cases, the source text is the same. However, when rearrangement is on, ATSUI rearranges the glyphs so they are displayed appropriately.

Figure B-12 The word “hindi” drawn with rearrangement turned on and off



In some cases, users may not always want linguistic rearrangement to occur, preferring instead to enter characters in an “already rearranged” 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, using the selectors shown in Table B-15. This is a contextual feature type.

Table B-15 Selectors for the linguistic rearrangement feature

Feature selector	Description
kLinguisticRearrangementOnSelector	Allows the automatic rearrangement of certain glyphs as required by language rules. This is the default setting.
kLinguisticRearrangementOffSelector	Prevents the automatic rearrangement of certain glyphs as required by language rules.

Mathematical Extras Feature Type

The mathematical extras feature type (`kMathematicalExtrasType`) represents a collection of features that are used to set figures and mathematics. For example, this feature can change asterisks to multiplication symbols.

Figure B-13 shows an example of drawing mathematical text without and with the mathematical extras features selectors. The top line is drawn without mathematical extras features. The bottom line is drawn with two mathematical extras features enabled: hyphen to minus sign and asterisk to multiplication sign.

Figure B-13 Drawing text without and with mathematical extras

$$(a + b)/(c - d) = q * s$$

$$(a + b)/(c - d) = q \times s$$

Table B-16 shows the selectors you can use for the mathematical extras feature type. This is a noncontextual, nonexclusive feature type.

Table B-16 Selectors for the mathematical extras feature

Feature selector	Description
kHyphenToMinusOnSelector	Allows the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen-numeral) with a minus sign glyph (–). The default setting is to allow hyphen to minus replacement. See Figure B-13 (page 147).
kHyphenToMinusOffSelector	Prevents the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen-numeral) with a minus sign glyph (–).

Feature selector	Description
<code>kAsteriskToMultiply-OnSelector</code>	Allows the automatic replacement of the sequence space-asterisk-space (or the asterisk in the sequence numeral-asterisk-numeral) with a multiplication sign glyph (X). See Figure B-13 (page 147)
<code>kAsteriskToMultiply-OffSelector</code>	Prevents the automatic replacement of the sequence space-asterisk-space (or the asterisk in the sequence numeral-asterisk-numeral) with a multiplication sign glyph (X).
<code>kSlashedToDivide-OnSelector</code>	Allows the automatic replacement of the sequence space-slash-space (or the slash in the sequence numeral-slash-numeral) with a division sign glyph (÷).
<code>kSlashedToDivide-OffSelector</code>	Prevents the automatic replacement of the sequence space-slash-space (or the slash in the sequence numeral-slash-numeral) with a division sign glyph (÷).
<code>kInequalityLigatures-OnSelector</code>	Allows the automatic replacement of sequences such as “>=” and “<=” with the equivalent ligatures “>=” and “<=”.
<code>kInequalityLigatures-OffSelector</code>	Prevents the automatic replacement of sequences such as “>=” and “<=” with the equivalent ligatures “>=” and “<=”.
<code>kExponentsOnSelector</code>	Allows the automatic replacement of the sequence exponentiation glyph—numeral with the superior forms of the numeral.
<code>kExponentsOffSelector</code>	Prevents the automatic replacement of the sequence exponentiation glyph—numeral with the superior forms of the numeral.

Note: By convention, specifying the `kHyphenToMinusOnSelector` in the mathematical extras feature type overrides specifying the `kHyphenToEnDashOnSelector` in the typographic extras feature type.

Number Case Feature Type

The number case feature type (`kNumberCaseType`) specifies whether to use numerals that extend below the baseline. [Figure B-14](#) shows both kinds of numerals. Lowercase numerals (also called traditional or old-style) may descend below the baseline, as shown in the bottom line of the figure. Uppercase numerals (also called lining) do not descend below the baseline.

Figure B-14 Uppercase and lowercase numerals

0123456789
 0123456789

For fonts that support the number case feature type, you can select either kind of numeral. Table B-17 lists the selectors for this noncontextual feature.

