
Core Text Programming Guide

User Experience: Text Layout



2008-06-09



Apple Inc.
© 2008 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

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, Carbon, Cocoa, Mac, Mac OS, Macintosh, Objective-C, Quartz, and QuickDraw are trademarks of Apple Inc., registered in the United States and other countries.

Helvetica 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 7**

Organization of This Document 7
See Also 7

Chapter 1 **Core Text Overview 9**

Mac OS X Text Technologies 9
Design Goals and Principles 10
Core Text Features and Capabilities 10
 System Data Types and Services 11
 Core Text Input 11
 Characters and Glyphs 11
Core Text Objects 12
 Layout Objects 12
 Font Objects 15

Chapter 2 **Common Operations 17**

Simple Paragraphs 17
Simple Text Labels 18
Columnar Layout 19
Manual Line Breaking 20
Font Creation and Storage 21
Accessing Font Metrics 24
Creating Related Fonts 25

Document Revision History 27

Figures and Listings

Chapter 1 **Core Text Overview** 9

- Figure 1-1 Glyphs of the character A 12
- Figure 1-2 Ligatures 12
- Figure 1-3 Text layout data flow 13
- Figure 1-4 A frame object containing lines and glyph runs 14
- Figure 1-5 Creating a font from a font descriptor 15

Chapter 2 **Common Operations** 17

- Listing 2-1 Typesetting a simple paragraph 17
- Listing 2-2 Typesetting a simple text label 18
- Listing 2-3 Performing columnar text layout 19
- Listing 2-4 Performing manual line breaking 20
- Listing 2-5 Creating a font descriptor from a name and point size 21
- Listing 2-6 Creating a font descriptor from a family and traits 21
- Listing 2-7 Creating a font from a font descriptor 22
- Listing 2-8 Serializing a font 22
- Listing 2-9 Creating a font from serialized data 23
- Listing 2-10 Calculating line height 24
- Listing 2-11 Getting glyphs for characters 24
- Listing 2-12 Changing traits of a font 25
- Listing 2-13 Converting a font to another family 26

Introduction

Core Text is an advanced, low-level technology for laying out text and handling fonts. It is designed for high performance and ease of use. The Core Text API, introduced in Mac OS X v10.5, is accessible from all Mac OS X application environments.

The Core Text layout engine is designed specifically to make simple text layout operations easy to do and to avoid side effects. The Core Text font programming interface is complementary to the Core Text layout engine and is designed to handle Unicode fonts natively, unifying disparate Mac OS X font facilities into a single comprehensive programming interface.

This document is intended for developers who need to do text layout and font handling in Carbon applications and at a low level. If you can develop your application using higher-level constructs, such as `NSTextView`, then you should use the Cocoa text system, introduced in *Text System Overview*. If, on the other hand, you need to render text directly into a Core Graphics context, then you should use Core Text.

More information about the position of Core Text among other Mac OS X text technologies is presented in [“Mac OS X Text Technologies”](#) (page 9).

Organization of This Document

This document is organized into the following chapters:

[“Core Text Overview”](#) (page 9) describes the Core Text system in terms of its design goals and feature set. It also introduces the opaque types that encapsulate the text layout and font handling capabilities of the system.

[“Common Operations”](#) (page 17) presents snippets of code with commentary illustrating typical uses of the main Core Text opaque types.

See Also

In addition to this document, there are several that cover more specific aspects of Core Text or describe the software services used by Core Text.

- *Core Text Reference Collection* provides complete reference information for the Core Text layout and font API.
- *CoreTextTest* is a sample code project that shows how to use Core Text in the context of a complete Carbon application.
- *CoreTextArc* is a sample code project that illustrates the use of fonts, lines, and runs in a Core Text Carbon application.

- *Getting Started with Core Foundation* provides access to documentation for Core Foundation, a framework that provides abstractions for common data types and fundamental software services used by Core Text.

Most of the following text-related documents were written before the advent of Core Text, but they provide valuable information about other text-related technologies in Mac OS X.

- *Getting Started with Text and Fonts* provides pointers to the various text-related technologies in Mac OS X.
- *Text System Overview* gives an introduction to the Cocoa text system.
- *Text Layout Programming Guide for Cocoa* describes the Cocoa text layout engine.
- *ATSUI Programming Guide* describes the Apple Type Services for Unicode Imaging technology.

Core Text Overview

The Core Text framework is an advanced, low-level technology for laying out text and handling fonts. Designed for high performance and ease of use, the Core Text layout engine is up to twice as fast as ATSUI (Apple Type Services for Unicode Imaging). The Core Text layout API is simple, consistent, and tightly integrated with Core Foundation, Core Graphics, and Cocoa.

The Core Text font API is complementary to the Core Text layout engine. Core Text font technology is designed to handle Unicode fonts natively, bridging the gap between Carbon and Cocoa font references, and providing efficient font handling for Core Text layout. Core Text brings the capabilities and coherent design of Cocoa text and fonts to a broader, lower-level client base.

Mac OS X Text Technologies

The Macintosh operating system has provided sophisticated text handling and typesetting capabilities from its beginning. In fact, these features sparked the desktop publishing revolution. Core Text is the most modern text-handling technology on the platform. It is designed specifically for Mac OS X and is written in C, so it can be called from any language in the system. It is positioned as a core technology to provide consistent, high-performance text services to other frameworks throughout the system, and the Core Text API is accessible to applications that need to use it directly. Core Text resides in the Application Services umbrella framework (`ApplicationServices`) so that it is callable from both Carbon and Cocoa and has all of the lower-level services it needs.

