Quick Look Programming Guide

User Experience > Files & Software Installation

2008-02-08

Contents

Figures, Tables, and Listings

Introduction to Quick Look Programming Guide

Quick Look is a technology introduced in Mac OS X version 10.5 that enables client applications, such as Spotlight and the Finder, to display thumbnail images and full-size previews of documents. For documents of common content types—notably HTML, RTF, plain text, TIFF, PNG, JPEG, PDF, and QuickTime movies—this support is automatic. However, applications with documents that are of less common or even private content types can still take advantage of the Quick Look feature. Those applications can include Quick Look generators: plug-ins that convert a given document from its native format into a format that Quick Look can display to users.

This document describes the Quick Look technology and explains how you, as an application developer, can create a generator so Quick Look can display thumbnail and preview images of your documents. Although Quick Look generators are designed as CFPlugIn-style bundles, all the gritty details of plug-in implementation are handled for you. And although the programmatic interface for Quick Look generators is an ANSI C interface, you can write generators using Objective-C code that calls methods of the Cocoa frameworks.

Organization of This Document

The *Quick Look Programming Guide* has the following chapters:

- "Quick Look and the User Experience" (page 9) describes what the Quick Look technology does and points out the advantages for applications that make use of the technology. it also defines terms that have special meaning in Quick Look.
- "Quick Look Architecture" (page 13) describes the various components of Quick Look, including their roles and how they communicate with each other.
- "Creating and Configuring a Quick Look Project" (page 17) explains how to create a Quick Look generator project and how to specify the properties of a generator.
- "Overview of Generator Implementation" (page 21) summarizes the approaches for generating thumbnails and previews and identifies the best contexts for each approach.
- "Drawing Thumbnails and Previews In a Graphics Context" (page 25) shows how to draw thumbnails and previews in graphics context optimized for bitmap, single-page vector, and multipage vector graphics.
- "Dynamically Generating Previews" (page 29) discusses how you can dynamically generate text-based previews in a supported content type such as RTF or HTML; for HTML previews it also shows how you can include attachments such as images.
- "Saving Previews and Thumbnails in the Document" (page 33) describes the approach where the application saves the thumbnail or preview image in the document and the generator simply retrieves the image for Quick Look. It also describes the function to use when the image data returned to Quick Look is in a format supported by the Image I/O framework.
- "Assigning Core Graphics Images to Thumbnails" (page 37) shows how you can return an image (as a CGImage object) when that image is not in a format supported by the Image I/O framework.

- "Canceling Previews and Thumbnails" (page 39) explains how you can, when requested by Quick Look, cancel the generation of previews and thumbnails.
- "Debugging and Testing a Generator" (page 41) describes the tools and techniques you can use to debug and test a Quick Look generator.

See Also

Consult the following documents for descriptions of Quick Look generator functions and constants:

QLPreviewRequest Reference
QLThumbnailRequest Reference

Because generating a thumbnail or preview image often requires drawing or the creation of an image, the following documents might be of help:

Quartz 2D Programming Guide
Image I/O Reference Collection
Cocoa Drawing Guide
Core Image Programming Guide

Quick Look and the User Experience

Some applications on a Mac OS X system present users with lists of document files. Among these applications are Finder, Spotlight, and Time Machine. These applications show a document icon, the filename, and perhaps metadata related to the document, but often this information is insufficient for users to distinguish one document from another. To identify a particular document by its content in versions of Mac OS X prior to version 10.5, users had to open each document in the list (often requiring them to launch of the application) until they find the one they want. Needless to say, this is a time-consuming procedure.

Quick Look is a feature of Mac OS X introduced in version 10.5 that makes it possible for users to quickly discover the contents of listed documents, both as thumbnail images and as full-size preview images, without requiring the launch of a document's application. The following sections describe the Quick Look feature and identify those applications that are likely candidates for generating Quick Look thumbnails and previews for their documents.

Thumbnails and Previews

Quick Look displays two representations of documents: thumbnails and previews. These two representations fulfill different needs.

A thumbnail is a static image that depicts a document. Although the size may vary, it is typically smaller than a preview and larger than a document icon. The intent of a thumbnail is to give users a notion of a document's contents within a fairly small bounds. At larger sizes, a thumbnail for some kinds of documents—say, image files—might be as useful as a full-size preview. But at smaller sizes, a thumbnail might not be any better than a document icon at conveying what a document contains.

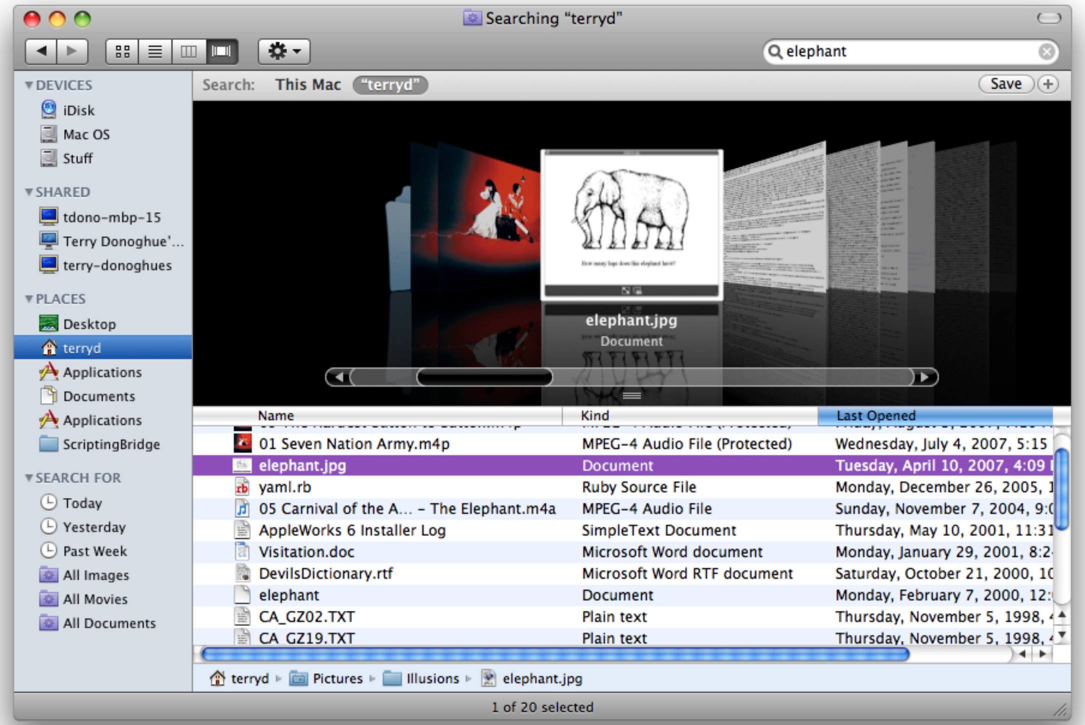
A preview is a larger representation of a document, usually a full-size rendering of it, that is contained by the Quick Look panel (see Figure 1-2 (page 11)). Quick Look displays previews without the need to open the document in its owning application. Users request previews when thumbnails are either not available or do not reveal enough detail to allows them to distinguish one document from another.

A client may display multiple thumbnails at once, while previews are generally displayed one at a time.

Quick Look in Operation

To appreciate how Quick Look contributes to the user experience, let's consider how it is used. If you search for an item in Spotlight—say, "elephant"—and click the Show All option you might see a Finder window like the example in Figure 1-1 .