Table B-17 Selectors for the number case feature

Feature selector	Description
<code>kLowerCaseNumbersSelector</code>	Specifies the use of lowercase numerals.
<code>kUpperCaseNumbersSelector</code>	Specifies the use of uppercase numerals.

Number Spacing Feature Type

The number spacing feature type (`kNumberSpacingType`) specifies whether to use fixed-width or proportional-width glyphs for numerals. Figure B-15 shows the difference between fixed-width and proportional-width numerals. In proportional-width numerals, the 1 is narrower than the 0; whereas in fixed-width numerals all numerals have identical widths. Fixed-width numerals are also called collimating because they align well in text that consists of columns of numerical data.

Figure B-15 Fixed-width and proportional-width numerals

0123456789
0123456789

Table B-18 lists the selectors for the number spacing feature type. It is a noncontextual feature.

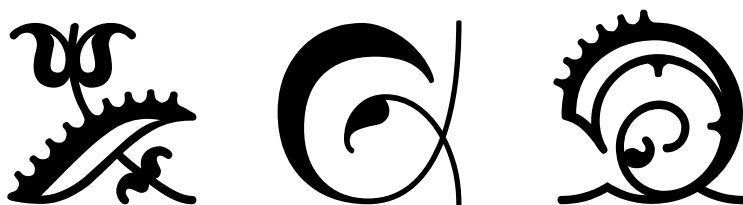
Table B-18 Selectors for the number spacing feature

Feature selector	Description
<code>kMonospacedNumbersSelector</code>	Specifies the use of fixed-width numerals, useful for displaying in columns.
<code>kProportionalNumbersSelector</code>	Specifies the use of proportional-width numerals.
<code>kThirdWidthNumbersSelector</code>	Specifies the use of fixed-width numerals, using an alternate spacing.
<code>non-alphanumeric</code>	Specifies the use of fixed-width numerals, using an alternate spacing.

Ornament Sets Feature Type

The ornament sets feature type (`kOrnamentSetsType`) specifies non-alphanumeric glyph sets, such as decorative borders or musical symbols. Figure B-16 shows an example of glyphs from a fleurons ornamental set.

Figure B-16 Ornamental glyphs



If a font supports the ornament set feature type, you may be able to select among glyph sets, using the selectors shown in Table B-19.

Table B-19 Selectors for the ornament sets feature

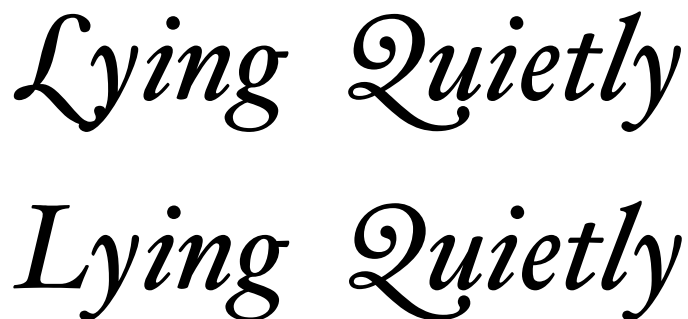
Feature selector	Description
<code>kNoOrnamentSelector</code>	Specifies not to use ornamental glyphs sets. This is the default setting.
<code>kDingbatsSelector</code>	Specifies the use of dingbats: arrows, stars, bullets, or other miscellaneous symbols used for occasional emphasis in display.
<code>kPiCharactersSelector</code>	Specifies a set of related symbols designed for a particular purpose (for example, cartography or musical notation) that do not make up a formal alphabet.
<code>kFleuronsSelector</code>	Specifies the use of ornaments in the shape of flowers, vines leaves, and so forth.
<code>kDecorativeBordersSelector</code>	Specifies the use of decorative borders: glyphs used in interlocking patterns to form text borders.
<code>kInternational-SymbolsSelector</code>	Specifies the use of international symbols, such as the barred circle representing “no.”
<code>kMathSymbolsSelector</code>	Specifies the use of special symbols used to set mathematical or logical expressions.

Overlapping Glyphs Feature Type

The overlapping glyphs feature type (`kOverlappingCharactersType`) controls whether long tails on glyphs are permitted to collide with other glyphs. 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”, as shown in Figure B-17.

