Image I/O Programming Guide

Developer

Contents

Introduction 4
Who Should Read This Document? 4
Organization of This Document 4
See Also 4

Basics of Using Image I/O 5
Using the Image I/O Framework in Your Application 5
Supported Image Formats 5

Creating and Using Image Sources 7
Creating an Image from an Image Source 7
Creating a Thumbnail Image from an Image Source 9
Incrementally Loading an Image 11
Displaying Image Properties 11

Working with Image Destinations 16
Setting the Properties of an Image Destination 16
Writing an Image to an Image Destination 17

Document Revision History 19

Figures, Tables, and Listings

Basics of Using Image I/O 5
Table 1-1 Common uniform type identifiers (UTIs) and image content type constants 6
Listing 1-1 Getting and printing supported UTIs 6

Creating and Using Image Sources 7
Figure 2-1 An Info window that displays image properties 12
Listing 2-1 Creating an image from an image source 7
Listing 2-2 Creating a thumbnail image 9
Listing 2-3 A routine that creates an image source and retrieves properties 12

Working with Image Destinations 16
Listing 3-1 Setting the properties of an image destination 17
Listing 3-2 A method that writes an image to a URL 18

Introduction

The Image I/O programming interface allows applications to read and write most image file formats. Originally part of the Core Graphics framework, Image I/O resides in its own framework to allow developers to use it independently of Core Graphics (Quartz 2D). Image I/O provides the definitive way to access image data because it is highly efficient, allows easy access to metadata, and provides color management.

The Image I/O interface is available in OS X v10.4 and later and in iOS 4 and later.

Who Should Read This Document?

This document is intended for developers who read or write image data in an application. Any developer currently using image importers or other image handling libraries should read this document to see how to use the Image I/O framework instead.

Organization of This Document

This document is organized into the following chapters:

- "Basics of Using Image I/O" (page 5) discusses supported image formats and shows how to include the framework in an Xcode project.
- "Creating and Using Image Sources" (page 7) shows how to create an image source, create an image from it, and extract properties for display in the user interface.
- "Working with Image Destinations" (page 16) provides information on creating an image destination, setting up its properties, and adding an image to it.

See Also

The *Image I/O Reference Collection* provides detailed descriptions of the functions, data types, and constants in the Image I/O framework.

Basics of Using Image I/O

The Image I/O framework provides opaque data types for reading image data from a source (`CGImageSourceRef`) and writing image data to a destination (`CGImageDestinationRef`). It supports a wide range of image formats, including the standard web formats, high dynamic range images, and raw camera data. Image I/O has many other features such as:

- The fastest image decoders and encoders for the Mac platform
- The ability to load images incrementally
- Support for image metadata
- Effective caching

You can create image source and image destination objects from:

- URLs. Images whose location can be specified as a URL can act as a supplier or receiver of image data. In Image I/O, a URL is represented as the Core Foundation data type `CFURLRef`.
- The Core Foundation objects `CFDataRef` and `CFMutableDataRef`.
- Quartz data consumer (`CGDataConsumerRef`) and data provider (`CGDataProviderRef`) objects.

Using the Image I/O Framework in Your Application

Image I/O resides in the Application Services framework in OS X, and in the Image I/O framework in iOS. After adding the framework to your application, import the header file by including this statement:

```
#import <ImageIO/ImageIO.h>
```

Supported Image Formats

The Image I/O framework understands most of the common image file formats, such as JPEG, JPEG2000, RAW, TIFF, BMP, and PNG. Not all formats are supported on each platform. For the most up-to-date list of what Image I/O supports, you can call the these functions:

- `CGImageSourceCopyTypeIdentifiers` returns an array of the Uniform Type Identifiers (UTIs) that Image I/O supports as image sources.

- `CGImageDestinationCopyTypeIdentifiers` returns an array of the uniform type identifiers (UTIs) that Image I/O supports as image destinations.