Figure 1-1 Thumbnails in the Finder's Cover Flow view

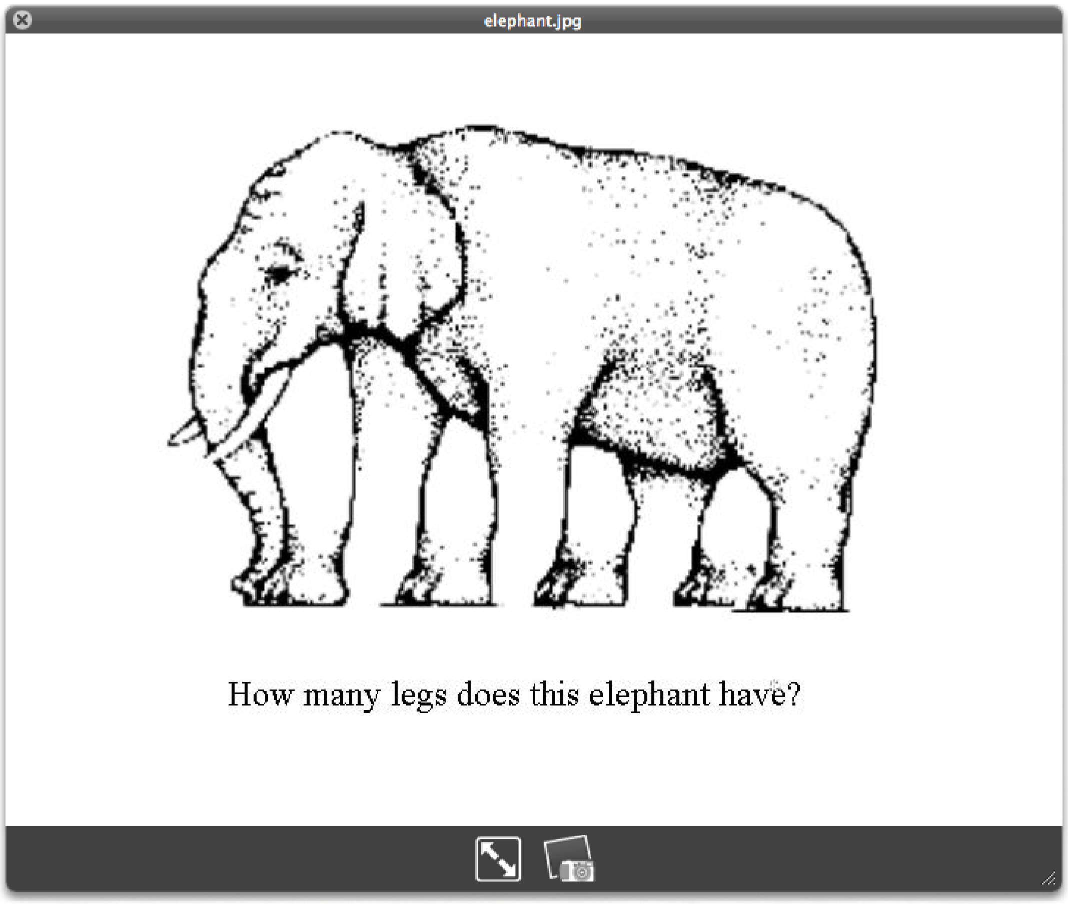


Name	Kind	Last Opened
01 Seven Nation Army.m4p	MPEG-4 Audio File (Protected)	Wednesday, July 4, 2007, 5:15
elephant.jpg	Document	Tuesday, April 10, 2007, 4:09
yaml.rb	Ruby Source File	Monday, December 26, 2005, 1
05 Carnival of the A… – The Elephant.m4a	MPEG-4 Audio File	Sunday, November 7, 2004, 9:(
AppleWorks 6 Installer Log	SimpleText Document	Thursday, May 10, 2001, 11:31
Visitation.doc	Microsoft Word document	Monday, January 29, 2001, 8:2
DevilsDictionary.rtf	Microsoft Word RTF document	Saturday, October 21, 2000, 1(
elephant	Document	Monday, February 7, 2000, 12:
CA_GZ02.TXT	Plain text	Thursday, November 5, 1998,
CA_GZ19.TXT	Plain text	Thursday, November 5, 1998,

The image in the middle of the Finder's Cover Flow view is a thumbnail generated by Quick Look. Thumbnails appear in various places in Mac OS X v10.5. In addition to Cover Flow, they appear in the Finder in icon mode and column mode, in the Dock as stacks of icons, and in the Quick Look index sheet when multiple previews are requested. Thumbnails also appear in the Quick Look panel when Quick Look can't generate a preview (but can generate a thumbnail).

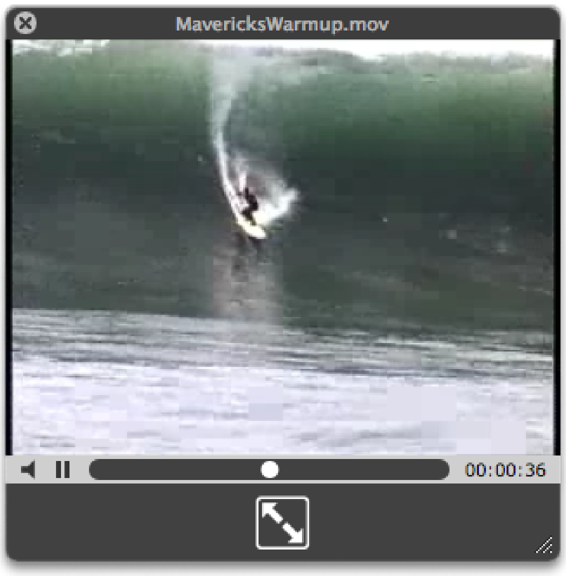
If the user want to get a closer look at a document, he or she could select it in Spotlight or a Finder window and press the space bar. Quick Look displays a full-size preview image of the document similar to the one in Figure 1-2. (In the Finder you can also press Command-Y to view a preview of the selected item.)

Figure 1-2 A Quick Look preview of an image



Quick Look preview include not only static images and documents, but can include QuickTime movies, as in the example in Figure 1-3.

Figure 1-3 A Quick Look preview of a movie



When users request previews for multiple documents in the Finder, the Quick Look panel enables them to cycle through the previews or look at them all at once (as thumbnails) in an index sheet. The Quick Look panel also includes controls that permit users to resize a preview dynamically, expand it to take up the screen, close the panel, and (depending on document type) perform operations such as "play movie" and "add to iPhoto."

Developing for Quick Look

Architecturally, Quick Look has a consumer, or client, side and a side that provides the thumbnail and preview images to the consumer side for display. (For a detailed look at the various parts of Quick Look and how they work together, see "Quick Look Architecture" (page 13).) Clients of Quick Look request thumbnails and previews for listed and selected documents, respectively, and receive images for display.

Quick Look supports the display of document thumbnails and previews if the format is one of its native types. The native Quick Look types are plain text, RTF, HTML, PDF, images (in various standard formats, such as JPG, PNG, and TIFF), and QuickTime movies and sounds. However, if a document is not in one of the native types, the document's application must include a Quick Look generator if it wants to take advantage of the Quick Look feature. The generator is a bundle that creates representations of the application's documents in one of the native types for display as previews and thumbnails. The purpose of a Quick Look generator is to provide upon demand, and as efficiently as possible, a thumbnail or preview image of a document in one of the native Quick Look types.

Of the various components of Quick Look, only the generator bundle (which is based on the `CFPlugIn` architecture) exposes a programmatic interface for third-party developers. Because of its `CFPlugIn` foundation, and to make Quick Look accessible to as many applications as possible, the generator API is in ANSI C, not Objective-C.

A Quick Look generator bundle must have an extension of `qlgenerator` and be installed in a file-system location described in "Installing Quick Look Generators" (page 16).

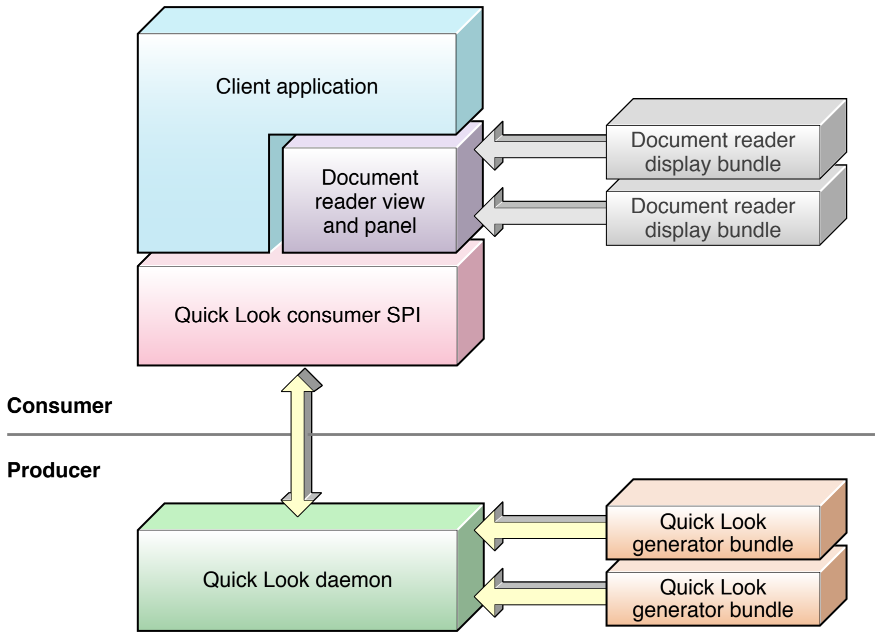
Quick Look Architecture

The follow sections examine the architecture of Quick Look. A general picture of this architecture helps you to understand the role and constraints of generators.

Quick Look Consumers and Producers

The architecture of Quick Look is based on a consumer—producer model. The consumer (or client) is an application that wants to display thumbnail and preview representations of documents. The producer side of the architecture provides those representations to the consumer. (Some Quick Look clients are system applications such as Finder, Spotlight, and FIleSync.) Clients have access to the public function `QLThumbnailImageCreate`, but most of the part of Quick Look that supports the consumer consists of private resources and programmatic interfaces (system programmatic interfaces, or SPI). Figure 2-1 illustrates the architecture of Quick Look.

Figure 2-1 The Quick Look architecture


Client application
Document reader view and panel
Document reader display bundle
Document reader display bundle
Quick Look consumer SPI
Consumer
Producer
Quick Look daemon
Quick Look generator bundle
Quick Look generator bundle

The consumer portion of Quick Look has three components: a document reader (consisting of a custom view and panel), display bundles for that reader, and an SPI to enable communication with the client. Each of these components has a specific role to play in support of the consumer:

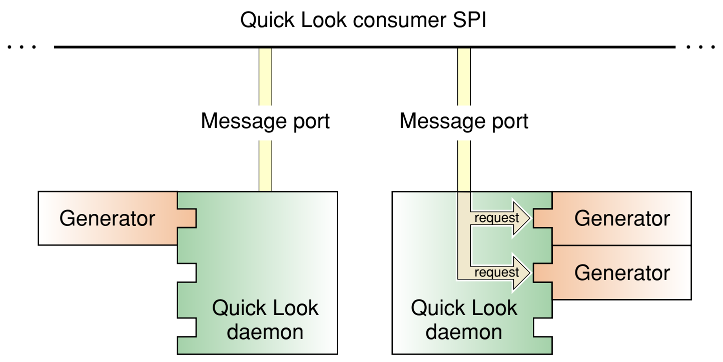
- Document reader—Quick Look implements a view (NSView) and panel (NSPanel) customized for displaying document previews. Along with the preview content, the view might include (at the client's option) controls for manipulating the preview, such as page-forward, page-backward, start playing, rewind, and text-search. A client application can embed this view in its user interface if it chooses. The Quick Look panel contains a Quick Look view and various controls that let the user take some action with the preview, such making the preview image full-screen or starting a slideshow.
- Display bundles—The Quick Look view itself doesn't display document previews but delegates that work to a display bundle. There is one Quick Look display bundle for each native document type. See "Developing for Quick Look" (page 12) for a discussion of the Quick native types. If a document is not of a native type, it must be converted to a one in order to be displayed as a Quick Look thumbnail or preview.
- Consumer SPI—The client application talks with Quick Look through this interface, making requests for previews and thumbnails and accessing the Quick Look document reader.