Figure B-17 shows text that allows and prevents glyph overlap. The first line does not prevent glyph overlap whereas the second line does. Preventing glyph overlap means that the script “Q” can remain because the following “u” has no descender to collide with it, whereas the script “L” is replaced with a simpler form to avoid collision with the “y”.

Figure B-17 Allowing and preventing glyph overlap



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 overlap, ATSUI substitutes a non-overlapping form of the glyph. Table B-20 lists the selectors that turn the overlapping glyphs feature type on and off. It is a contextual feature.

Table B-20 Selectors for the overlapping characters feature

Feature selector	Description
<code>kPreventOverlapOnSelector</code>	Prevents the collision of an extended part of one glyph with an adjacent glyph.
<code>kPreventOverlapOffSelector</code>	Allows the collision of an extended part of one glyph with an adjacent glyph.

Ruby Kana Feature Type

Ruby text is a run of text associated with another run of text (the base text), usually used to annotate the base text. Ruby text is often used to provide annotations or indicate pronunciation for Asian languages. Figure B-18 shows ruby text—characters drawn in a smaller point size placed above the base text.

Figure B-18 Ruby text above the base text

にほん ことば
 日本にほんの言葉ことば

You can select or deselect ruby variant kana glyphs for the Hiragino font by using the ruby kana feature type (`kRubyKanaType`) with the selectors such as shown in Table B-21.

Table B-21 Selectors for the ruby kana feature type

Feature selector	Description
<code>kRubyKanaOffSelector</code>	Do not use ruby variant glyphs. This is the default setting.
<code>kRubyKanaOnSelector</code>	Use ruby variant glyphs.
<code>kNoRubyKanaSelector</code>	Do not use ruby variant glyphs. This is deprecated; instead use <code>kRubyKanaOffSelector</code> .
<code>kRubyKanaSelector</code>	Use ruby variant glyphs. This is deprecated; instead use <code>kRubyKanaOnSelector</code> .

Smart Swashes Feature Type

The smart swashes feature type (`kSmartSwashType`) controls contextual swash substitution. (A **swash** is a variation, often ornamental, of an existing glyph.) The feature determines 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.

Figure B-19 shows the same phrase written four times, each using a different feature selector for the smart swash feature. The first line is drawn without swash variants. The second line is drawn with only initial swashes enabled. The third line is drawn with only final swashes enabled. The last line is drawn with both initial and final swashes enabled.

Figure B-19 Text drawn with different swash feature selectors

whale voyage

whale voyage

whale voyage

whale voyage

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 B-22. This feature is contextual and nonexclusive.

Table B-22 Selectors for the smart swash feature

Feature selector	Description
kWordInitialSwashesOnSelector	Allows the substitution of swash variants that appear at the start of a word (or a line). This is the default setting.
kWordInitialSwashesOffSelector	Prevents the substitution of swash variants that appear at the start of a word (or a line).
kWordFinalSwashesOnSelector	Allows the substitution of swash variants that appear at the end of a word (or a line).
kWordFinalSwashesOffSelector	Prevents the substitution of swash variants that appear at the end of a word (or a line).
kLineInitialSwashesOnSelector	Allows the substitution of swash variants that appear only at the start of a line.
kLineInitialSwashesOffSelector	Prevents the substitution of swash variants that appear only at the start of a line.
kLineFinalSwashesOnSelector	Allows the substitution of swash variants that appear only at the end of a line.
kLineFinalSwashesOffSelector	Prevents the substitution of swash variants that appear only at the end of a line.
kNonFinalSwashesOnSelector	Allows the substitution of swash variants that are used at the beginning or middle of words. An example of this is the archaic long “s”.

Feature selector	Description
<code>kNonFinalSwashesOffSelector</code>	Prevents the substitution of swash variants that are used at the beginning or middle of words. An example of this is the archaic long “s”.

Style Options Feature Type

The style options feature type (`kStyleOptionsType`) allows the font designer to group together collections of noncontextual substitutions into named sets. These substitutions give the text a specific style or appearance. You can select among sets, using such selectors as those listed in Table B-23.