You can then use the `CFShow` function to print the array to the debugger console in Xcode, as shown in Listing 1-1. The strings in the array returned by these functions take the form of `com.apple.pict`, `public.jpeg`, `public.tiff`, and so on. Table 1-1 (page 6) lists the UTIs for many common image file formats. OS X and iOS define constants for most common image file formats; The full set of constants are declared in the `UTCoreTypes.h` header file. You can use these constants when you need to specify an image type, either as a hint for an image source (`kCGImageSourceTypeIdentifierHint`) or as an image type for an image destination.

Listing 1-1 Getting and printing supported UTIs

```
CFArrayRef mySourceTypes = CGImageSourceCopyTypeIdentifiers();
CFShow(mySourceTypes);
CFArrayRef myDestinationTypes = CGImageDestinationCopyTypeIdentifiers();
CFShow(myDestinationTypes);
```

Table 1-1 Common uniform type identifiers (UTIs) and image content type constants

Uniform type identifier	Image content type constant
public.image	`kUTTypeImage`
public.png	`kUTTypePNG`
public.jpeg	`kUTTypeJPEG`
public.jpeg-2000 (OS X only)	`kUTTypeJPEG2000`
public.tiff	`kUTTypeTIFF`
com.apple.pict (OS X only)	`kUTTypePICT`
com.compuserve.gif	`kUTTypeGIF`

Creating and Using Image Sources

An image source abstracts the data-access task and eliminates the need for you to manage data through a raw memory buffer. An image source can contain more than one image, thumbnail images, properties for each image, and the image file. When you are working with image data and your application runs in OS X v10.4 or later, image sources are the preferred way to move image data into your application. After creating a `CGImageSource` object, you can obtain images, thumbnails, image properties, and other image information using the functions described in *CGImageSource Reference*.

Creating an Image from an Image Source

One of the most common tasks you'll perform with the Image I/O framework is to create an image from an image source, similar to what's shown in Listing 2-1. This example shows how to create an image source from a path name and then extract the image. When you create an image source object, you can provide a hint as to the format of the image source file.

When you create an image from an image source, you must specify an index and you can provide a dictionary of properties (key-value pairs) to specify such things as whether to create a thumbnail or allow caching. *CGImageSource Reference* and *CGImageProperties Reference* list keys and the expected data type of the value for each key.

You need to supply an index value because some image file formats allow multiple images to reside in the same source file. For an image source file that contains only one image, pass `0`. You can find out the number of images in an image source file by calling the function `CGImageSourceGetCount`.

Listing 2-1 Creating an image from an image source

```
CGImageRef MyCreateCGImageFromFile (NSString* path)
{
    // Get the URL for the pathname passed to the function.
    NSURL *url = [NSURL fileURLWithPath:path];
    CGImageRef        myImage = NULL;
    CGImageSourceRef  myImageSource;
    CFDictionaryRef   myOptions = NULL;
    CFStringRef       myKeys[2];
```

```
CFTypeRef         myValues[2];

// Set up options if you want them. The options here are for
// caching the image in a decoded form and for using floating-point
// values if the image format supports them.
myKeys[0] = kCGImageSourceShouldCache;
myValues[0] = (CFTypeRef)kCFBooleanTrue;
myKeys[1] = kCGImageSourceShouldAllowFloat;
myValues[1] = (CFTypeRef)kCFBooleanTrue;
// Create the dictionary
myOptions = CFDictionaryCreate(NULL, (const void **) myKeys,
               (const void **) myValues, 2,
               &kCFTypeDictionaryKeyCallBacks,
               & kCFTypeDictionaryValueCallBacks);
// Create an image source from the URL.
myImageSource = CGImageSourceCreateWithURL((CFURLRef)url, myOptions);
CFRelease(myOptions);
// Make sure the image source exists before continuing
if (myImageSource == NULL){
    fprintf(stderr, "Image source is NULL.");
    return  NULL;
}
// Create an image from the first item in the image source.
myImage = CGImageSourceCreateImageAtIndex(myImageSource,
                                          0,
                                          NULL);

CFRelease(myImageSource);
// Make sure the image exists before continuing
if (myImage == NULL){
     fprintf(stderr, "Image not created from image source.");
     return NULL;
}
```

```
    return myImage;
}
```

Creating a Thumbnail Image from an Image Source

Some image source files contain thumbnail images that you can retrieve. If thumbnails aren't already present, Image I/O gives you the option of creating them. You can also specify a maximum thumbnail size and whether to apply a transform to the thumbnail image.