Core Text is not meant to replace the Cocoa text system, although it provides the underlying implementation for many Cocoa text technologies. If you can deal with high-level constructs, such as text views, you can probably use Cocoa. For this reason, Cocoa developers typically have no need to use Core Text directly. Carbon developers, on the other hand, will find Core Text faster and easier to use, in many cases, than preexisting Mac OS X text layout and font APIs.

To decide whether Core Text is the right Mac OS X text technology for your application, apply the following guidelines:

- If you can, use Cocoa text. The `NSTextView` class is the most advanced, full-featured, flexible text view in Mac OS X. For small amounts of text, use `NSTextField`.
- To display web content in your application, use Web Kit.
- If you need to use Carbon only, consider using `NSTextView` with `HICocoaView`.
- If you need a lower-level API for drawing any kind of text into a Quartz graphics context (`CGContext`), consider using Core Text directly.
- If you need features that Core Text does not provide (such as text editing and font activation) and you can't use Cocoa, look at other technologies such as MLTE, ATSUI, and ATS.

Generally speaking, Core Text is for applications that need a low-level text-handling technology correlating with the Core Graphics framework (Quartz). If you work directly with Quartz and you need to draw some text, use Core Text. If, for example, you have your own page layout engine—you have some text and you know where it needs to go in your view—you can use Core Text to generate the glyphs and position them relative to each other with all the features of fine typesetting, such as kerning, ligatures, line-breaking, and justification.

Design Goals and Principles

Core Text is designed to provide the following benefits:

- A comprehensive, unified set of text-layout and font APIs
- High performance and ease of use
- Tight integration with Cocoa, Core Foundation, and Core Graphics (Quartz)
- Native Unicode handling
- 64-bit application support
- Clean, simple, consistent API design
- Simple interfaces for simple operations
- A flexible interface to layout and glyph data
- A predictable cost structure and rational division of labor

A primary design goal of Core Text layout is to make simple things easy to do. So, for example, if you want to draw a paragraph of text or a simple text label on the screen, you don't need much code. A corollary principle of the Core Text design is that clients are not required to pay for features they don't use.

The objects defined by Core Text opaque types provide a progression from simplicity to complexity, in terms of their use and interface. That is, higher-level objects do more for you, and so they are easier to use (although they may be more complex internally). For example, the highest-level object in Core Text is the framesetter, which fills a path (defined by a `CGPath` object representing a rectangle) with text. The framesetter object uses other Core Text objects, such as typesetter, line, and glyph run objects, to accomplish its work: creating frame objects, which are lines of glyphs laid out within a shape.

Clients who simply need to lay out a paragraph need only work with the framesetter. Clients who need to intervene in the text layout process at a lower level can deal with lower level objects, such as line objects. Line objects can draw themselves individually or be used to obtain glyph information. With Core Text you use the highest-level object you can to get your job done.

Core Text Features and Capabilities

Core Text performs text layout and font access. The text layout engine generates glyphs from characters and positions the glyphs into glyph runs, lines, and multiline frames. It also provides glyph- and layout-related data, such as glyph positioning and measurement of lines and frames. The API handles character attributes and paragraph styles, including various types of tab styles and positioning.

The Core Text font API brings to Carbon developers the same capabilities enjoyed by Cocoa developers through `NSFont` and `NSFontDescriptor`. The API provides font viewing and selecting. It provides font references, font descriptors (objects that encapsulate font data sufficient to instantiate a font reference), and easy access to font data. It also provides support for multiple master fonts, font variations, font cascading, and font linking. The Core Text font API is designed to be very complete, so that you don't have to go to different layers to do what you need to do.

System Data Types and Services

Core Text uses system data types and services wherever possible, and you use the same conventions that pertain to the other core frameworks in Mac OS X. So, for example, Core Text uses Core Foundation objects for many input and output parameters, enabling them to be retained, released, and stored in Core Foundation collection classes. Other objects handled by Core Text are provided by the Core Graphics framework, for example, `CGPath` objects. Moreover, because many Core Foundation objects are toll-free bridged with Cocoa Foundation objects, you can usually use Foundation objects in place of Core Foundation objects passed into Core Text functions. Use of these standard types and toll-free bridging ensure that you don't have to perform expensive type conversions to get data into and out of Core Text.

Core Text is built to work directly with Core Graphics, also known as Quartz, which is the high-speed graphics rendering engine that handles two-dimensional imaging at the lowest level in Mac OS X. Quartz is the only way to get glyphs drawn at a fundamental level, and, because Core Text provides all data in a form directly usable by Quartz, the result is high-performance text rendering.

Core Text Input

The input type most basic to Core Text is the Core Foundation attributed string, represented by `CFAttributedStringRef` or its Cocoa counterpart, `NSAttributedString`, which are toll-free bridged. The attributes are key-value pairs that define style characteristics of the characters in the string, which are grouped in ranges that share the same attributes. Examples of text attributes are font and color. The attributes themselves are passed into attributed strings, and retrieved from them, using `CFDictionary` objects. (Though `CFDictionaryRef` and `NSDictionary` are also toll-free bridged, the individual attribute objects stored in the dictionary may not be.) The typesetting mechanism in Core Text uses the information in the attributed string to perform character-to-glyph conversion.