Table B-23 Selectors for the style options feature

Feature selector	Description
<code>kNoStyleOptionsSelector</code>	Specifies to use plain text. This is the default setting.
<code>kDisplayTextSelector</code>	Specifies the use of a glyph set that is designed for best display at large sizes (typically over 24 point).
<code>kEngravedTextSelector</code>	Specifies the use of a glyph set that has contrasting strokes parallel to the main stroke, giving an engraved-in-stone effect.
<code>kIlluminatedCapsSelector</code>	Specifies the use of a glyph set with complex decoration surrounding the glyphs of capital letters, in the manner used by medieval scribes.
<code>kTiltingCapsSelector</code>	Specifies the use of a glyph set in which capital letters have a special form for display in titles.
<code>kTallCapsSelector</code>	Specifies the use of a glyph set in which capital letters have a taller form than is typical.

Text Spacing Feature Type

The text spacing feature type (`kTextSpacingType`) specifies whether to use proportional, monospaced, or half-width forms of glyphs for characters in a font. Use of this feature is optional; for more precise control see “[Kana Spacing Feature Type](#)” (page 143) and “[Ideographic Spacing Feature Type](#)” (page 142).

Figure B-20 shows text drawn using the text spacing feature type. The first line uses the default text spacing for the font, which is proportional. The next line uses the monospaced text selector. The last line is drawn using the proportional text spacing selector, which in this case, is the same as the default.

Figure B-20 Normal, monospaced, and proportional text spacing

text spacing
t e x t s p a c i n g
text spacing

Table B-24 lists the selectors for the text spacing feature type. It is a noncontextual feature.

Table B-24 Selectors for the text spacing feature

Feature selector	Description
<code>kProportionalTextSelector</code>	Specifies the use of proportional forms of characters. This is the default setting.
<code>kMonospacedTextSelector</code>	Specifies the use of monospaced forms of characters.
<code>kHalfWidthTextSelector</code>	Specifies the use of half-width forms of characters.

Transliteration Feature Type

The transliteration feature type (`kTransliterationType`) specifies that text in one format is to be displayed using another format. For example, displaying a Hanja string as Hangul, as shown in Figure B-21. The top line is Hanja and the bottom line is the Hangul transliteration.

Figure B-21 Hanja text (top) displayed as Hangul (bottom)

天地가 있는 然後에 萬物이 生하다.
천지가 있는 연후에 만물이 생하다.

Table B-25 lists the selectors for the transliteration feature type. It is a noncontextual, exclusive feature.

Table B-25 Selectors for the transliteration feature

Feature selector	Description
<code>kNoTransliterationSelector</code>	Specifies not to use transliteration. This is the default setting.
<code>kHanjaToHangulSelector</code>	Specifies to display Hanja as Hangul.
<code>kHiraganaToKatakanaSelector</code>	Specifies to display Hiragana as Katakana.

Feature selector	Description
<code>kKatakanaToHiraganaSelector</code>	Specifies to display Katakana as Hiragana.
<code>kKanaToRomanizationSelector</code>	Specifies to display Kana as Roman.
<code>kRomanizationToHiraganaSelector</code>	Specifies to display Roman as Hiragana.
<code>kRomanizationToKatakanaSelector</code>	Specifies to display Roman as Katakana.
<code>kHanjaToHangulAltOneSelector</code>	Specifies to display Hanja as Hangul, using alternative glyph set 1.
<code>kHanjaToHangulAltTwoSelector</code>	Specifies to display Hanja as Hangul, using alternative glyph set 2.
<code>kHanjaToHangulAltThreeSelector</code>	Specifies to display Hanja as Hangul, using alternative glyph set 3.

Typographical Extras Feature Type

The typographical extras features type (`kTypographicExtrasType`) represents a collection of effects that are associated with sophisticated typography, such as substitution of en dashes for hyphens. Fonts that support the typographic extras feature type allow you to specify certain typographic conventions, using such selectors as those shown in Table B-26. This feature is noncontextual and nonexclusive.