Listing 2-2 shows how to create an image source from data, set up a dictionary that contains options related to the thumbnail, and then create a thumbnail image. You use the `kCGImageSourceCreateThumbnailWithTransform` key to specify whether the thumbnail image should be rotated and scaled to match the orientation and pixel aspect ratio of the full image.

Listing 2-2 Creating a thumbnail image

```
CGImageRef MyCreateThumbnailImageFromData (NSData * data, int imageSize)
{
    CGImageRef        myThumbnailImage = NULL;
    CGImageSourceRef  myImageSource;
    CFDictionaryRef   myOptions = NULL;
    CFStringRef       myKeys[3];
    CFTypeRef         myValues[3];
    CFNumberRef       thumbnailSize;

   // Create an image source from NSData; no options.
   myImageSource = CGImageSourceCreateWithData((CFDataRef)data,
                                               NULL);
   // Make sure the image source exists before continuing.
   if (myImageSource == NULL){
        fprintf(stderr, "Image source is NULL.");
        return  NULL;
   }

   // Package the integer as a  CFNumber object. Using CFTypes allows you
   // to more easily create the options dictionary later.
```

```
    thumbnailSize = CFNumberCreate(NULL, kCFNumberIntType, &imageSize);

    // Set up the thumbnail options.
    myKeys[0] = kCGImageSourceCreateThumbnailWithTransform;
    myValues[0] = (CFTypeRef)kCFBooleanTrue;
    myKeys[1] = kCGImageSourceCreateThumbnailFromImageIfAbsent;
    myValues[1] = (CFTypeRef)kCFBooleanTrue;
    myKeys[2] = kCGImageSourceThumbnailMaxPixelSize;
    myValues[2] = (CFTypeRef)thumbnailSize;

    myOptions = CFDictionaryCreate(NULL, (const void **) myKeys,
                    (const void **) myValues, 2,
                    &kCFTypeDictionaryKeyCallBacks,
                    & kCFTypeDictionaryValueCallBacks);

  // Create the thumbnail image using the specified options.
  myThumbnailImage = CGImageSourceCreateThumbnailAtIndex(myImageSource,
                                          0,
                                          myOptions);
  // Release the options dictionary and the image source
  // when you no longer need them.
  CFRelease(thumbnailSize);
  CFRelease(myOptions);
  CFRelease(myImageSource);

    // Make sure the thumbnail image exists before continuing.
    if (myThumbnailImage == NULL){
          fprintf(stderr, "Thumbnail image not created from image source.");
          return NULL;
    }

    return myThumbnailImage;
}
```

Incrementally Loading an Image

If you have a very large image, or are loading image data over the web, you may want to create an incremental image source so that you can draw the image data as you accumulate it. You need to perform the following tasks to load an image incrementally from a CFData object:

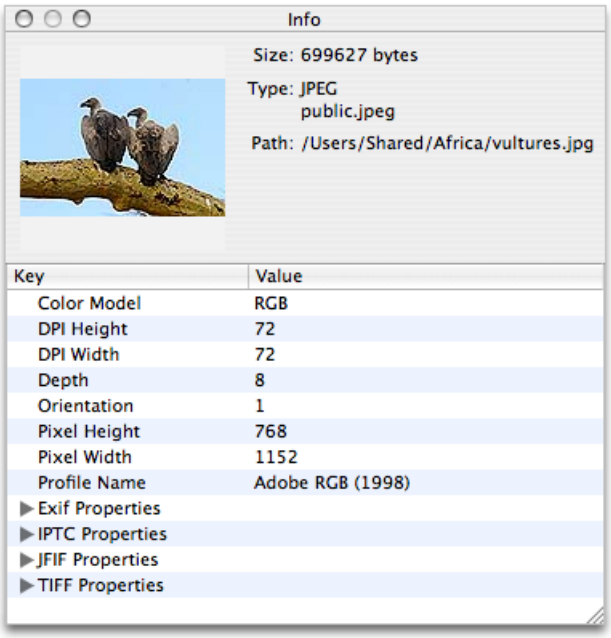
1. Create the CFData object for accumulating the image data.
2. Create an incremental image source by calling the function `CGImageSourceCreateIncremental`.
3. Add image data to the CFData object.
4. Call the function `CGImageSourceUpdateData`, passing the CFData object and a Boolean value (`bool` data type) that specifies whether the data parameter contains the entire image, or just partial image data. In any case, the data parameter must contain all the image file data accumulated up to that point.
5. If you have accumulated enough image data, create an image by calling `CGImageSourceCreateImageAtIndex`, draw the partial image, and then release it.
6. Check to see if you have all the data for an image by calling the function `CGImageSourceGetStatusAtIndex`. If the image is complete, this function returns `kCGImageStatusComplete`. If the image is not complete, repeat steps 3 and 4 until it is.
7. Release the incremental image source.

Displaying Image Properties

Digital photos are tagged with a wealth of information about the image—image dimensions, resolution, orientation, color profile, aperture, metering mode, focal length, creation date, keywords, caption, and much more. This information is extremely useful for image handling and editing, but only if the data is exposed in the user interface. Although the `CGImageSourceCopyPropertiesAtIndex` function retrieves a dictionary of all the properties associated with an image in an image source, you'll need to write code that traverses that dictionary to retrieve and then display that information.

In this section you'll take a close look at a routine from the OS X *ImageApp* sample code, which is an image display application that you can download and experiment with. One of the features of the *ImageApp* sample code is an image Info window that displays a thumbnail image and image properties for the currently active image, as shown in Figure 2-1.

Figure 2-1 An Info window that displays image properties



You can take a look at the `ImageInfoPanel.h` and `ImageInfoPanel.m` files for all the implementation details of this panel; you'll also need to look at the nib file for the project to see how the window and bindings are set up. To get an idea of how you can use CGImageSource functions to support an image editing application, take a look at Listing 2-3. A detailed explanation for each numbered line of code appears following the listing. (Keep in mind that this routine is not a standalone routine—you can't simply paste it into your own program. It is an excerpt from the *ImageApp* sample code.)

Listing 2-3 A routine that creates an image source and retrieves properties

```
- (void) setURL:(NSURL*)url
{
    if ([url isEqual:mUrl])
        return;

    mUrl = url;
```

```
    CGImageSourceRef source = CGImageSourceCreateWithURL((CFURLRef)url, NULL);   // 1
    if (source)
    {
        NSDictionary* props =
           (NSDictionary*) CGImageSourceCopyPropertiesAtIndex(source, 0, NULL);  // 2
        [mTree setContent:[self propTree:props]];                                // 3
        NSDictionary* thumbOpts = [NSDictionary dictionaryWithObjectsAndKeys:
            (id)kCFBooleanTrue, (id)kCGImageSourceCreateThumbnailWithTransform,
          (id)kCFBooleanTrue, (id)kCGImageSourceCreateThumbnailFromImageIfAbsent,
           [NSNumber numberWithInt:128], (id)kCGImageSourceThumbnailMaxPixelSize,
            nil];                                                                // 4
        CGImageRef image = CGImageSourceCreateThumbnailAtIndex(source, 0,
                                    (CFDictionaryRef)thumbOpts);                 // 5
        [mThumbView setImage:image];                                             // 6
        CGImageRelease(image);                                                   // 7
        [mFilePath setStringValue:[mUrl path]];                                  // 8

        NSString* uti = (NSString*)CGImageSourceGetType(source);                // 9
        [mFileType setStringValue:[NSString stringWithFormat:@"%@\n%@",
                        ImageIOLocalizedString(uti), uti]];                      // 10

        CFDictionaryRef fileProps = CGImageSourceCopyProperties(source, nil);    // 11
        [mFileSize setStringValue:[NSString stringWithFormat:@"%@ bytes",
            (id)CFDictionaryGetValue(fileProps, kCGImagePropertyFileSize)]];     // 12
    }
    else                                                                         // 13
    {
        [mTree setContent:nil];
        [mThumbView setImage:nil];
        [mFilePath setStringValue:@""];
        [mFileType setStringValue:@""];
        [mFileSize setStringValue:@""];
    }
}
```