The "producer" part of Quick Look is based on a plug-in architecture that enables applications to provide thumbnails and previews of their documents, if those documents are not one of the native Quick Look types. The consumer and producer parts of Quick Look communicate over one or more Mach ports.

A Closer Look at Quick Look Daemons and Generators

The "producer" side of Quick Look is where third-party development takes place, and thus it merits a closer look. it consists of one or more Quick Look daemons and multiple Quick Look generators. Figure 2-2 shows how these things are related to one another.

Figure 2-2 Quick Look provider component



A Quick Look generator is CFPlugIn-based bundle that provides thumbnail images and previews for an application's documents. The job of a generator is to convert the document data into one of the Quick Look native types for each preview or thumbnail request it receives. The daemon loads a generator when it first receives a request for a document's preview or thumbnail. It locates the generator associated with a particular document using the document's content-type UTI, which is specified in the generator's information property list. It looks for the generator inside the application bundle or in one of the standard file-system locations for generators, such as `/Library/QuickLook`. The binary of a Quick Look generator must be universal and must be 32-bit only.

The Quick Look daemon (`quicklookd`) is a faceless background application that acts as a host for the CFPlugIn-based generators. It communicates with the consumer side of Quick Look through a Mach port, and (as noted above) locates and loads generators when it first receives a request for a preview or thumbnail for a document whose format is not one of the native types. It conveys requests from clients to generators and returns their responses.

There are advantages to having a daemon as an intermediary between the consumer SPI and the generators. If a Quick Look daemon crashes, it can be restarted immediately to resume service where it left off. If a generator is not thread-safe or needs to be isolated for any other reason, a separate daemon can be run to handle requests for that generator. When the daemon is idle, Quick Look can terminate it, thereby freeing up memory and providing a cheap form of garbage collection.

With all the architectural pieces in place, let's follow what happens when a client application such as Finder asks to display a preview of a document. The user opens a folder, displaying a list of documents of various types; some of these documents are of native Quick Look types and others are specific to certain applications. The user selects a document—say, a JPG file—and chooses the Quick Look Preview command from the File menu. In Quick Look the following sequence of actions occurs:

1. The client (Finder) sends a message to the consumer part of Quick Look requesting a preview for the document.
2. Quick Look sees that the document format is of a native type, so it loads the appropriate display bundle (if necessary)
3. The display bundle draws the document in the document reader (that is, in the Quick Look view, which is the content view of the Quick Look panel).

The user next requests a preview for a document that is not of a Quick Look native type. The following sequences of steps happens:

1. The client (Finder) sends a message to the consumer part of Quick Look requesting a preview for the document.
2. The Quick Look framework sees that the document format is *not* of a native type, so it forwards the message to the Quick Look daemon.
3. Using the document's content-type UTI, the daemon locates the appropriate generator and loads it if necessary.
4. It forwards the preview request to the generator, which creates a preview and either returns it or tells the generator where to find it.
5. The daemon returns the generator's response to the consumer part of Quick Look.
6. Quick Look loads the appropriate display bundle (if necessary).
7. The display bundle draws the document in the document reader.

Installing Quick Look Generators

You can store a Quick Look generator in an application bundle (in `MyApp.app/Contents/Library/QuickLook/`) or in one of the standard file-system locations:

`~/Library/QuickLook`—third party generators, accessible only to logged-in user

`/Library/QuickLook`—third party generators, accessible to all users of the system

`/System/Library/QuickLook`—Apple-provided generators, accessible to all users of the system

When Quick Look searches for a generator to use, it first looks for it in the bundle of the associated application and then in the standard file-system locations in the order given in the list above. If two generators have the same UTI, Quick Look uses the first one it finds in this search order. If two generators claim the same UTI at the same level (for example, in `/Library/QuickLook`), there is no way to determine which one of them will be chosen.

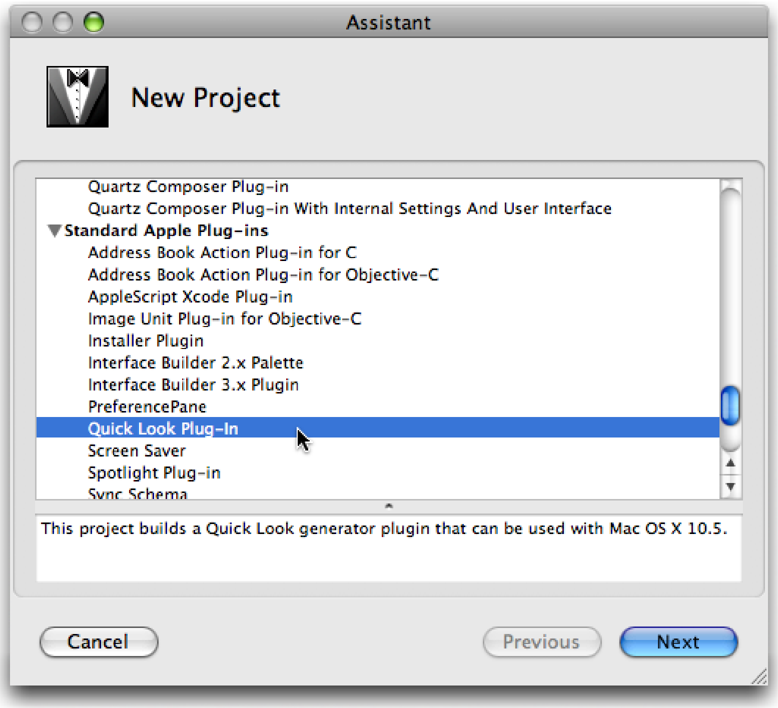
Creating and Configuring a Quick Look Project

Xcode projects for Quick Look generators originate from a special template that sets up important aspects of the project. However, you still must specify generator-specific configuration information and add any resources for the generator, typically before you write any code.

Creating and Setting Up the Project

To create a Quick Look generator project, start by choosing New Project from the File menu in the Xcode application. In the project-creation assistant, select Quick Look Plug-In in the list of project templates (as shown in Figure 3-1) and click Next.

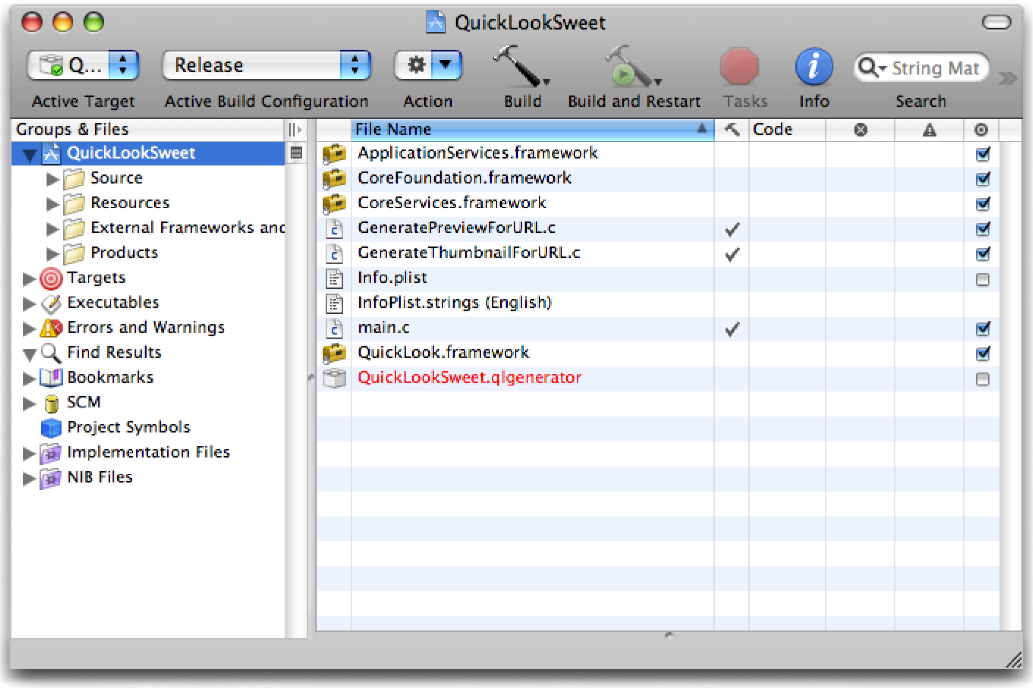
Figure 3-1 Choosing the Quick Look plug-in template


Assistant
New Project
Quartz Composer Plug-in
Quartz Composer Plug-in With Internal Settings And User Interface
Standard Apple Plug-ins
Address Book Action Plug-in for C
Address Book Action Plug-in for Objective-C
AppleScript Xcode Plug-in
Image Unit Plug-in for Objective-C
Installer Plugin
Interface Builder 2.x Palette
Interface Builder 3.x Plugin
PreferencePane
Quick Look Plug-In
Screen Saver
Spotlight Plug-in
Sync Schema
Cancel
Previous
Next

This project builds a Quick Look generator plugin that can be used with Mac OS X 10.5.

After you specify a name and location for the project, Xcode displays a project window similar to the example in Figure 3-2.

Figure 3-2 Default items in a Quick Look plug-in project



File Name		Code			
ApplicationServices.framework					☑
CoreFoundation.framework					☑
CoreServices.framework					☑
GeneratePreviewForURL.c	✓				☑
GenerateThumbnailForURL.c	✓				☑
Info.plist					☐
InfoPlist.strings (English)					
main.c	✓				☑
QuickLook.framework					☑
QuickLookSweet.qlgenerator					☐

The following items in this window have some special relevance to Quick Look:

- `QuickLook.framework`—The Quick Look framework, which includes both consumer and producer parts of the architecture.

  If you want additional frameworks, add them to the project and insert the appropriate `#include` or `#import` directives. For example, if you want to write code using Cocoa API, add `Cocoa.framework` to the project.

- `main.c` — This file contains all of the code required for a `CFPlugin`-based plug-in. You should not have to add or modify any of this code.