Table B-26 Selectors for the typographical extras feature

Feature selector	Description
<code>kHyphensToEmDashOnSelector</code>	Allows the automatic replacement of two adjacent hyphens with an em dash. This is the default setting.
<code>kHyphensToEmDashOffSelector</code>	Prevents the automatic replacement of two adjacent hyphens with an em dash.
<code>kHyphenToEnDashOnSelector</code>	Allows the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen-numeral) with an en dash.
<code>kHyphenToEnDashOffSelector</code>	Prevents the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen-numeral) with an en dash.
<code>kSlashedZeroOnSelector</code>	Allows the forced use of the zero glyph with a slash, regardless of whether the font specifies a slashed zero as the default.
<code>kSlashedZeroOffSelector</code>	Prevents the forced use of the zero glyph with a slash, regardless of whether the font specifies a slashed zero as the default.

Feature selector	Description
kFormInterrogBangOnSelector	Allows the automatic replacement of the sequence “?!” or “!?” with the font’s interrobang glyph.
kFormInterrogBangOffSelector	Prevents the automatic replacement of the sequence “?!” or “!?” with the font’s interrobang glyph.
kSmartQuotesOnSelector	Allows the automatic contextual replacement of straight quotation marks with curly ones.
kSmartQuotesOffSelector	Prevents the automatic contextual replacement of straight quotation marks with curly ones.
kPeriodsToEllipsisOnSelector	Allows the automatic replacement of three adjacent periods with an ellipsis.
kPeriodsToEllipsisOffSelector	Prevents the automatic replacement of three adjacent periods with an ellipsis.

Unicode Decomposition Feature Type

The Unicode decomposition feature type (`kUnicodeDecompositionType`) controls whether combining marks are joined with base characters to form composite glyphs. Table B-27 shows the selectors you can use for this feature.

Table B-27 Selectors for the Unicode decomposition feature type

Feature selector	Description
kCanonicalCompositionOnSelector	Specifies that Unicode canonical composing sequences should be recognized and combined into glyphs.
kCanonicalCompositionOffSelector	Specifies that Unicode canonical composing sequences should not be recognized and combined into glyphs.
kCompatibilityCompositionOnSelector	Specifies that Unicode compatibility composing sequences should be recognized and combined into glyphs.
kCompatibilityCompositionOffSelector	Specifies that Unicode compatibility composing sequences should not be recognized and combined into glyphs.
kTranscodingCompositionOnSelector	Specifies that Apple transcoding hints (for legacy Mac OS character sets) should be recognized and combined into glyphs.
kTranscodingCompositionOffSelector	Specifies that Apple transcoding hints (for legacy Mac OS character sets) should not be recognized and combined into glyphs.

Vertical Position Feature Type

The vertical position feature type (`kVerticalPositionType`) controls the selection of superscript and subscript glyph sets. Figure B-22 shows text drawn using normal, superior, inferior, and ordinal vertical positions.

Figure B-22 Normal, superior, inferior, and ordinal vertical positions

normal *superiors* *inferiors* *ordinals*

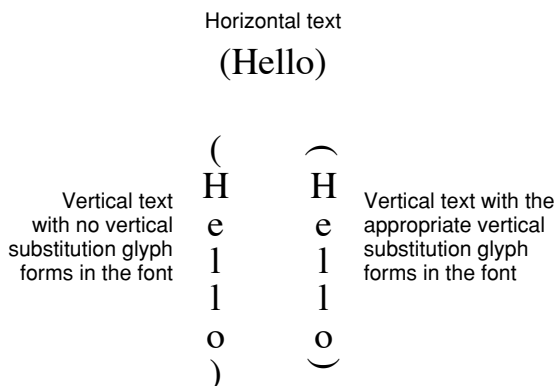
The vertical position feature type is a contextual, exclusive feature type. The selectors for this feature are described in Table B-28.

Table B-28 Selectors for the vertical position feature

Feature selector	Description
<code>kNormalPositionSelector</code>	Specifies to display the text with no vertical displacement. This is the default setting. See the first word in Figure B-22.
<code>kSuperiorsSelector</code>	Specifies use of superiors; changes any characters having superior forms in the font into those forms, used typically for superscripts. See the second word in Figure B-22.
<code>kInferiorsSelector</code>	Specifies use of inferiors; changes any characters having inferior forms in the font into those forms, used typically for subscripts. See the third word in Figure B-22.
<code>kOrdinalsSelector</code>	Specifies to contextually change certain letters into their superior forms. As shown in the fourth word in Figure B-22, the ordinals selector may not effect drawing.