Characters and Glyphs

One of the most important capabilities of fine typesetting is character-to-glyph conversion. It is important to distinguish between characters and glyphs in discussing a text layout engine. Characters are essentially numbers representing code points in a character set or encoding scheme, such as Unicode, the character set used for all text in Mac OS X. The Unicode standard provides a unique number for every character in every modern written language in the world, independent of the platform, program, and programming language being used.

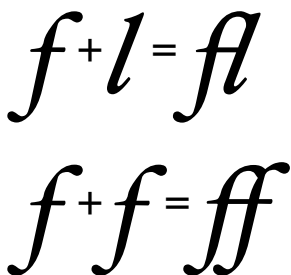
A glyph is a graphic shape used to depict a character. Glyphs are also represented by numeric codes, called glyph codes, that are indexes into a particular font. Glyphs are selected during composition and layout processing by the character-to-glyph conversion process. There are any number of glyphs that correspond to a particular character. For example, the character "uppercase A" has different glyphs for different typefaces (such as Helvetica and Times) and type styles (such as bold and italic). Figure 1-1 shows various glyphs, all of which represent an "uppercase A."

Figure 1-1 Glyphs of the character A



Moreover, the correspondence between characters and glyphs is not one to one, and the context within which a character appears can affect the glyph chosen to represent it. For example, in many fonts an “f” and “l” appearing side-by-side in a character string are replaced by a ligature, which is a single glyph depicting the letters joined together. Figure 1-2 shows two examples of individual characters and the single-glyph ligatures often used when they are adjacent. Character-to-glyph conversion is a complex and difficult task that Core Text performs quickly and efficiently.

Figure 1-2 Ligatures



Core Text Objects

Core Text objects are based on the corresponding opaque types defined by the framework. In the sections that follow, you learn how the primary Core Text objects interact to accomplish various client tasks.

Layout Objects

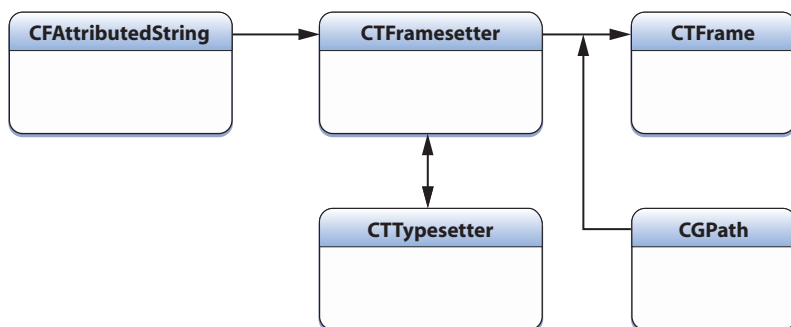
Layout objects make up the Core Text layout engine. This section discusses the primary layout objects: framesetter, frame, typesetter, line, and glyph run objects. In addition this section briefly discusses the other Core Text layout objects: paragraph styles, text tabs, and glyph info objects.

Framesetters and Frames

The framesetter is the highest-level object in the Core Text layout engine, represented by the `CTFramesetter` opaque type. A framesetter generates text frames by filling a path with text. That is, `CTFramesetter` is an object factory for `CTFrame` objects that are ready to draw.

The framesetter takes an attributed string object (`CFAttributedString`) and a shape descriptor object (`CGPath`) and calls into the typesetter to create line objects that fill that shape. The output is a frame object containing an array of lines. This array of lines is a paragraph, a multiline layout. The frame can draw itself directly into a graphic context. You can also retrieve the lines to manipulate before drawing. For example, you might adjust their positioning. Figure 1-3 shows the data flow among objects performing text layout.

Figure 1-3 Text layout data flow



The framesetter applies paragraph styles to the frame text as it is laid out. Paragraph styles are represented in Core Text by objects storing attributes that affect paragraph layout. Among these attributes are alignment, tab stops, writing direction, line-breaking mode, and indentation settings.

It's advantageous to use the framesetter to perform the common operation of typesetting a multiline paragraph because it handles all of the details of producing frames, instantiating other objects, such as the typesetter, as needed. The `CTFramesetter` opaque type provides functions to create a framesetter with an attributed string, to create frame objects, and to return its typesetter. As with all Core Text objects, `CTFramesetter` can also return its Core Foundation type identifier.

Typesetters

A typesetter performs the fundamental text layout operations of character-to-glyph conversion and positioning of those glyphs into lines. That is, it determines which glyphs to use and where to place them relative to each other, producing line objects. Typesetters are represented by the `CTTypesetter` opaque type.

The typesetter also suggests line breaks. It finds how many glyphs can fit within a single line within a given space. It then determines the length of the line by using word breaks, word wrapping, or finer-grained cluster breaks. Simple word wrapping is the default method of creating line breaks.

The framesetter instantiates a typesetter and uses it to create the line objects used to fill a frame. You can also use a typesetter directly, as described in [“Manual Line Breaking”](#) (page 20).

Lines and Glyph Runs

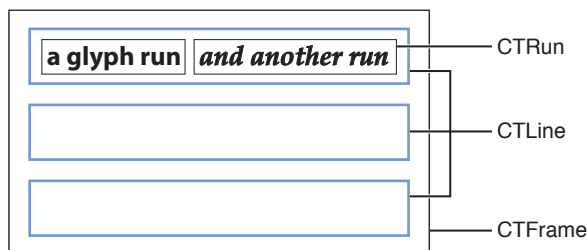
A line object represents a line of text and is represented in Core Text by the `CTLine` opaque type. A `CTLine` object contains an array of glyph runs. Line objects are created by the typesetter during a framesetting operation and, like frames, can draw themselves directly into a graphics context. Line objects hold the glyphs that are the result of the text layout process, created from text and style information.