- `GeneratePreviewForURL.c` and `GenerateThumbnailForURL.c` — The first file contains code templates for the callbacks `GeneratePreviewForURL` and `CancelPreviewGeneration`; the second file contains code templates for the callbacks `GenerateThumbnailForURL` and `CancelThumbnailGeneration`.

  If your implementation code is going to be Objective-C, be sure to change the extensions of these files from `c` to `m` *in* Xcode (that is, by selecting the file and choosing Rename from the File menu).

Although a Quick Look generator does not (and should not) have nib files as resources, you can add other resources if necessary.

Project Configuration

The information property list (`Info.plist`) of a Quick Look generator project includes some special properties whose values you should set in addition to standard properties such as `CFBundleIdentifier` and `CFBundleVersion`. The following sections describe these properties.

The Content-Type UTI and CFPlugIn Properties

One important property for Quick Look generators is `LSItemContentTypes`, a subproperty of `CFBundleDocumentTypes`. Listing 3-1 shows the `CFBundleDocumentTypes` property when unedited. (Note that the Quick Look project template specifies the value (`QLGenerator`) of the `CFBundleTypeRole` property for you.)

Listing 3-1 The subproperties of `CFBundleDocumentTypes`

```
    <key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeRole</key>
            <string>QLGenerator</string>
            <key>LSItemContentTypes</key>
            <array>
                <string>SUPPORTED_UTI_TYPE</string> // change this!
            </array>
        </dict>
    </array>
```

Replace the string "`SUPPORTED_UTI_TYPE`" with one or more UTI s identifying the content types of the documents for which this generator generates thumbnails and previews. For example, the QuickLookSketch example project specifies the UTIs for Sketch documents:

```
<key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeRole</key>
            <string>QLGenerator</string>
            <key>LSItemContentTypes</key>
            <array>
                <string>com.apple.sketch2</string>
                <string>com.apple.sketch1</string>
            </array>
        </dict>
    </array>
```

For more information on Uniform Type Identifiers (UTIs) for document-content types, see *Uniform Type Identifiers Overview*.

As Listing 3-2 shows, a large segment of the `Info.plist` in a Quick Look generator project are properties related to `CFPlugIn`. You should not have to edit these properties.

Listing 3-2 CFPlugIn properties

```
    <key>CFPlugInDynamicRegisterFunction</key>
    <string></string>
    <key>CFPlugInDynamicRegistration</key>
    <string>NO</string>
    <key>CFPlugInFactories</key>
    <dict>
        <key>27EB40F9-21D6-4438-9395-692B52DB53FB</key>
        <string>QuickLookGeneratorPluginFactory</string>
    </dict>
    <key>CFPlugInTypes</key>
    <dict>
        <key>5E2D9680-5022-40FA-B806-43349622E5B9</key>
        <array>
            <string>27EB40F9-21D6-4438-9395-692B52DB53FB</string>
        </array>
    </dict>
    <key>CFPlugInUnloadFunction</key>
    <string></string>
```

Other Property List Keys

You can specify these additional key-value pairs in the information property list (`Info.plist`) of a Quick Look generator:

Key	Allowed value	Description
`QLThumbnailMinimum-Size`	Real number `<real>`*n*`</real>`	Specifies the minimum use size along one dimension (in points) of thumbnails for the generator. Quick Look does not call the `GenerateThumbnailForURL` callback function for thumbnail sizes less than this value. The default size is 17. If your generator is fast enough, you can remove this property so the thumbnail image can appear in standard lists.
`QLPreviewWidth`	Real number `<real>`*n*`</real>`	This number gives Quick Look a hint for the width (in points) of previews. It uses these values if the generator takes too long to produce the preview.
`QLPreviewHeight`	Real number `<real>`*n*`</real>`	This number gives Quick Look a hint for the height (in points) of previews. It uses these values if the generator takes too long to produce the preview.
`QLSupportsConcurrent-Requests`	`YES` or `NO`	Controls whether the generator can handle concurrent thumbnail and preview requests.
`QLNeedsToBeRun-InMainThread`	`YES` or `NO`	Controls whether the generator can be run in threads other than the main thread

The properties `QLSupportsConcurrentRequests` and `QLNeedsToBeRunInMainThread` are Quick Look properties that affect the multithreaded characteristics of the generator. They are discussed in "Generators and Thread Safety" (page 23).

Overview of Generator Implementation

The Quick Look generator API gives you several approaches for implementing generators. This chapter describes what they are and suggests the approach most suitable for applications based on their document types. It also discusses thread safety and multithreading issues related to Quick Look generators.

This chapter summarizes only the generation of thumbnails and previews. See "Canceling Previews and Thumbnails" (page 39) for a discussion of how to cancel the generation of thumbnails and previews.

The Quick Look Generator API

The header file `QLGenerator.h` in the Quick Look framework declares the programmatic interface for Quick Look generators. (Another header file, `QLBase.h`, is also in the `Headers` folder, but this file merely contains definitions of various macros used by both the Quick Look public and private interfaces.) The programmatic interface for generators is divided between thumbnail requests and preview requests, represented by opaque types `QLThumbnailRequestRef` and `QLPreviewRequestRef`, respectively. The API falls into three distinct categories:

- **Callbacks**

  Generators must implement a callback function typed as `GenerateThumbnailForURL` to create and return a thumbnail representation of a document. They must implement a callback function typed as `GeneratePreviewForURL` to create and return a preview of a document. As noted in "Creating and Configuring a Quick Look Project" (page 17), the Xcode template for generators makes the default names of the callback functions the same as their type names.

  An additional pair of callback functions can be implemented to cancel the generation of previews and thumbnails that a generator is currently performing. For more information on these callbacks, see "Canceling Previews and Thumbnails" (page 39).

- **Functions used in generating thumbnails and previews**

  Quick Look provides a range of functional alternatives for generators to create and return thumbnails and previews. For example, the `QLThumbnailRequestCreateContext` and `QLPreviewRequestCreateContext` functions provide a graphics context for drawing bitmap and vector-based images in. You use the `QLPreviewRequestSetDataRepresentation` function to return an embedded or dynamically generated preview, often for HTML content enriched with attachments. With the `QLThumbnailRequestSetImage` function you return a static thumbnail image representing a document.

  "Approaches to Thumbnail and Preview Generation" (page 22) describes these functions and related functions in greater detail and identifies the situations best suited to their use.

- **Functions that return information about the request or generator**

  The remaining functions in `QLGenerator.h` allow you get the attributes of preview or thumbnail requests or to obtain other data related to them. For example, the `QLThumbnailRequestCopyURL` function returns the URL identifying the document for which a thumbnail is requested. The

QLThumbnailRequestGetGeneratorBundle function returns a reference (CFBundleRef) to the generator's bundle. And the QLPreviewRequestCopyContentUTI function returns the UTI identifier of the current document's content (for example, com.apple.sketch1).

An important distinction to keep in mind when programming generators is the difference between options and properties. Both are names of CFDictionaryRef parameters in Quick Look functions. But the *options* parameter in the callback functions GenerateThumbnailForURL and GeneratePreviewForURL is a dictionary of options, or hints, from the client to the generator for how the request should be handled. The *properties* parameter is the last parameter in the QLThumbnailRequest and QLPreviewRequest functions used for creating thumbnails and previews,; the properties dictionary contains data supplemental to the created thumbnail or preview.

Approaches to Thumbnail and Preview Generation

The approach you take toward thumbnail and preview generation, and the Quick Look functions you use, depend on the kind of document your generator is intended for. Ask yourself these questions about the document:

- Is it bundled (as is, for example, a Pages document) or is it non-bundled (or flat)?
- Does it contain graphics or text? Or both graphics and text?
- If graphics, is it a bitmap or vector image?
- Does it have a single page or multiple pages?

Of course, whether the request is for a thumbnail or a preview enters into your choice of approach. If a request is for a thumbnail with a size no larger than a regular document icon, then a thumbnail at that size may be no better than the icon. If the request is for preview of a multipage document, do you show just the first page of the document or all of it? Whether the request is for a thumbnail or a preview, the performance of your generator is of paramount importance. For example, when a client requests thumbnails, it can request them for dozens of different documents; inefficient generators can make the client's display of thumbnails appear sluggish. If the client requests a preview for a document that is over 200 pages, perhaps you should include only enough of the document for the user to identify it. For your generator you should adopt proper memory-management practices and the appropriate multithreading strategy. For more information about multithreaded generators and thread-safety issues, see "Generators and Thread Safety" (page 23)

If you want to specify static thumbnail and preview images for a bundled document, you can take the easiest approach—it doesn't even require a generator. Just have your application place the images inside the document bundle in a subfolder named QuickLook; the image file for thumbnails should be named Thumbnail.*ext* and the file for previews should be named Preview.*ext* (where *ext* is an extension such as tiff, png, or jpg). If you decide on this approach, you should not create a generator.

Programmatically, you can take one the following approaches for generating your thumbnails and previews, depending on the document and other circumstances:

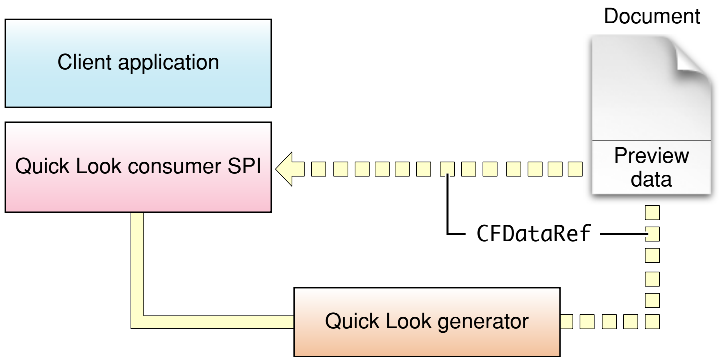
- If the document is single page containing bitmap graphics, vector graphics, or even text (generally when it is a graphical element of the preview), you can draw the thumbnail or preview in a graphics context returned by, respectively, the QLThumbnailRequestCreateContext or QLPreviewRequestCreateContext function.