Vertical Substitution Feature Type

The vertical substitution feature type (`kVerticalSubstitutionType`) specifies that certain glyphs (such as parentheses) should change their appearance in vertical runs of text. Figure B-23 illustrates how vertical substitution works. The top line shows horizontal text—a word enclosed in parentheses. When the text is displayed vertically without turning on vertical substitution (left side of the figure), the parentheses appear odd. When vertical substitution is turned on (right side of the figure) the parentheses appear in the appropriate orientation.

Figure B-23 Vertical substitution forms in a font

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. For vertical substitution to work, 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 even when you turn on this feature explicitly. Note that vertical substitution is not used to rotate glyphs, simply to substitute forms, as shown in Figure B-23.

Table B-29 lists the selectors for the vertical substitution feature type. It is a contextual feature.

Table B-29 Selectors for the vertical substitution feature

Feature selector	Description
kSubstituteVerticalFormsOnSelector	Allows the substitution of alternate glyph forms in vertical lines.
kSubstituteVerticalFormsOffSelector	Prevents the substitution of alternate glyph forms in vertical lines.

Document Revision History

This table describes the changes to *ATSUI Programming Guide*.

Date	Notes
2007-07-10	Fixed typographical error, changed title.
	Added an ampersand to <code>myDescent</code> in the second call to the function <code>ATSUGetLineControl</code> in Listing 3-5 (page 86).
	Changed the title from <i>Rendering Unicode Text With ATSUI</i> to make the title consistent with other documents of this kind.
2007-03-06	Fixed typographical errors.
	Corrected code in step 2 of “Creating Style Objects and Setting Attributes” (page 70).
	Made a correction to the code in “Calculating Line Height” (page 86).
	Made a correction to the code in “Breaking Lines” (page 83).
2006-09-05	Corrected typographical errors.
	Changed <code>err</code> to <code>status</code> in Listing 6-2 (page 119).
2006-01-10	Updated the selectors for the CJK italic Roman and ruby kana feature types.
2005-11-09	Fixed hyperlinks.
2005-07-07	Fixed typographical errors.
2002-12-02	Revised the index for PDF and print versions of this document.
	Changed formatting for style and layout attributes.
2002-10-11	Added additional information on the constants used to specify font substitution.
	Fixed typographical and formatting errors.
2002-09-16	First release of this document. Includes conceptual and task information through ATSUI version 2.4.
	Supersedes these technical notes:
	TN 2033, <i>How to use the ATSUI Low Level APIs to get glyph outlines</i>
	TN 1027, <i>Improving text drawing performance when using ATSUI</i>

REVISION HISTORY

Document Revision History

ATSUI Glossary

absolute position A specific position, given in coordinates, for the origin of each character or glyph in a line of text. Compare [relative position](#).

active end When selecting text, the point at which the user releases the mouse button. See also [anchor point](#).

advance delta The distance between the end of one glyph's advance and the next glyph's real position.

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. Also referred to as flushness.

anchor point The position in the text at which the user positions the pointer and presses the mouse button. See also [active end](#).

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.

anti-aliasing The smoothing of jagged edges on a displayed glyph by modifying the transparencies of individual pixels along the glyph's edge.

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.

backing store A file in which the Virtual Memory Manager stores the contents of unneeded pages of memory.

baseline An imaginary line used to align glyphs in a line of text.

baseline delta The distance (in points) between a baseline and $y = 0$; sometimes referred to as delta-y. See also [baseline type](#).

baseline type The classification of baseline used with a particular kind of text. See, for example, [Roman baseline](#).

Bézier curve A cubic equation originally developed by Pierre Bézier. In typography, used to define the shape of a glyph.

bidirectional See [bidirectional script system](#).

bidirectional script system A script system in which 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.

blit Slang for copying an image from memory to the screen.