A line corresponds to a range of characters. It could be miles long or, more often, one of a series of lines contained within a paragraph. The paragraph is represented in Core Text by a `CTFrame` object, which contains the paragraph's line objects. Accordingly, you can retrieve line objects from their frame object.

A line object contains glyph-run objects, represented by the `CTRun` opaque type. A glyph run is a set of consecutive glyphs sharing the same attributes and direction. The typesetter creates glyph runs as it produces lines from character strings, attributes, and font objects. That is, a line is constructed of one or more glyphs

runs. Glyph runs can draw themselves into a graphic context, if desired, although most clients have no need to interact directly with glyph runs. Figure 1-4 shows the conceptual hierarchy of a frame object containing line objects that, in turn, contain glyph-run objects.

Figure 1-4 A frame object containing lines and glyph runs



CTLine has a convenience method for creating a freestanding line independent of a frame, `CTLineCreateWithAttributedString`. You can use this method to create a line object directly from an attributed string without needing to create and manage a typesetter. Without a typesetter, however, there's no way to calculate line breaks, so this method is meant for a single line only (for example, creating a text label).

After you have a line object, you can do a number of things with it. For example, you can have the line create a justified or truncated copy of itself, and you can ask the line for pen offsets for various degrees of flushness. You can use these pen offsets to draw the line with left, right, or centered alignments. You can also ask the line for measurements, such as its image bounds and typographic bounds. Image bounds represent the rectangle tightly enclosing the graphic shapes of the glyphs actually appearing in the line. Typographic bounds include the height of the ascenders in the font and the depth of its descenders, regardless of whether those features appear in the glyphs in a given line.

Like a frame object, a line object is ready to draw. You simply set the text position in a Core Graphics context and have the line draw itself. Core Text uses the same placement strategy as Quartz, setting the origin of the text on the text baseline.

In Quartz, you specify the location of text in user-space coordinates. The text matrix specifies the transform from text space to user space. The text position is stored in the `tx` and `ty` variables of the text matrix. When you first create a graphics context, it initializes the text matrix to the identity matrix; thus text-space coordinates are initially the same as user-space coordinates. Quartz conceptually concatenates the text matrix with the current transformation matrix and other parameters from the graphics state to produce the final text-rendering matrix, that is, the matrix actually used to draw the text on the page.

Other Layout Objects

In addition to the framesetter, frame, typesetter, and line objects, Core Text provides other objects to complete the text layout process: paragraph style, text tab, and glyph info objects.

Paragraph style objects encapsulate paragraph or ruler attributes in an attributed string and are represented by the `CTParagraphStyle` opaque type. A paragraph style object is a complex attribute value in an attributed string, storing a number of subattributes that affect paragraph layout for the characters of the string. Among these subattributes are alignment, tab stops, writing direction, line-breaking mode, and indentation settings. The `CTTextTab` opaque type represents a tab stop in a paragraph style, storing an alignment type and location. The `CTGlyphInfo` opaque type enables you to override a font's specified mapping from Unicode to the glyph ID.

Font Objects

Font objects are those Core Text objects dealing directly with fonts: the font reference itself, font descriptor objects, and font collection objects.

Fonts

Fonts provide assistance in laying out glyphs relative to one another and are used to establish the current font when drawing in a graphics context. The Core Text font opaque type (`CTFont`) is a specific font instance that encapsulates a lot of information. Its reference type, `CTFontRef`, is toll-free bridged with `NSFont`. When you create a `CTFont` object, you typically specify (or use a default) point size and transformation matrix, which gives the font instance specific characteristics. You can then query the font object for many kinds of information about the font at that particular point size, such as character-to-glyph mapping, encodings, font metric data, and glyph data, among other things. Font metrics are parameters such as ascent, descent, leading, cap height, x-height, and so on. Glyph data includes parameters such as bounding rectangles and glyph advances.

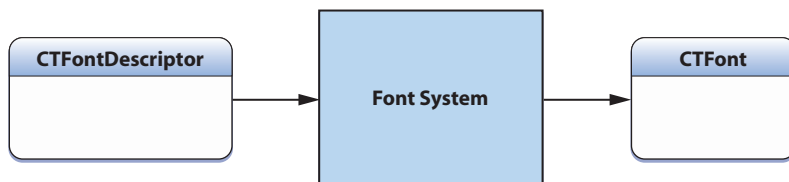
There are many ways to create font references. The preferred method is from a font descriptor using `CTFontCreateWithFontDescriptor`. You can also use a number of conversion APIs, depending on what you have to start with. For example, you can use the PostScript name of the typeface (`CTFontCreateWithName`), an ATS font reference (`CTFontCreateWithPlatformFont`), a Core Graphics font reference (`CTFontCreateWithGraphicsFont`), or a QuickDraw font reference (`CTFontCreateWithQuickdrawInstance`). There's also `CTFontCreateUIFontForLanguage`, which creates a reference for the user-interface font for the application you're using in the localization you're using.