Here's what the code does:

1. Creates an image source object from the URL passed to the routine.
2. Copies the properties for the image located at index location `0`. Some image file formats can support more than one image, but this example assumes a single image (or that the image of interest is always the first one in the file). The `CGImageSourceCopyPropertiesAtIndex` function returns a CFDictionary object. Here, the code casts the CFDictionary as an `NSDictionary` object, as these data types are interchangeable (sometimes referred to as toll-free bridged).

   The dictionary that's returned contains properties that are key-value pairs. However, some of the values are themselves dictionaries that contain properties. Take a look at Figure 2-1 (page 12) and you'll see not only simple key-value pairs (such as Color Model-RGB) but you'll also see Exif properties, IPTC Properties, JFIF Properties, and TIFF Properties, each of which is a dictionary. Clicking a disclosure triangle for one of these displays the properties in that dictionary. You'll need to get these dictionaries and their properties so they can be displayed appropriately in the Info panel. That's what the next step accomplishes.
3. Extracts properties from the dictionary and sets them to a tree controller. If you look at the `ImageInfoPanel.h` file, you'll see that the `mTree` variable is an `NSTreeController` object that is an outlet in Interface Builder. This controller manages a tree of objects. In this case, the objects are properties of the image.

   The `propTree:` method is provided in the `ImageInfoPanel.m` file. It's purpose is to traverse the property dictionary retrieved in the previous step, extract the image properties, and build the array that's bound to the `NSTreeController` object.

   The properties appears in a table of keys and values in Figure 2-1 (page 12).
4. Sets up a dictionary of options to use when creating an image from the image source. Recall that options are passed in a dictionary. The Info panel shown in Figure 2-1 (page 12) displays a thumbnail image. The code here sets up options that create a thumbnail that is rotated and scaled to the same orientation and aspect ratio of the full image. If a thumbnail does not already exist, one is created, and its maximum pixel size is 128 by 128 pixels.
5. Creates a thumbnail image from the first image in the image source, using the options set up in the previous step.
6. Sets the thumbnail image to the view in the Info panel.
7. Releases the image; it is no longer needed.
8. Extracts the path from the URL passed to the method, and sets the string to the text field that's bound to the Info panel. This is the Path text field in Figure 2-1 (page 12).
9. Gets the uniform type identifier of the image source. (This can be different from the type of the images in the source.)

10. Calls a function to retrieve the localized string for the UTI (`ImageIOLocalizedString` is declared in `ImagePanel.m`) and then sets the string to the text field that's bound to the Info panel. This is the Type text field in Figure 2-1 (page 12).
11. Retrieves a dictionary of the properties associated with the image source. These properties apply to the container (such as the file size), not necessarily the individual images in the image source.
12. Retrieves the file size value from the image source dictionary obtained in the previous step, then sets the associated string to the text field that's bound to the Info panel. This is the Size text field shown in Figure 2-1 (page 12).
13. If the source is not created, makes sure that all the fields in the user interface reflect that fact.

Working with Image Destinations

An image destination abstracts the data-writing task and eliminates the need for you to manage data through a raw buffer. An image destination can represent a single image or multiple images. It can contain thumbnail images as well as properties for each image. After creating a CGImageDestination object for the appropriate destination (URL, CFData object, or Quartz data consumer), you can add image data and set image properties. When you are finished adding data, call the function `CGImageDestinationFinalize`.

Setting the Properties of an Image Destination

The function `CGImageDestinationSetProperties` adds a dictionary (`CFDictionaryRef`) of properties (key-value pairs) to the images in an image destination. Although setting properties is optional, there are many situations for which you will want to set them. For example, if your application allows users to add keywords to images or change saturation, exposure, or other values, you'll want to save that information in the options dictionary.