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.

byte offset The numbering of character codes in source text. Compare [edge offset](#).

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 [dual caret](#).

caret angle The angle of a caret or of 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 a caret offset in the source text.

caret type A designation of the behavior of the caret at direction boundaries in text. See also [dual 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 clusters 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 A numerical representation of a character. Each writing system or language has one or more character encodings—tables that relate character codes to the characters they represent.

character encoding An internal conversion table for interpreting a specific character set.

contextual features Features that are applied to a glyph depending on the glyph's position relative to adjacent glyphs. Compare [noncontextual features](#).

contextual form An alternate form of a glyph whose use depends on the glyph's placement in a word.

contiguous highlighting Highlighting that consists of a single, contiguous shape across direction boundaries, even when it does not exactly match the selection range to which it corresponds. Compare [discontinuous highlighting](#).

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.

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](#).

cubic curve A curve defined by a cubic equation. See also [Bézier curve](#).

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.

device delta A value used to adjust truncated fractional values for cases in which fractional positioning can't be used; for example, to compensate for integer drawing in QuickDraw. Device delta values are usually used when anti-aliasing is turned off. However, these values can be used when anti-aliasing is on, to assure that the glyphs in a connected script (such as one that uses the Zapfino font) are connected smoothly.

device advance The number of pixels of the advance for the glyph as actually drawn on the screen.

device space The coordinate system that defines the position and scale (pixel size) of a specific view device.

diacritical marks A mark, such as an accent, that is used in conjunction with a character to indicate phonetic value.

direct-access functions ATSUI functions that allow you to manipulate glyph data directly.

direction boundary A point, between offsets in memory or glyphs in a display, at which the direction of stored or displayed text changes.

discontinuous highlighting Highlighting that exactly matches the selection range it corresponds to. It may consist of discontinuous 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. Compare [input order](#).

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](#).

drop capital A large uppercase letter that drops below the main line of text for aesthetic reasons.

dual 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.

edge offset A byte offset into the source text associated with a text layout object that specifies a position between byte values. Edge offsets in source text are related to caret positions in display text. Compare [byte offset](#).

exclusive feature type A feature for which you can choose only one of the available feature selectors, such as whether numbers are to be proportional or fixed-width. Compare [nonexclusive feature type](#).

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](#).

flushness See [alignment](#).

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 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 the text associated with a text layout object.

font ID A value that identifies a font to the font management system. The font ID is assigned to a font at system startup; the specific value does not persist across system startups.

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 variation An algorithmic way to produce a range of tpestyles along a particular variation axis.

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, which in turn specify individual glyphs.

glyph direction The direction in which successive glyphs are read.

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 0; each succeeding glyph to the right has an index one greater than the previous glyph. Compare [edge offset](#).

glyph orientation A value that specifies which direction (vertical or horizontal) glyphs should be drawn.

glyph origin The point used to position a glyph when drawing.

glyph outline The curves that make up the shape of the glyph.

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.

Hangul A Korean subscript that consists of blocks of component glyphs called Jamo that are characters different from typical character clusters in that they are treated as singular units in memory; there are no principal characters and attachments.

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.

hints Information provided with a font that can be used to scale glyphs to various sizes.

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.

hyphenation point An entry in an array of edge offsets in the source text at which it is appropriate to break a line of display text.

ideal metrics Resolution-independent measurements used to describe how a glyph is drawn. Compare [screen metrics](#).

image bounding rectangle The smallest rectangle that completely encloses the filled or framed parts of a block of text. See also [typographic bounding rectangle](#).

imposed width A 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 position](#).

Jamo An individual phonetic glyph in the Korean script that is transformed and combined into clusters called Hangul.

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 override The degree to which ATSUI should override justification behavior for glyphs in a style run.

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 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.

Last Resort font A collection of glyphs that represent types of Unicode characters. These glyphs can be used as a backup to any other font; if the font cannot represent any particular Unicode character, the appropriate “missing” glyph from the Last Resort font can be used instead.

layout cache A cache that contains all the information ATSUI needs to draw a range of text associated with a text layout object. This includes caret positions, the memory locations of glyphs, and other information needed to lay out the glyphs.