- If the document has more than one page of vector graphics or text, you can draw the preview as PDF content in the graphics context supplied by `QLPreviewRequestCreatePDFContext`. You can call regular Core Graphics functions to draw the preview image.

  The advantage of this and the previous approach is that you completely control what's drawn; however, you have to handle the layout yourself. Applications that are good candidates for this approach are Font Book, Keynote, OmniGraffle, and Pages.

- For any kind of document, the application can write the thumbnail and preview image as part of the document data, which the generator retrieves and returns with the functions `QLThumbnailRequestSetImageWithData` and `QLPreviewRequestSetDataRepresentation`, respectively. Figure 4-1 illustrates this approach. For previews, you must specify which native Quick Look type the preview data is in through the *`contentTypeUTI`* parameter. For thumbnails, the returned data must in a format that can be processed by the Image I/O framework : JPG, TIFF, PNG, and so on.

Figure 4-1 Returning preview data stored in the document



- For multipage documents, typically textual documents, the generator can dynamically generate the preview "on the fly" and return it with the `QLPreviewRequestSetDataRepresentation` function.

  Although you can do this for a preview in any native Quick Look type (such as RTF), a recommended approach for documents with "enriched" textual content is to use `QLPreviewRequestSetDataRepresentation` with a *`contentTypeUTI`* parameter of `kUTTypeHTML`. This combination of function and parameter tells Quick Look to use the Web Kit to handle the layout of the preview. In the final parameter of the function, the `properties` dictionary, you can specify attachments in the HTML (such as images, sounds, and even things like Address Book cards). For this approach to be feasible, of course, the document data must be convertible to HTML. Some applications that are good candidates for this approach are OmniOutliner, Delicious Library, and Microsoft Messenger.

- When you cannot provide Quick Look (via `QLThumbnailRequestSetImageWithData`) a version of a thumbnail image that is in a format suitable for the Image I/O framework, but you can generate a serialized thumbnail image in some other format, you can use the `QLThumbnailRequestSetImage` function to return this image to Quick Look.

Generators and Thread Safety

For performance reasons, the Quick Look daemon (`quicklookd`) prefers to run a generator in its own thread, usually concurrently with other generators or even with the same generator when that generator is working on multiple documents. Given this, several thread-safety questions arise when you write code for a generator:

- Is the generator code itself thread-safe?
- Are the frameworks that the generator calls into thread-safe in the current context?

  For example, the Application Kit is generally thread-safe, but its text system (consisting of `NSText`, `NSTextView`, `NSLayoutManager`, `NSTextStorage`, and related classes) is not thread-safe if the layout of text is done asynchronously.
- Is the generator code or the framework code called by the generator able to be run in a non-main thread?

If you can determine the answer to these questions, you can configure your generator for optimum performance by setting the `QLSupportsConcurrentRequests` and `QLNeedsToBeRunOnMainThread` properties in you generator's information property list (`Info.plist`). (If you are unsure of the answer to any of the above questions, assume the most conservative answer in terms of thread safety.) Table 4-1 summarizes the thread-safety status that Quick Look assumes when you assign different values to these two properties.

Table 4-1 Quick Look properties for specifying the thread-safety status of the generator

Quick Look property pair	Values	Thread-safety status
`QLSupportsConcurrentRequests` `QLNeedsToBeRunOnMainThread`	`NO` `NO`	Default. The generator code is not thread safe but it uses thread-safe frameworks. The generator is never called twice at the same time, but might be called on different threads.
`QLSupportsConcurrentRequests` `QLNeedsToBeRunOnMainThread`	`YES` `NO`	The generator code is thread save and uses thread-safe frameworks. Quick Look can call the generator for several documents at the same time in different threads, including the main thread.
`QLSupportsConcurrentRequests` `QLNeedsToBeRunOnMainThread`	`NO` `YES`	The safest context, because Quick Look calls the generator serially in the main thread.
`QLSupportsConcurrentRequests` `QLNeedsToBeRunOnMainThread`	`YES` `YES`	In some situations, the Quick Look daemon may spin off a subprocess to handle requests from clients, so those requests might be dispatched to the same generator code in two different processes. This combination indicates that the generator is thread safe in that context.

For information about thread-safety issues, including the thread-safe status of the Carbon and Cocoa frameworks, see *Threading Programming Guide*.

Drawing Thumbnails and Previews In a Graphics Context

For previews or thumbnails of documents that consist primarily or solely of graphics and images, the best approach a generator can take is to draw the image of that document in a graphics context provided by Quick Look. The generator draws the document images directly in the client—the graphics context acts as a kind of window onto a surface of the client application. By doing this, you can avoid the overhead of creating and compressing an image into a native type and then requiring the client to decompress and load it on their end. Three graphics contexts are available, each for a different kind of document:

- A graphics context for drawing one or more bitmap images that fit on one page
- A graphics context for drawing one or more vector images that fit on one page
- A graphics context for drawing multiple pages of vector images

"Drawing Document Images in a Graphics Context" (page 25) describes how to draw a thumbnail or preview for the first two situations. "Drawing Previews in a PDF Context" (page 27) discusses the third kind of graphics context and explains how to use it.

Drawing Document Images in a Graphics Context

The strategy for drawing single-page previews and thumbnails in a graphics context is the same. Implement the appropriate callback function—`GenerateThumbnailForURL` or `GeneratePreviewForURL`—to read the given document (located by the `CFURLRef` parameter) into memory. Then get the Quick Look graphics context with a call to `QLThumbnailRequestCreateContext` or `QLPreviewRequestCreateContext` and draw the thumbnail or preview image in the provided context. Listing 5-1 shows the code for generating a preview of a Sketch document.

Listing 5-1 Drawing a Sketch preview in a Quick Look graphics context

```
OSStatus GeneratePreviewForURL(void *thisInterface, QLPreviewRequestRef preview,
 CFURLRef url, CFStringRef contentTypeUTI, CFDictionaryRef options)
{
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

    // Create and read the document file
    SKTDrawDocument* document = [[SKTDrawDocument alloc] init];

    if(![document readFromURL:(NSURL *)url ofType:(NSString *)contentTypeUTI])
 {
        [document release];
        [pool release];
        return noErr;
    }

    NSSize canvasSize = [document canvasSize];
    // Preview will be drawn in a vectorized context
    CGContextRef cgContext = QLPreviewRequestCreateContext(preview, *(CGSize
*)&canvasSize, false, NULL);
    if(cgContext) {
        NSGraphicsContext* context = [NSGraphicsContext
graphicsContextWithGraphicsPort:(void *)cgContext flipped:YES];
        if(context) {
            [document drawDocumentInContext:context];
        }
        QLPreviewRequestFlushContext(preview, cgContext);
        CFRelease(cgContext);
    }
    [pool release];
    return noErr;
}
```

Before you draw the preview or thumbnail in the provided graphics context, make sure you save the current context and then restore that context when you're finished drawing. You should then flush the context with `QLPreviewRequestFlushContext` or `QLThumbnailRequestFlushContext` and release it as shown in the above example.

The `QLPreviewRequestCreateContext` and `QLThumbnailRequestCreateContext` functions have identical sets of parameters. The first parameter identifies the preview request or thumbnail request object passed into the callback. The other parameters have a more direct bearing on the created graphics context.

- The second parameter (parameter named *`size`* in the function declaration) is the size of the image to be drawn in either pixels or points depending on whether the graphics context is bitmap or vector, respectively.
- The third parameter (*`isBitmap`*) tells Quick Look whether the returned graphics context should be suited for bitmap or vector graphics; in the example above, a vector-optimized graphics context is requested with a `false` value.
- The fourth and final parameter is a dictionary of properties that you can pass back to Quick Look as hints for handling the drawn image; see *QLPreviewRequest Reference* and *QLThumbnailRequest Reference* for details.

A generator implementing the `GenerateThumbnailForURL` callback might be passed in the *`options`* directory a floating-point value that specifies how much Quick Look is scaling the thumbnail image. (You can access this value using the `kQLThumbnailOptionScaleFactorKey` key.). If you a drawing a vector image for a thumbnail using the graphics context returned from `QLThumbnailRequestCreateContext`, you don't have to worry about scaling the image; just draw it normally in the the given size, which is in points. Quick Look creates a context with the specified size multiplied by the scale factor in pixels but also applies the appropriate affine transform so that the drawing context appears to the generator to be of the stated size.

Drawing Previews in a PDF Context

If your application has documents that (potentially) have more than one page of vector graphics, you should consider using the `QLPreviewRequestCreatePDFContext` function to create the graphics context for drawing the preview in. This function returns a graphics context suited for PDF content. The procedure is similar to the one described in "Drawing Document Images in a Graphics Context" (page 25). However, there are some important differences:

- Some parameters of the `QLPreviewRequestCreatePDFContext` are different from those of `QLPreviewRequestCreateContext`:
    - The second parameter (*`mediaBox`*) is a pointer to a rectangle that defines the location and size of the PDF page.
    - The third parameter (*`auxiliaryInfo`*) is a dictionary containing auxiliary PDF information.
- You must precede the drawing of each page by calling `CGPDFContextBeginPage` and call `CGPDFContextEndPage` when you have finished drawing a page.

As with `QLPreviewRequestCreateContext`, when you have finished drawing the preview, be sure to call `QLPreviewRequestFlushContext`.

Dynamically Generating Previews

If a textual document can readily be converted from its native format into an appropriate Quick Look format (HTML, RTF, PDF, and plain text), your generator can perform that conversion for previews of that document. In addition, if you can generate HTML data for your preview, you can also include attachments for such items as images, QuickTime movies, and Flash animations.

An important difference between HTML previews and other kinds of textual previews is that in the former case, the Web Kit handles the layout of the preview for you. For previews in other textual formats, your generator must handle the layout of the text.

"Creating Textual Representations On the Fly" discusses how you might dynamically create a preview for a textual document (in this case, RTF). "Generating Enriched HTML" (page 30) describes the HTML data-plus-attachments approach.

