Common Font Operations

This chapter describes some common font-handling operations and shows how to code them using Core Text. These operations are the same on iOS and OS X. The following operations with code listings are included in this chapter:

- Creating Font Descriptors
- Creating a Font from a Font Descriptor
- Creating Related Fonts
- Serializing a Font
- Creating a Font from Serialized Data
- Changing Kerning
- Getting Glyphs for Characters

Creating Font Descriptors

The example function in Listing 3-1 creates a font descriptor from parameter values specifying a PostScript font name and the point size.

Listing 3-1 Creating a font descriptor from a name and point size

```
CTFontDescriptorRef CreateFontDescriptorFromName(CFStringRef
postScriptName,
                                                 CGFloat size)
{
    return CTFontDescriptorCreateWithNameAndSize(postScriptName,
size);
}
```

The example function in Listing 3-2 creates a font descriptor from a font family name and font traits.

Listing 3-2 Creating a font descriptor from a family and traits

```
NSString* familyName = @"Papyrus";
CTFontSymbolicTraits symbolicTraits = kCTFontTraitCondensed;
CGFloat size = 24.0;
```

```
NSMutableDictionary* attributes = [NSMutableDictionary
dictionary];
[attributes setObject:familyName forKey:
(id)kCTFontFamilyNameAttribute];


// The attributes dictionary contains another dictionary, the
traits dictionary,
// which in this example specifies only the symbolic traits.
NSMutableDictionary* traits = [NSMutableDictionary dictionary];
[traits setObject:[NSNumber numberWithUnsignedInt:symbolicTraits]
                                            forKey:
(id)kCTFontSymbolicTrait];


[attributes setObject:traits forKey:(id)kCTFontTraitsAttribute];
[attributes setObject:[NSNumber numberWithFloat:size]
                                           forKey:
(id)kCTFontSizeAttribute];


CTFontDescriptorRef descriptor =

CTFontDescriptorCreateWithAttributes((CFDictionaryRef)attributes);
CFRelease(descriptor);
```

Creating a Font from a Font Descriptor

Listing 3-3 shows how to create a font descriptor and use it to create a font. When you call `CTFontCreateWithFontDescriptor`, you usually pass `NULL` for the matrix parameter to specify the default (identity) matrix. The size and matrix (second and third) parameters of `CTFontCreateWithFontDescriptor` override any specified in the font descriptor unless they are unspecified (`0.0` for size and `NULL` for matrix).

Listing 3-3 Creating a font from a font descriptor

```
NSDictionary *fontAttributes =
                  [NSDictionary dictionaryWithObjectsAndKeys:
                           @"Courier", (NSString
*)kCTFontFamilyNameAttribute,
                           @"Bold", (NSString
*)kCTFontStyleNameAttribute,
                           [NSNumber numberWithFloat:16.0],
```

```
                            (NSString *)kCTFontSizeAttribute,
                            nil];
// Create a descriptor.
CTFontDescriptorRef descriptor =

CTFontDescriptorCreateWithAttributes((CFDictionaryRef)fontAttributes);


// Create a font using the descriptor.
CTFontRef font = CTFontCreateWithFontDescriptor(descriptor, 0.0,
NULL);
CFRelease(descriptor);
```

Creating Related Fonts

It is often useful to convert an existing font to a related or similar font. The example function in Listing 3-4 shows how to make 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 3-4 Changing traits of a font

```
CTFontRef CreateBoldFont(CTFontRef font, Boolean makeBold)
{
    CTFontSymbolicTraits desiredTrait = 0;
    CTFontSymbolicTraits traitMask;


    // If requesting that the font be bold, set the desired trait
    // to be bold.
    if (makeBold) desiredTrait = kCTFontBoldTrait;


    // Mask off the bold trait to indicate that it is the only
trait
    // to be modified. As CTFontSymbolicTraits is a bit field,
    // could change multiple traits if 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,
```

```
    // returns NULL.


    return CTFontCreateCopyWithSymbolicTraits(font, 0.0, NULL,
desiredTrait, traitMask);

}
```

The example function in Listing 3-8 converts a given font to a similar font in another font family, preserving traits if possible. It may return `NULL`. Passing in `0.0` for the size parameter and `NULL` for the matrix parameter preserves the size from the original font.

Listing 3-5 Converting a font to another family

```
CTFontRef CreateFontConvertedToFamily(CTFontRef font, CFStringRef
family)

{

    // 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(font, 0.0, NULL, family);

}
```

Serializing a Font

The example function in Listing 3-6 shows how to create XML data to serialize a font that can be embedded 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 recreate the exact font at a later time.

Listing 3-6 Serializing a font

```
CFDataRef CreateFlattenedFontData(CTFontRef font)

{

    CFDataRef           result = NULL;

    CTFontDescriptorRef descriptor;

    CFDictionaryRef     attributes;
```

```
    // Get the font descriptor for the font.
    descriptor = CTFontCopyFontDescriptor(font);

    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);

        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);
            }
        }
    }
    return result;
}
```

Creating a Font from Serialized Data

The example function in Listing 3-7 shows how to create a font reference from flattened XML data. It shows how to unflatten font attributes and create a font with those attributes.

Listing 3-7 Creating a font from serialized data

```
CTFontRef CreateFontFromFlattenedFontData(CFDataRef iData)
{
    CTFontRef           font = NULL;
```

```
    CFDictionaryRef     attributes;
    CTFontDescriptorRef descriptor;

    // Create our font attributes from the property list.
    // For simplicity, this example creates an immutable object.
    // If you needed to massage or convert certain attributes
    // from their serializable form to the Core Text usable form,
    // do it here.
    attributes =
          (CFDictionaryRef)CFPropertyListCreateFromXMLData(
                                kCFAllocatorDefault,
                                iData, kCFPropertyListImmutable,
NULL);
    if (attributes != NULL) {
        // Create the font descriptor from the attributes.
        descriptor =
CTFontDescriptorCreateWithAttributes(attributes);
        if (descriptor != NULL) {
            // Create the font from the font descriptor. This
sample uses
            // 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);
        }
    }
    return font;
}
```

Changing Kerning

Ligatures and kerning are enabled by default. To disable, set the `kCTKernAttributeName` attribute to `0`. Listing 3-8 sets the kern size to a large number for the first few characters drawn.

Listing 3-8 Setting kerning

```
// Set the color of the first 13 characters to red
// using a previously defined red CGColor object.
CFAttributedStringSetAttribute(attrString, CFRangeMake(0, 13),
kCTForegroundColorAttributeName, red);

// Set kerning between the first 18 chars to be 20
CGFloat otherNum = 20;
CFNumberRef otherCFNum = CFNumberCreate(NULL,
kCFNumberCGFloatType, &otherNum);
CFAttributedStringSetAttribute(attrString, CFRangeMake(0,18),
                                        kCTKernAttributeName,
otherCFNum);
```

Getting Glyphs for Characters

Listing 3-9 shows 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 3-9 Getting glyphs for characters

```
void GetGlyphsForCharacters(CTFontRef font, CFStringRef string)
{
    // Get the string length.
    CFIndex count = CFStringGetLength(string);

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

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

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

    // Do something with the glyphs here. Characters not mapped by this font will be zero.
    // ...

    // Free the buffers
    free(characters);
    free(glyphs);
}
```