Core Text font references provide a sophisticated, automatic font-substitution mechanism called font cascading. This mechanism takes font traits into account, so it does a better job than previous schemes of picking an appropriate font to substitute for a missing font. Font cascading is based on cascade lists, which are arrays of ordered font descriptors. There is a system default cascade list (which is polymorphic, based on the user's language setting and current font) and a font cascade list that is specified at font creation time. Using the information in the font descriptors, the cascading mechanism can match fonts according to style as well as matching characters. The `CTFontCreateForString` function uses cascade lists to pick an appropriate font to encode a given string. You specify and retrieve font cascade lists using the `kCTFontCascadeListAttribute` property.

Font Descriptors

Font descriptors, represented by the `CTFontDescriptor` opaque type, provide a mechanism to describe a font completely with a dictionary of attributes. `CTFontDescriptorRef` is toll-free bridged to `NSFontDescriptor`. The attributes are properties such as PostScript name, family, and style, and traits such as bold, italic, and monospace. The font descriptor can then be used to create or modify a `CTFont` object. Font descriptors can be serialized and stored in a document to provide persistence for fonts. Figure 1-5 illustrates the font system using a font descriptor to create a specific font instance.

Figure 1-5 Creating a font from a font descriptor



A font descriptor can also be considered as a query into the font system. You can create a font descriptor with an incomplete specification, that is, with one or just a few values in the attribute dictionary, and the system will choose the most appropriate font from those available. The system can also give you a complete list of font descriptors matching your query via `CTFontDescriptorCreateMatchingFontDescriptors`.

Font Collections

Font collections are unions of font descriptors, that is, groups of font descriptors taken as a single object. A font collection is represented by the `CTFontCollection` opaque type. Font collections provide the capabilities of font enumeration, access to global and custom font collections, and access to the font descriptors comprising the collection. You can, for example, create a font collection of all the fonts available in the system by calling `CTFontCollectionCreateFromAvailableFonts`, and you can use the collection to obtain an array of all of the member font descriptors. There is also a function that takes a callback parameter used to sort the returned array of font descriptors.

Common Operations

This chapter describes some common text-layout and font-handling operations and shows, through portions of sample code, how they can be accomplished using Core Text. In addition to the code fragments in this chapter, see the following sample code applications that use Core Text:

- *CoreTextTest* shows how to use Core Text to typeset blocks of text with varying attributes in the context of a complete Carbon application.
- *CoreTextArc* illustrates the use of Core Text fonts, lines, and runs in a Carbon application that sets type along an arched path.

Simple Paragraphs

One of the most common operations in typesetting is laying out a multiline paragraph within an arbitrarily sized rectangular area. Core Text makes this operation easy, requiring only a few lines of Core Text–specific code. To lay out the paragraph, you need a graphics context to draw into, a rectangular path to provide the area where the text is laid out, and an attributed string. Most of the code in this example is required to create and initialize the context, path, and string. After that is done, Core Text requires only three lines of code to do the layout.

Listing 2-1 uses Cocoa to simplify initialization of the graphics context. To see how that operation is done in Carbon, see the *CoreTextTest* sample code or “Graphics Contexts” in the *Quartz 2D Programming Guide*.

Listing 2-1 Typesetting a simple paragraph

```
// Initialize a graphics context and set the text matrix to a known value.
CGContextRef context = (CGContextRef)[[NSGraphicsContext currentContext]
                                     graphicsPort];
CGContextSetTextMatrix(context, CGAffineTransformIdentity);

// Initialize a rectangular path.
CGMutablePathRef path = CGPathCreateMutable();
CGRect bounds = CGRectMake(10.0, 10.0, 200.0, 200.0);
CGPathAddRect(path, NULL, bounds);

// Initialize an attributed string.
CFStringRef string = CFSTR("We hold this truth to be self-evident, that
                           everyone is created equal.");
CFMutableAttributedStringRef attrString =
    CFAttributedStringCreateMutable(kCFAllocatorDefault, 0);
CFAttributedStringReplaceString(attrString,
                                CFRangeMake(0, 0), string);

// Create a color and add it as an attribute to the string.
CGColorSpaceRef rgbColorSpace = CGColorSpaceCreateDeviceRGB();
CGFloat components[] = { 1.0, 0.0, 0.0, 0.8 };
```

```

CGColorRef red = CGColorCreate(rgbcColorSpace, components);
CGColorSpaceRelease(rgbcColorSpace);
CFAttributedStringSetAttribute(attrString, CFRangeMake(0, 50),
                              kCTForegroundColorAttributeName, red);

// Create the framesetter with the attributed string.
CTFramesetterRef framesetter =
    CTFramesetterCreateWithAttributedString(attrString);
CFRelease(attrString);

// Create the frame and draw it into the graphics context
CTFrameRef frame = CTFramesetterCreateFrame(framesetter,
                                             CFRangeMake(0, 0), path, NULL);

CFRelease(framesetter);
CTFrameDraw(frame, context);
CFRelease(frame);

```

Simple Text Labels

Another very common typesetting operation is drawing a single line of text to use as a label for a user-interface element. In Core Text this requires only two lines of code, one to create the line object with an attributed string and another to draw the line into a graphic context.

Listing 2-2 omits initialization of the plain text string, font, and graphics context, but it shows how to create an attributes dictionary and use it to create the attributed string. To see how to create a Core Text font, see [“Font Creation and Storage”](#) (page 21).