Creating Textual Representations "On the Fly"

The code example in Listing 6-1 (page 29) illustrates how a generator might create and return an RTF version of a document as a preview. Although most of the generator code is related to methods of a private framework, there are two important things to point out:

- The generator uses a private CSS parser object to assist in the layout of the preview.
- The native format of the document is XML, which the generator then converts (using private methods) to RTF.

The important aspect of this code from a Quick Look perspective is the call to `QLPreviewRequestSetDataRepresentation` after the RTF data has been created. As parameters to this function, the generator provides the RTF data and a UTI constant that indicates the native Quick Look type of the provided data.

Listing 6-1 Generating a preview in RTF format

```
OSStatus GeneratePreviewForURL(void *thisInterface, QLPreviewRequestRef preview,
 CFURLRef url, CFStringRef contentTypeUTI, CFDictionaryRef options)
{
    static CSSParser *theCSS = nil;
    NSError *theErr = nil;
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    // cache the CSS parser
    if(theCSS == nil) {
        NSString *cssPath = [[NSBundle
bundleWithIdentifier:@"com.apple.quicklooksweet"] pathForResource:@"editor"
ofType:@"css"];
        if(cssPath == nil) {
            return noErr;
        }
        theCSS = [[CSSParser parserFromPath:cssPath] retain];
    }
    if(theCSS == nil) {
        return noErr;
    }
    NSLog(@"GeneratePreviewForURL start...");
   NSXMLDocument *theDoc = [[[NSXMLDocument alloc] initWithContentsOfURL:(NSURL
 *)url options:0 error:&theErr] autorelease];
    if (!theDoc && theErr) {
        NSLog(@"Error creating the XML, %@", theErr);
        [pool release];
        return noErr;
    }
    XMLAttributedStringCreation *theXMLStr = [XMLAttributedStringCreation
XMLAttributedStringCreator];
    NSMutableAttributedString *theAttrStr = [theXMLStr
attributedStringForNode:[theDoc rootElement] CSSParser:theCSS];
    if (!theAttrStr) {
        [pool release];
        return noErr;
    }
    NSData *theRTF = [theAttrStr RTFFromRange:NSMakeRange(0, [theAttrStr
length]-1) documentAttributes:nil];
   QLPreviewRequestSetDataRepresentation(preview, (CFDataRef)theRTF, kUTTypeRTF,
 NULL);

    [pool release];
    return noErr;
}
```

Generating Enriched HTML

A generally useful but slightly more complex approach to generating a preview dynamically is to create HTML to which you attach other data, such as images, Java applets, and Flash animations. This approach can be ideally suited for applications that aren't primarily textual or graphical in nature, such as applications whose document user interface is a combination of text and graphics, or applications that display their document data in a user interface consisting of table views, text and form fields, labeled checkboxes, and so on.

An example of the latter sort of application is the Core Data example application, Event Manager. The Event Manager application allows users to enter information on social and work events, including the occasion, the description of the event, and the start and end dates. It uses Core Data to store and manage the entered information. The implementation of `GeneratePreviewForURL` shown in Listing 6-2gets the managed object representing the document file, creates a static HTML file using an `NSMutableString` object, and inserts in the appropriate places document data fetched from the managed object. It also creates the properties dictionary to be passed back to Quick Look in the call to `QLPreviewRequestSetDataRepresentation`; the properties in this dictionary define the HTML data and the attachments associated with that data.

Listing 6-2 Generating a preview composed of HTML data plus an image attachment

```
OSStatus GeneratePreviewForURL(void *thisInterface, QLPreviewRequestRef preview,
 CFURLRef url, CFStringRef contentTypeUTI, CFDictionaryRef options)
{
    NSAutoreleasePool *pool;
    NSMutableDictionary *props,*imgProps;
    NSManagedObject *occasion=NULL;
    NSMutableString *html;
    NSString *momPath;
    NSData *image;

    pool = [[NSAutoreleasePool alloc] init];
    // Initializes the Core Data stack to read from the file and returns a
managed object
    // See WebViewQLPlugin in /Developer/Examples/QuickLook for full code
    occasion=InitializeCoreDataStackWithURL(url);
    // Before proceeding make sure the user didn't cancel the request
    if (QLPreviewRequestIsCancelled(preview))
        return noErr;
    if (occasion!=NULL)
    {
        props=[[[NSMutableDictionary alloc] init] autorelease];
        [props setObject:@"UTF-8" forKey:(NSString
*)kQLPreviewPropertyTextEncodingNameKey];
        [props setObject:@"text/html" forKey:(NSString
*)kQLPreviewPropertyMIMETypeKey];

        html=[[[NSMutableString alloc] init] autorelease];
        [html appendString:@"<html><body bgcolor=white>"];
        [html appendString:@"<img src=\"cid:tabs.png\"><br>"];
        [html appendString:@"<h1>Occasion:"];
        [html appendString:[occasion valueForKey:@"name"]];
        [html appendString:@"</h1><br><br><h2>Description:</h2><br>"];
        [html appendString:[occasion valueForKey:@"detailDescription"]];
        [html appendString:@"<br><h2>Start Date:</h2><br>"];
        [html appendString:[[occasion valueForKey:@"startDate"] description]];
        [html appendString:@"<br><h2>End Date:</h2><br>"];
        [html appendString:[[occasion valueForKey:@"endDate"] description]];
        [html appendString:@"</body></html>"];

        image=[NSData dataWithContentsOfFile:[NSString
stringWithFormat:@"%@%@",[[NSBundle
bundleWithIdentifier:@"com.apple.eventsmanager.qlgenerator"] bundlePath],
@"/Contents/Resources/tabs.png"]];
        imgProps=[[[NSMutableDictionary alloc] init] autorelease];
        [imgProps setObject:@"image/png" forKey:(NSString
*)kQLPreviewPropertyMIMETypeKey];
        [imgProps setObject:image forKey:(NSString
*)kQLPreviewPropertyAttachmentDataKey];
        [props setObject:[NSDictionary dictionaryWithObject:imgProps
forKey:@"tabs.png"] forKey:(NSString *)kQLPreviewPropertyAttachmentsKey];

        QLPreviewRequestSetDataRepresentation(preview,(CFDataRef)[html
dataUsingEncoding:NSUTF8StringEncoding],kUTTypeHTML,(CFDictionaryRef)props);
    }
    else {
        NSLog(@"Couldn't get managed object!");
    }
    [pool release];
    return noErr;
}
```

Note: In the interests of brevity, the `InitializeCoreDataStackWithURL` function in the above listing is a placeholder for the real code in the example project that initializes the Core Data stack from the passed-in file and returns a managed object.

There are a few things worthy of special notice in Listing 6-2:

- The HTML references the image attachment using the URL scheme `cid:`*identifier*. The identifier is always used as the key for a dictionary containing attachment data (*`imgProps`*) that is added to the properties dictionary.
- The properties dictionary (`props`) contains the HTML encoding and HTML MIME type (`kQLPreviewPropertyTextEncodingNameKey` and `kQLPreviewPropertyMIMETypeKey`) and any attachment subdictionaries.
- In this case there is one attachment subdictionary; it contains the MIME type of the image attachment and the image data (accessed with `kQLPreviewPropertyMIMETypeKey` and `kQLPreviewPropertyAttachmentDataKey`, respectively)

When the generator calls `QLPreviewRequestSetDataRepresentation` it passes in the HTML data (in the specified encoding), the properties dictionary, and the UTI constant identifying HTML content. With the HTML and the properties dictionary set up in this way, the Web Kit can load the HTML and, when it parses it, load the attachments into the web view.

Although the code listing uses an `img` HTML element for the `cid`-scheme reference to the image attachment, you can also use the `object` element for all kinds of attachments (images, audio, videos, Java applets, and Flash animations). It is not recommended that you use Web Kit plug-ins in enriched HTML passed back to Quick Look.

Saving Previews and Thumbnails in the Document

As one approach for providing thumbnail and preview data to Quick Look, the application can store that data as part of the document data. The generator can then access it and return it to Quick Look in a call to `QLThumbnailRequestSetImageWithData` or `QLPreviewRequestSetDataRepresentation`. This approach permits a quick response time for the generator, but at the expense of a larger document file.

To illustrate how your generator might provide previews and thumbnails using this approach, the following listings show modifications to the code for the Sketch application that writes a thumbnail image as part of the document data. Listing 7-1 shows how you might define a property of the `NSDocument` subclass to hold the image data.

Listing 7-1 Sketch example project: adding a thumbnail property

```
@interface SKTDrawDocument : NSDocument {
    @private
    NSMutableArray *_graphics;
    // ...other instance variables here...
    NSData *_thumbnail;
}
// ...existing methods here...
- (NSData *)thumbnail;
```

Implement the `thumbnail` accessor method to return the thumbnail image. To the the `NSDocument` method that prepares the document data for writing out to a file (`dataOfType:error:`) are added the lines of code in Listing 7-2 indicated by the “new” labels.