Image I/O defines an extensive set of keys to specify such things as compression quality, background compositing color, Exif dictionary keys, color model values, GIF dictionary keys, Nikon and Canon camera keys, and many more. See *CGImageProperties Reference*.

When setting up the dictionary, you have two choices. You can either create a CFDictionary object or you can create an `NSDictionary` object, then cast it as a `CFDictionaryRef` when you pass the options dictionary to the function `CGImageDestinationSetProperties`. (CFDictionary and `NSDictionary` are interchangeable, or toll-free bridged.) Listing 3-1 shows a code fragment that assigns key-value pairs for three properties, then creates a dictionary that contains those properties. Because this is a code fragment, the necessary calls to release the CFNumber and CFDictionary objects created by the code are not shown. When you write your code, you need to call `CFRelease` when you no longer need each of these objects.

When you set up a key-value pair for a property, you need to consult the reference documentation (see *CGImageDestination Reference* and *CGImageProperties Reference*) for the expected data type of the value. As you can see in Listing 3-1, numerical values typically need to be wrapped in a CFNumber object. When you use Core Foundation types for dictionary values, you can also supply the callback constants when you create the dictionary—`kCFTypeDictionaryKeyCallBacks` and `kCFTypeDictionaryValueCallBacks`. (See *CFDictionary Reference*.)

Listing 3-1 Setting the properties of an image destination

```
float compression = 1.0; // Lossless compression if available.
int orientation = 4; // Origin is at bottom, left.
CFStringRef myKeys[3];
CFTypeRef   myValues[3];
CFDictionaryRef myOptions = NULL;
myKeys[0] = kCGImagePropertyOrientation;
myValues[0] = CFNumberCreate(NULL, kCFNumberIntType, &orientation);
myKeys[1] = kCGImagePropertyHasAlpha;
myValues[1] = kCFBooleanTrue;
myKeys[2] = kCGImageDestinationLossyCompressionQuality;
myValues[2] = CFNumberCreate(NULL, kCFNumberFloatType, &compression);
myOptions = CFDictionaryCreate( NULL, (const void **)myKeys, (const void **)myValues,
 3,
                      &kCFTypeDictionaryKeyCallBacks,
&kCFTypeDictionaryValueCallBacks);
// Release the CFNumber and CFDictionary objects when you no longer need them.
```

Writing an Image to an Image Destination

To write an image to a destination, you first need to create an image destination object by calling the `CGImageDestinationCreateWithURL`, `CGImageDestinationCreateWithData`, or `CGImageDestinationCreateWithDataConsumer` functions. You need to supply the UTI of the resulting image file. You can either supply a UTI or the equivalent constant, if one if available. See Table 1-1 (page 6).

After you create an image destination, you can add an image to it by calling the `CGImageDestinationAddImage` or `CGImageDestinationAddImageFromSource` functions. If the format of the image destination file supports multiple images, you can repeatedly add images. Calling the function `CGImageDestinationFinalize` signals Image I/O that you are finished adding images. Once finalized, you cannot add any more data to the image destination.

Listing 3-2 shows how you might implement a method to write an image file. Although this listing shows how to use an image destination from within an Objective-C method, you can just as easily create and use an image destination in a procedural C function. The options parameter includes any properties you want to specify for the image, such as camera or compression settings.

Listing 3-2 A method that writes an image to a URL

```
- (void) writeCGImage: (CGImageRef) image toURL: (NSURL*) url withType: (CFStringRef)
 imageType andOptions: (CFDictionaryRef) options

{

   CGImageDestinationRef myImageDest = CGImageDestinationCreateWithURL((CFURLRef)url,
 imageType, 1, nil);

   CGImageDestinationAddImage(myImageDest, image, options);

   CGImageDestinationFinalize(myImageDest);

   CFRelease(myImageDest);

}
```

Document Revision History

This table describes the changes to *Image I/O Programming Guide*.

Date	Notes
2010-07-14	Minor edits.
2010-06-25	Revised content to reflect that Image I/O is available on iOS.
2007-07-02	New document that explains how to read and write image data using the Image I/O framework.

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