Listing 2-2 Typesetting a simple text label

```

CFStringRef string; CTFontRef font; CGContextRef context;
// Initialize string, font, and context
CFStringRef keys[] = { kCTFontAttributeName };
CFTyperef values[] = { font };

CFDictionaryRef attributes =
    CFDictionaryCreate(kCFAllocatorDefault, (const void**)&keys,
                     (const void**)&values, sizeof(keys) / sizeof(keys[0]),
                     &kCFTypedictionaryCallbacks,
                     &kCFTypedictionaryValueCallbacks);

CFAttributedStringRef attrString =
    CFAttributedStringCreate(kCFAllocatorDefault, string, attributes);
CFRelease(string);
CFRelease(attributes);

CTLineRef line = CTLineCreateWithAttributedString(attrString);

// Set text position and draw the line into the graphics context
CGContextSetTextPosition(context, 10.0, 10.0);
CTLineDraw(line, context);
CFRelease(line);

```

Columnar Layout

Laying out text in multiple columns is another common typesetting operation. Strictly speaking, Core Text itself only performs the layout of one column at a time and does not calculate the column sizes or locations. You do those operations before calling Core Text to lay out the text within the rectangular path area you've calculated. In this sample, Core Text, in addition to laying out the text in each column, also provides the subrange within the text string for each column.

Listing 2-3 mixes Cocoa method calls in Objective-C with function calls into Carbon frameworks and Core Text. It includes an implementation of the `drawRect:` method of `NSView`, which calls the local `createColumns` function, defined first in this listing. This code resides in an `NSView` subclass in a Cocoa document-based application. The `NSView` subclass includes an `attributedString` accessor method, which is not shown here but is called in this listing to return the attributed string to be laid out.

Listing 2-3 Performing columnar text layout

```
- (NSArrayRef)createColumns {
    CGRect bounds = CGRectMake(0, 0, NSWidth([self bounds]),
                               NSHeight([self bounds]));

    int column;
    CGRect* columnRects = (CGRect*)calloc(_columnCount, sizeof(*columnRects));

    // Start by setting the first column to cover the entire view.
    columnRects[0] = bounds;
    // Divide the columns equally across the frame's width.
    CGFloat columnWidth = CGRectGetGetWidth(bounds) / _columnCount;
    for (column = 0; column < _columnCount - 1; column++) {
        CGRectDivide(columnRects[column], &columnRects[column],
                    &columnRects[column + 1], columnWidth, CGRectMinXEdge);
    }

    // Inset all columns by a few pixels of margin.
    for (column = 0; column < _columnCount; column++) {
        columnRects[column] = CGRectInset(columnRects[column], 10.0, 10.0);
    }

    // Create an array of layout paths, one for each column.
    CFMutableArrayRef array = CFArrayCreateMutable(kCFAllocatorDefault,
                                                  _columnCount, &kCFTypeArrayCallbacks);
    for (column = 0; column < _columnCount; column++) {
        CGMutablePathRef path = CGPathCreateMutable();
        CGPathAddRect(path, NULL, columnRects[column]);
        CFArrayInsertValueAtIndex(array, column, path);
        CFRelease(path);
    }
    free(columnRects);
    return array;
}

- (void)drawRect:(NSRect)rect {
    // Draw a white background.
    [[NSColor whiteColor] set];
    [NSBezierPath fillRect:[self bounds]];

    // Initialize the text matrix to a known value.
```

```

CGContextRef context = (CGContextRef)[[NSGraphicsContext currentContext]
                                     graphicsPort];
CGContextSetTextMatrix(context, CGAffineTransformIdentity);

CTFramesetterRef framesetter =
    CTFramesetterCreateWithAttributedString(
        (CFAttributedStringRef)[self attributedString]);
CFArrayRef columnPaths = [self createColumns];

CFIndex pathCount = CFArrayGetCount(columnPaths);
CFIndex startIndex = 0;
int column;
for (column = 0; column < pathCount; column++) {
    CGPathRef path = (CGPathRef)CFArrayGetValueAtIndex(columnPaths, column);

    // Create a frame for this column and draw it.
    CTFrameRef frame = CTFramesetterCreateFrame(framesetter,
        CFRangeMake(startIndex, 0), path, NULL);
    CTFrameDraw(frame, context);

    // Start the next frame at the first character not visible in this frame.
    CFRange frameRange = CTFrameGetVisibleStringRange(frame);
    startIndex += frameRange.length;

    CFRelease(frame);
}
CFRelease(columnPaths);
}

```

Manual Line Breaking

You usually don't need to do manual line breaking unless you have a special hyphenation process or a similar requirement. A framesetter performs line breaking automatically. Listing 2-4 shows how to create a typesetter, an object used by the framesetter, and use it directly to find appropriate line breaks and create a typeset line manually. This sample also shows how to center the line before drawing.

Listing 2-4 Performing manual line breaking

```

double width; CGContextRef context; CGPoint textPosition; CFAttributedStringRef
attrString;
// Initialize those variables.

// Create a typesetter using the attributed string.
CTTypesetterRef typesetter = CTTypesetterCreateWithAttributedString(attrString);

// Find a break for line from the beginning of the string to the given width.
CFIndex start = 0;
CFIndex count = CTTypesetterSuggestLineBreak(typesetter, start, width);

// Use the returned character count (to the break) to create the line.
CTLineRef line = CTTypesetterCreateLine(typesetter, CFRangeMake(start, count));

// Get the offset needed to center the line.
float flush = 0.5; // centered

```

```
double penOffset = CTLineGetPenOffsetForFlush(line, flush, width);

// Move the given text drawing position by the calculated offset and draw the line.
CGContextSetTextPosition(context, textPosition.x + penOffset, textPosition.y);
CTLineDraw(line, context);

// Move the index beyond the line break.
start += count;
```

Font Creation and Storage

The example function in Listing 2-5 creates a font descriptor from a PostScript font name and a float specifying the point size.