Listing 7-2 Sketch example project: including the thumbnail with the document data

```
static NSString *SKTThumbnailImageKey = @"SketchThumbnail";
// new

- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError {
    NSData *data,;
    NSArray *graphics = [self graphics];
    NSPrintInfo *printInfo = [self printInfo];
    NSWorkspace *workspace = [NSWorkspace sharedWorkspace];
    BOOL useTypeConformance = [workspace
respondsToSelector:@selector(type:conformsToType:)];
    if ((useTypeConformance && [workspace type:SKTDrawDocumentNewTypeName
conformsToType:typeName]) || [typeName
isEqualToString:SKTDrawDocumentOldTypeName]) {
        NSData *tiffRep;                                              //
 new
        NSMutableDictionary *properties = [NSMutableDictionary dictionary];
        [properties setObject:[NSNumber
numberWithInt:SKTDrawDocumentCurrentVersion] forKey:SKTDrawDocumentVersionKey];
        [properties setObject:[SKTGraphic propertiesWithGraphics:graphics]
forKey:SKTDrawDocumentGraphicsKey];
       [properties setObject:[NSArchiver archivedDataWithRootObject:printInfo]
 forKey:SKTDrawDocumentPrintInfoKey];
```

```
        tiffRep = [self TIFFDataWithGraphics:graphics error:outError];
// new
        [properties setObject:tiffRep forKey:SKTThumbnailImageKey];
// new
        data = [NSPropertyListSerialization dataFromPropertyList:properties
format:NSPropertyListBinaryFormat_v1_0 errorDescription:NULL];
    } else if ((useTypeConformance && [workspace type:(NSString *)kUTTypePDF
conformsToType:typeName]) || [typeName isEqualToString:NSPDFPboardType]) {
    data = [SKTRenderingView pdfDataWithGraphics:graphics];
    } else {
    NSParameterAssert((useTypeConformance && [workspace type:(NSString
*)kUTTypeTIFF conformsToType:typeName]) || [typeName
isEqualToString:NSTIFFPboardType]);
        data = [SKTRenderingView tiffDataWithGraphics:graphics error:outError];
    }
    return data;
}
```

In the corresponding `NSDocument` method for reading document data back in (`readFromData:ofType:error:`) "unpack" the thumbnail from the dictionary of document properties:

```
_thumbnail = [[properties objectForKey:SKTThumbnailImageKey] retain];
```

Now implementing the generator for Sketch is a simple matter of accessing the thumbnail image data and passing it to Quick Look in a call to `QLThumbnailRequestSetImageWithData`, as shown in Listing 7-3. (For previews, the corresponding function is `QLPreviewRequestSetDataRepresentation`.)

Listing 7-3 Returning the stored thumbnail image to Quick Look

```
OSStatus GenerateThumbnailForURL(void *thisInterface, QLThumbnailRequestRef
thumbnail, CFURLRef url, CFStringRef contentTypeUTI, CFDictionaryRef options,
CGSize maxSize)
{
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
    SKTDrawDocument* document = [[SKTDrawDocument alloc] init];
    if(![document readFromURL:(NSURL *)url ofType:(NSString *)contentTypeUTI])
 {
        [document release];
        [pool release];
        return noErr;
    }
    if ([document respondsToSelector:@selector(thumbnail)]) {  // runtime
verification
        NSData *tiffData = [document thumbnail];
        if (tiffData != nil) {
            NSDictionary *props = [NSDictionary
dictionaryWithObject:@"public.tiff" forKey:(NSString
*)kCGImageSourceTypeIdentifierHint];
            QLThumbnailRequestSetImageWithData(thumbnail, (CFDataRef)tiffData,
 (CFDictionaryRef)props);
            return noErr;
        }
    }
    NSSize canvasSize = [document canvasSize];
   CGContextRef cgContext = QLThumbnailRequestCreateContext(thumbnail, *(CGSize
 *)&canvasSize, false, NULL);
    if(cgContext) {
        NSGraphicsContext* context = [NSGraphicsContext
graphicsContextWithGraphicsPort:(void *)cgContext flipped:YES];
        if(context) {
            [document drawDocumentInContext:context];
        }
        QLThumbnailRequestFlushContext(thumbnail, cgContext);
        CFRelease(cgContext);
    }
    [pool release];
    return noErr;
}
```

In the call to `QLThumbnailRequestSetImageWithData`, the generator indicates the image format to Quick Look with the `kCGImageSourceTypeIdentifierHint` property. Note that this example checks whether the class of the document object implements the `thumbnail` accessor method (to exclude prior versions of the application) and, if so, it checks whether thumbnail data is returned. If it isn't, it draws the thumbnail image in a Quick Look–provided graphics context.

Assigning Core Graphics Images to Thumbnails

In some cases the simplest course for generating a thumbnail is to create a Core Graphics image rather than creating an image compatible with the I/O framework or drawing the image in a graphics context. For example, your generator might be using a framework that can directly provide a serialized version of the thumbnail image as a `CGImage` object. For this cases, you can create the Core Graphics image and then communicate that image to Quick Look by calling the `QLThumbnailRequestSetImage` function. The main difference between this function and `QLThumbnailRequestSetImageWithData` is that the latter function requires the image data to be in a format that is supported by the I/O framework.

Listing 8-1 illustrates approach, using methods of the QT Kit framework to get the poster frame of a movie as a Core Graphics image and setting that as the thumbnail for a movie file.

Listing 8-1 Creating and assigning a Core Graphics image

```
OSStatus GenerateThumbnailForURL(void *thisInterface, QLThumbnailRequestRef
thumbnail, CFURLRef url, CFStringRef contentTypeUTI, CFDictionaryRef options,
CGSize maxSize)
{
    NSError *theErr;
    QTMovie *theMovie = [QTMovie movieWithURL:(NSURL *)url error:&theErr];
    if (theMovie == nil) {
        if (theErr != nil) {
            NSLog(@"Couldn't load movie URL, error = %@", theErr);
        }
        return noErr;
    }
    [theMovie gotoPosterTime];
    QTTime mTime = [theMovie currentTime];
    NSDictionary *imgProp = [NSDictionary
dictionaryWithObject:QTMovieFrameImageTypeCGImageRef
forKey:QTMovieFrameImageType];
    CGImageRef theImage = (CGImageRef)[theMovie frameImageAtTime:mTime
withAttributes:imgProp error:&theErr];

    if (theImage == nil) {
        if (theErr != nil) {
            NSLog(@"Couldn't create CGImageRef, error = %@", theErr);
        }
        return noErr;
    }
    QLThumbnailRequestSetImage(thumbnail, theImage, NULL);
    return noErr;
}
```

Canceling Previews and Thumbnails

A client application or the Quick Look daemon (`quicklookd`) may decide that it no longer needs a preview or thumbnail image that it has requested from a generator. Often this occurs because a user has indicated (for example, by closing a Finder window) that he or she is no interested in the listed documents. When the client or `quicklookd` decides it no longer needs a thumbnail or preview, Quick Look informs the appropriate generator in two ways, described in the following sections. The generator should look for this cancellation, stop any image generation in progress, and clean up any resources used in generating the preview or thumbnail.

Canceling Through a Callback Function

When a client application no longer needs a thumbnail or preview that it has requested, it tells Quick Look, which then invokes one of two callback functions, depending on the type of item requested earlier:

`CancelThumbnailGeneration` for canceling the generation of thumbnails
`CancelPreviewGeneration` for canceling the generation of previews

The generator can implement these these functions to stop creating the previews and thumbnail images and clean up any resources so far used in their creation and return as quickly as possible.

However, it is not generally recommended that your code cancel the generation of previews and thumbnails by implementing one of these callback functions. Because Quick Look always calls these functions in a secondary thread, implementing it safely can be difficult. For example, you must be careful to match the cancellation request to the thread involved in the image generation. If you have any doubts about ensuring the thread-safety of your code, following the guidelines described in “Canceling Through Polling.”

Canceling Through Polling

When a client application no longer needs a thumbnail or preview that it has requested, it tells Quick Look, which then sets a Boolean flag for the request (in addition to invoking the `CancelThumbnailGeneration` or `CancelPreviewGeneration` callback function). A generator can access the value of this flag at any time by calling the `QLThumbnailRequestIsCancelled` function (for thumbnails) or `QLPreviewRequestIsCancelled` function (for previews).

In your generator code you can periodically call these functions to poll Quick Look for the cancellation status of the current request. If a call returns a true value, clean up any resources used so far in the generation of the thumbnail or preview and return `noErr`. For most generators, this approach is recommended over the approach described in “Canceling Through a Callback Function” (page 39).

You should call `QLThumbnailRequestIsCancelled` and `QLPreviewRequestIsCancelled` at appropriate places in your generator code. Which places are appropriate depends on what the code is doing and how well-factored it is. Generally, you should test for request cancellation before doing some task that is time-consuming, especially when you won't be able to query for cancellation status while that task is proceeding (for example, parsing a file).

An example is helpful here. The logic of a typical generator has the following structure in its `GeneratePreviewForURL` callback function:

1. Load document data.
2. Parse document data.
3. Composite the preview or convert it to a native Quick Look type.
4. Flush the graphics context or set the data in the response.

Given this structure, you probably should call `QLPreviewRequestIsCancelled` between steps 1 and 2 and again between steps 2 and 3. You don't need to call the function between steps 3 and 4 because Quick Look will discard the preview when you complete step 4, after which you release your resources anyway.) The important idea is to poll for cancellation wisely; you shouldn't poll too often, but at the same time you should poll often enough so that a cancelled preview or thumbnail doesn't affect performance.

Debugging and Testing a Generator

Quick Look gives developers some facilities for debugging and testing their generator code. The following sections describe those facilities and offers some strategies and advice for debugging and testing generators.

Debugging Facilities

Because a generator is a plug-in and is not a self-contained executable, debugging it could be problematic if you were left on your own. Fortunately, Quick Look gives you a way to debug generator code easily: the `qlmanage` diagnostic tool (installed in `/usr/bin`). `qlmanage` executes your project's generator in almost the same kind of environment as the Quick Look daemon (`quicklookd`) does. You can run this tool as your project's executable and, by specifying certain arguments, you can step through your generator code and see how it handles previews and thumbnails.