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 breaking up of a ligature into its component glyphs during justification so that the individual glyphs may more evenly occupy the space allotted to the ligature.

ligature splitting The division of a ligature for hit-testing purposes into regions corresponding to each of 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 and layout attributes Attributes that specify how the lines of text associated with the text layout object are displayed and formatted. Line attributes control an individual line of text; layout attributes control all of the text associated with a text layout object.

line direction The overall direction in which a line of text is read. The 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.

margins The left, right, top, and bottom sides of the text area.

metamorphosis The process by which glyphs are rearranged, substituted, deleted, and inserted based upon their properties and contextual states.

native curve type The curve type—cubic or quadratic—used by a font designer to specify a font.

negative justification A layout in which the glyphs on a line do not naturally fit within the line width set by the developer.

noncontextual features Features that are applied in the same manner to a glyph regardless of the adjacent glyphs. See also [contextual features](#).

nonexclusive feature type A feature for which you can enable any number of feature selectors at once. Compare [exclusive feature type](#).

offsets Monotonically increasing or decreasing values. In ATSUI, offsets are in `Unichars` units and are typically used to specify starting and ending points for a string of text.

optical alignment The fine adjustment of glyph positions at the ends of lines to give a more even visual appearance to margins.

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.

quadratic curve A curve specified by a quadratic equation.

real position The actual drawing position on the x-axis for the origin of each character or glyph in a line of text given in coordinates relative to the preceding character or glyph.

relative position A position for the origin of each character or glyph in a line of text given in coordinates relative to the preceding character or glyph. Compare [absolute position](#).

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.

ruby text Text usually used to provide annotations or indicate pronunciation for Asian languages. Ruby text is displayed using a smaller font size than the text it annotates.

run A sequence of glyphs that are contiguous in memory and share a set of common attributes.

screen metrics Resolution-dependent measurements used to describe how a glyph is drawn. Compare [ideal metrics](#).

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 uppercase letter “M” in Times Roman typeface.

smart swash A variation of an existing glyph (often ornamental) that is contextual. Compare [swash](#).

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 text](#).

split caret See [dual caret](#).

storage order See [input order](#), [display order](#).

style run A sequence of glyphs (contiguous in memory backing store) that share the same style.

surrogates Values that allow additional characters to be mapped to the Unicode 16-bit character set.

swash A variation of an existing glyph (often ornamental) that is noncontextual. Compare [smart swash](#).

style attributes A collection of values and settings that override the font-specified behavior for displaying and formatting text in a style run.

style object An opaque object that contains a collection of stylistic attributes. Style objects can be applied to runs within a text layout object.

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 face An algorithmic way for your application to produce typesyles.

text layout object An opaque object that contains information to control the display and formatting of the text to which the object is associated.

text run A complete unit of text made up of character codes or glyph codes.

text styles The visual attributes, other than size, applied as a systematic variation to the plain (unstyled) characteristics of a font’s glyphs. Some typical text styles include plain, bold, italic, underline, outline, shadow, condensed, and extended.

text width The area between the margins; it is the length available for displaying a line of text.

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 a line of text, not just the kerning pairs already defined by the font. You can increase or decrease interglyph spacing by adjusting the tracking setting. See [tracking setting](#); compare [kerning](#).

tracking 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.

triple Three values that consist of an attribute tag, a value for that tag, and the size of the value. In ATSUI, triples are used to specify style, line, and layout attributes.

typestyle See [text styles](#).

typographic bounding rectangle The smallest rectangle that encloses the full span of the glyphs from the ascent line to the descent line. See also [image bounding rectangle](#).

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 the Mac OS.

Unicode A character encoding system designed to support the interchange, processing, and display of all the written texts of the diverse languages of the modern world.

Unicode decomposition Splitting a composite glyph into its component parts, such as a base character and a combining mark.

variation axis A range included in a font by the font designer that allows a font to produce different typestyles.

with-stream shift A uniform shift parallel to the baseline of the positions of individual pairs or sets of glyphs in the style run. Compare [cross-stream shift](#).