Listing 2-5 Creating a font descriptor from a name and point size

```
CTFontDescriptorRef CreateFontDescriptorFromName(CFStringRef iPostScriptName,
                                                CGFloat iSize)
{
    assert(iPostScriptName != NULL);
    return CTFontDescriptorCreateWithNameAndSize(iPostScriptName, iSize);
}
```

The example function in Listing 2-6 creates a font descriptor from a font family name and font traits.

Listing 2-6 Creating a font descriptor from a family and traits

```
CTFontDescriptorRef CreateFontDescriptorFromFamilyAndTraits(CFStringRef iFamilyName,
                                                         CTFontSymbolicTraits iTraits, CGFloat iSize)
{
    CTFontDescriptorRef descriptor = NULL;
    CFMutableDictionaryRef attributes;

    assert(iFamilyName != NULL);
    // Create a mutable dictionary to hold our attributes.
    attributes = CFDictionaryCreateMutable(kCFAllocatorDefault, 0,
                                         &kCFTypedictionaryKeyCallbacks, &kCFTypedictionaryValueCallbacks);
    check(attributes != NULL);

    if (attributes != NULL) {
        CFMutableDictionaryRef traits;
        CFNumberRef symTraits;

        // Add a family name to our attributes.
        CFDictionaryAddValue(attributes, kCTFontFamilyNameAttribute, iFamilyName);

        // Create the traits dictionary.
        symTraits = CFNumberCreate(kCFAllocatorDefault, kCFNumberSInt32Type,
                                 &iTraits);
        check(symTraits != NULL);

        if (symTraits != NULL) {
            // Create a dictionary to hold our traits values.
            traits = CFDictionaryCreateMutable(kCFAllocatorDefault, 0,
```

```

        &kCFTypeDictionaryKeyCallbacks, &kCFTypeDictionaryValueCallbacks);
    check(traits != NULL);

    if (traits != NULL) {
        // Add the symbolic traits value to the traits dictionary.
        CFDictionaryAddValue(traits, kCTFontSymbolicTrait, symTraits);

        // Add the traits attribute to our attributes.
        CFDictionaryAddValue(attributes, kCTFontTraitsAttribute, traits);
        CFRelease(traits);
    }
    CFRelease(symTraits);
}
// Create the font descriptor with our attributes and input size.
descriptor = CTFontDescriptorCreateWithAttributes(attributes);
check(descriptor != NULL);

CFRelease(attributes);
}
// Return our font descriptor.
return descriptor;
}

```

The example function in Listing 2-7 creates a font from a provided font descriptor. It calls `CTFontCreateWithFontDescriptor`, passing `NULL` for the matrix parameter to specify the default (identity) matrix.

Listing 2-7 Creating a font from a font descriptor

```

CTFontRef CreateFont(CTFontDescriptorRef iFontDescriptor, CGFloat iSize)
{
    check(iFontDescriptor != NULL);

    // Create the font from the font descriptor and input size. Pass
    // NULL for the matrix parameter to use the default matrix (identity).

    return CTFontCreateWithFontDescriptor(iFontDescriptor, iSize, NULL);
}

```

The example function in Listing 2-8 creates XML data to serialize a font for embedding in a document. Alternatively, and preferably, `NSArchiver` could be used. This is just one way to accomplish this task, but it preserves all data from the font needed to re-create the exact font at a later time.

Listing 2-8 Serializing a font

```

CFDataRef CreateFlattenedFontData(CTFontRef iFont)
{
    CFDataRef          result = NULL;
    CTFontDescriptorRef descriptor;
    CFDictionaryRef    attributes;

    check(iFont != NULL);

    // Get the font descriptor for the font.
    descriptor = CTFontCopyFontDescriptor(iFont);
    check(descriptor != NULL);
}

```

```

if (descriptor != NULL) {
    // Get the font attributes from the descriptor. This should be enough
    // information to recreate the descriptor and the font later.
    attributes = CTFontDescriptorCopyAttributes(descriptor);
    check(attributes != NULL);

    if (attributes != NULL) {
        // If attributes are a valid property list, directly flatten
        // the property list. Otherwise we may need to analyze the attributes
        // and remove or manually convert them to serializable forms.
        // This is left as an exercise for the reader.
        if (CFPropertyListIsValid(attributes, kCFPropertyListXMLFormat_v1_0)) {
            result = CFPropertyListCreateXMLData(kCFAllocatorDefault,
                                                attributes);

            check(result != NULL);
        }
    }
}
return result;
}

```

The example function in Listing 2-9 creates a font reference from flattened XML data. It shows how to unflatten font attributes and create a font with those attributes.