To set up your Quick Look project for debugging, complete the following steps:

1. Choose New Custom Executable from the Project menu.

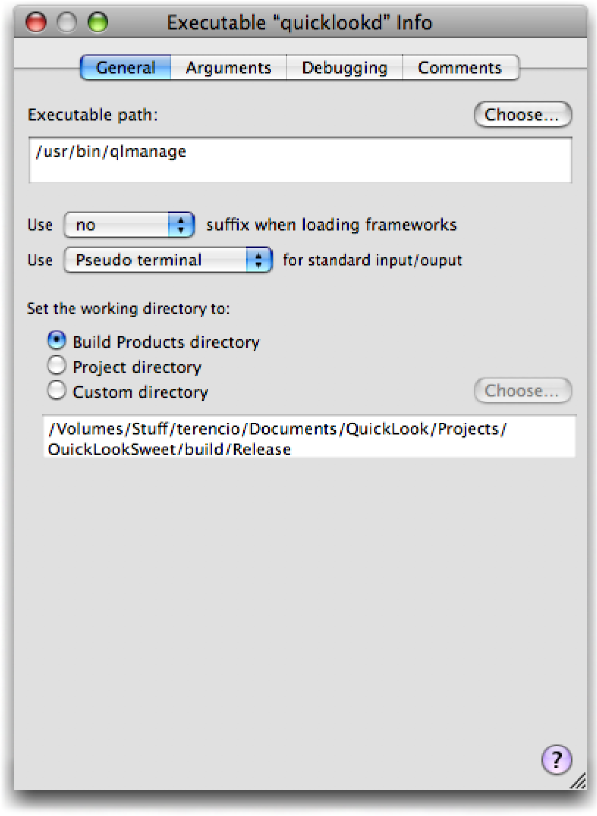
2. In the Assistant window, enter "qlmanage" as the executable name. In the Executable Path field specify the full path to the tool:

   ```
   /usr/bin/qlmanage
   ```

   Click Finish to dismiss the Assistant.

3. The Executable Info window appears for `qlmanage`, as shown in Figure 10-1. Click the Arguments tab.

Figure 10-1 Setting `qlmanage` as a custom executable



Executable path: /usr/bin/qlmanage

Use no suffix when loading frameworks

Use Pseudo terminal for standard input/ouput

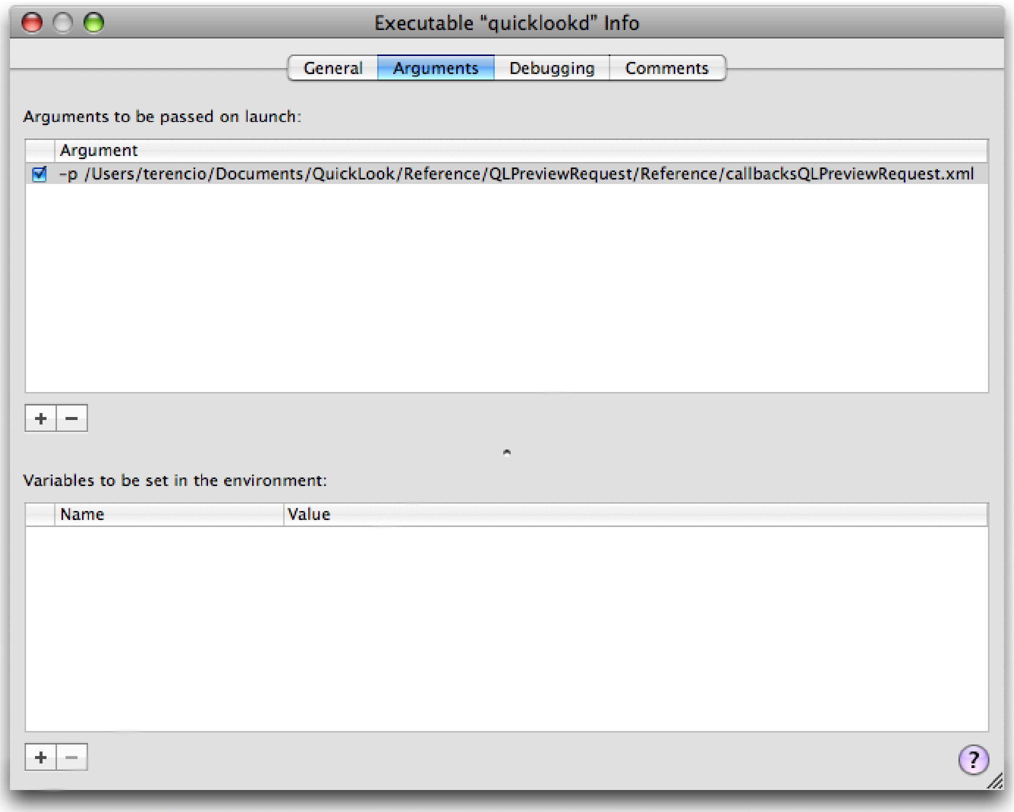
Set the working directory to:

- [✓ Build Products directory]
- Project directory
- Custom directory

/Volumes/Stuff/terencio/Documents/QuickLook/Projects/QuickLookSweet/build/Release

4. In the Arguments pane of the Executable Info window (Figure 10-2) enter one or more debugging options in the Arguments table.

Figure 10-2 Specifying a document for which `qlmanage` requests a preview



Arguments to be passed on launch:

	Argument
✓	-p /Users/terencio/Documents/QuickLook/Reference/QLPreviewRequest/Reference/callbacksQLPreviewRequest.xml

Variables to be set in the environment:

Name	Value

The `qlmanage` tool takes the following arguments:

Flag	Value	Description
`-p`	Absolute path to document	Requests preview of specified document
`-t`	Absolute path to document	Requests thumbnail of specified document.
`-r`	None	Resets `quicklookd` and the Quick Look client's generator cache
`-m`	None	Prints information on `quicklookd` actions, including a list of detected generators
`-h`	None	Prints a brief description of options

You can also run the `qlmanage` tool from the command line. The following example requests a thumbnail of a specified document:

```
qlmanage -t /tmp/MySketchDoc.sketch2
```

This example displays a preview for a particular document:

```
qlmanage -p /tmp/MySketchDoc.sketch2
```

The `-m` option for `qlmanage` is useful, as it prints (to standard output) a report from the Quick Look daemon on current generator status.

Listing 10-1 Sample output of `qlmanage -m`

```
2007-04-05 17:00:46.998 qlmanage[1190:d03] Server statistics:
server: living for 21s (9 requests handled)
memory used: 10 MB (10551296 bytes)
last burst: during 0s - 1 requests - 0s idle
plugins:
  com.apple.ichat.ichat -> /System/Library/QuickLook/iChat.qlgenerator
  com.apple.safari.bookmark ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Bookmark.qlgenerator
  com.apple.sketch1 -> /Library/QuickLook/QuickLookSketch.qlgenerator
  public.rtf ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Text.qlgenerator
  public.audio ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Audio.qlgenerator
  com.apple.dashboard-widget ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/StandardBundles.qlgenerator
  com.apple.rtfd ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Text.qlgenerator
  com.microsoft.word.doc -> /System/Library/QuickLook/Office.qlgenerator
  com.apple.addressbook.person ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Contact.qlgenerator
  public.plain-text ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Text.qlgenerator
  com.apple.quartz-composer-composition ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Movie.qlgenerator
  public.xml -> /Library/QuickLook/QuickLookSweet.qlgenerator
  com.apple.eventmanager.events -> /Library/QuickLook/WebViewQLPlugin.qlgenerator
  com.apple.sketch2 -> /Library/QuickLook/QuickLookSketch.qlgenerator
  com.apple.package ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Package.qlgenerator
  com.apple.ical.bookmark ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/iCal.qlgenerator
  com.adobe.pdf ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/PDF.qlgenerator
  public.font ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Font.qlgenerator
  com.apple.mail.emlx ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Mail.qlgenerator
  com.microsoft.excel.xls -> /System/Library/QuickLook/Office.qlgenerator
  com.apple.eventmanager.eventsbin -> /Library/QuickLook/WebViewQLPlugin.qlgenerator
  com.apple.mail.email ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Mail.qlgenerator
  com.apple.ical.ics ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/iCal.qlgenerator
  com.apple.systempreference.prefpane ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/StandardBundles.qlgenerator
  com.apple.safari.history ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Bookmark.qlgenerator
  public.html ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Web.qlgenerator
```

```
  com.apple.eventmanager.eventsq -> /Library/QuickLook/WebViewQLPlugin.qlgenerator
  com.apple.addressbook.group ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Contact.qlgenerator
  public.movie ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Movie.qlgenerator
  com.apple.application ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/StandardBundles.qlgenerator
  com.apple.ichat.transcript -> /System/Library/QuickLook/iChat.qlgenerator
  com.apple.ical.bookmark.todo ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/iCal.qlgenerator
  public.vcard ->
/System/Library/Frameworks/QuickLook.framework/Resources/Generators/Contact.qlgenerator
generators change detected: NO
```

Note: Consult the `qlmanage` man page for the syntax and complete list of options for this tool.

Once you have set up your Quick Look generator project for debugging, specify breakpoints in your code, change the build configuration to Debug, and choose Build and Debug from the Debug menu.

Testing Tools and Strategies

After your generator seems to be bug-free, you can test it further to determine if anything else needs to be improved. Copy the generator to an application bundle or to one of the standard file-system locations for Quick Look generators. Try out your generator with different client applications (Finder, Spotlight, Time Machine, and so forth). Using `qlmanage` as an executable (see "Debugging Facilities" (page 41)) you can test your generator to see how it handles thumbnails and previews. Force preview- or thumbnail-generation by closing a Finder or Spotlight window and see how well your generator responds.

In addition, check your generator to see how well it performs; if it takes longer than two seconds to generate a preview, then you should closely examine your code to find out where you could improve performance.

As an aid to testing, or even debugging, you can set the `QLEnableLogging` user default at the command line:

```
defaults write -g QLEnableLogging YES
```

After doing this, Quick Look prints log messages showing its activity, such as which generators it loads and which documents it requests previews and thumbnails for. Here is a sample log message:

```
2006-12-15 11:18:16.839 quicklookd[26260:3b03] [QL] Thumbnailing
/Users/jalon/Documents/PreviewableDocuments/Test5.sketch2. Content type UTI:
com.apple.sketch2. Generator used: <QLGenerator
/Library/QuickLook/quicklooksketch.qlgenerator>
```

Document Revision History

This table describes the changes to *Quick Look Programming Guide*.

Date	Notes
2008-02-08	Corrected minor errors.
2007-10-31	New document that describes the purpose and architecture of Quick Look generators and explains how to create them.