Listing 2-9 Creating a font from serialized data

```

CTFontRef CreateFontFromFlattenedFontData(CFDataRef iData)
{
    CTFontRef          font = NULL;
    CFDictionaryRef    attributes;
    CTFontDescriptorRef descriptor;

    check(iData != NULL);

    // Create our font attributes from the property list. We will create
    // an immutable object for simplicity, but if you needed to massage
    // the attributes or convert certain attributes from their serializable
    // form to the Core Text usable form, you could do it here.
    attributes =
        (CFDictionaryRef)CFPropertyListCreateFromXMLData(kCFAllocatorDefault,
                                                         iData, kCFPropertyListImmutable, NULL);
    check(attributes != NULL);

    if (attributes != NULL) {
        // Create the font descriptor from the attributes.
        descriptor = CTFontDescriptorCreateWithAttributes(attributes);
        check(descriptor != NULL);

        if (descriptor != NULL) {
            // Create the font from the font descriptor. We will use
            // 0.0 and NULL for the size and matrix parameters. This
            // causes the font to be created with the size and/or matrix
            // that exist in the descriptor, if present. Otherwise default
            // values are used.
            font = CTFontCreateWithFontDescriptor(descriptor, 0.0, NULL);
            check(font != NULL);
        }
    }
}

```

```
    return font;
}
```

Accessing Font Metrics

For every font, glyph designers provide a set of measurements, called metrics, which describe the spacing around each glyph in the font. The typesetter uses these metrics to determine glyph placement. Font metrics are parameters such as ascent, descent, leading, cap height, x-height, and so on.

The sample functions in this section illustrate how to query a font for its font metric data. The example function in Listing 2-10 shows how to use line metrics accessors to calculate the line height for a font. In most cases you should not need to do this yourself. If you have a `CTLineRef` object for a line of text, you could call `CTLineGetTypographicBounds` to get the line metrics for the line.

Listing 2-10 Calculating line height

```
CGFloat GetLineHeightForFont(CTFontRef iFont)
{
    CGFloat lineHeight = 0.0;

    check(iFont != NULL);

    // Get the ascent from the font, already scaled for the font's size
    lineHeight += CTFontGetAscent(iFont);

    // Get the descent from the font, already scaled for the font's size
    lineHeight += CTFontGetDescent(iFont);

    // Get the leading from the font, already scaled for the font's size
    lineHeight += CTFontGetLeading(iFont);

    return lineHeight;
}
```

The example function in Listing 2-11 demonstrates how to get glyphs for the characters in a string with a single font. Most of the time you should just use a `CTLine` object to get this information because one font may not encode the entire string. In addition, simple character-to-glyph mapping will not get the correct appearance for complex scripts. This simple glyph mapping may be appropriate if you are trying to display specific Unicode characters for a font.

Listing 2-11 Getting glyphs for characters

```
void GetGlyphsForCharacters(CTFontRef iFont, CFStringRef iString)
{
    UniChar *characters;
    CGGlyph *glyphs;
    CFIndex count;

    assert(iFont != NULL && iString != NULL);

    // Get our string length.
    count = CFStringGetLength(iString);
```

```

// Allocate our buffers for characters and glyphs.
characters = (UniChar *)malloc(sizeof(UniChar) * count);
assert(characters != NULL);

glyphs = (CGGlyph *)malloc(sizeof(CGGlyph) * count);
assert(glyphs != NULL);

// Get the characters from the string.
CFStringGetCharacters(iString, CFRangeMake(0, count), characters);

// Get the glyphs for the characters.
CTFontGetGlyphsForCharacters(iFont, characters, glyphs, count);

// Do something with the glyphs here, if a character is unmapped

// Free our buffers
free(characters);
free(glyphs);
}

```

Creating Related Fonts

The example functions in this section show how to query fonts for their attributes. The example function in Listing 2-12 makes a font bold or unbold based on the value of the Boolean parameter passed with the function call. If the current font family does not have the requested style, the function returns `NULL`.

Listing 2-12 Changing traits of a font

```

CTFontRef CreateBoldFont(CTFontRef iFont, Boolean iMakeBold)
{
    CTFontSymbolicTraits desiredTrait = 0;
    CTFontSymbolicTraits traitMask;

    // If we are trying to make the font bold, set the desired trait
    // to be bold.
    if (iMakeBold)
        desiredTrait = kCTFontBoldTrait;

    // Mask off the bold trait to indicate that it is the only trait
    // desired to be modified. As CTFontSymbolicTraits is a bit field,
    // we could choose to change multiple traits if we desired.
    traitMask = kCTFontBoldTrait;

    // Create a copy of the original font with the masked trait set to the
    // desired value. If the font family does not have the appropriate style,
    // this will return NULL.

    return CTFontCreateCopyWithSymbolicTraits(iFont, 0.0, NULL, desiredTrait, traitMask);
}

```

The example function in Listing 2-13 converts a given font to a similar font in another font family, preserving traits if possible. It may return `NULL`.

Listing 2-13 Converting a font to another family

```
CTFontRef CreateFontConvertedToFamily(CTFontRef iFont, CFStringRef iFamily)
{
    // Create a copy of the original font with the new family. This call
    // attempts to preserve traits, and may return NULL if that is not possible.
    // Pass in 0.0 and NULL for size and matrix to preserve the values from
    // the original font.

    return CTFontCreateCopyWithFamily(iFont, 0.0, NULL, iFamily);
}
```

Document Revision History

This table describes the changes to *Core Text Programming Guide*.

Date	Notes
2008-06-09	Revised Figure 1-3 "Text layout data flow" to show that a CGPath object is provided in the creation of a CTFrame object.
2008-02-08	Fixed bad link to sample code in Introduction.
2007-12-11	Made minor editorial corrections.
2007-07-16	New document that explains how to perform text layout and font-related operations using the Core Text programming interfaces.

REVISION HISTORY

Document Revision History