Bundle Programming Guide

Core Foundation > Resource Management

2005-11-09

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

Contents

Listings

Introduction to Bundle Programming Guide

A **bundle** is a directory in the file system that groups related resources together in one place. Applications, frameworks, and plug-ins are all examples of bundles. Programs can also use document bundles instead of flat files to save complex content.

Many bundles are also **packages**, that is, they are presented to the user as opaque files rather than as directories. This opaqueness has advantages for both users and developers. For users, it simplifies the user's interaction with applications and other bundles and makes it harder to delete critical resources accidentally. For developers, it simplifies the software distribution process.

Both Cocoa and Core Foundation provide API for accessing the contents of a bundle. For more information on accessing bundles from Cocoa, see the `NSBundle` class documentation. For information on accessing bundles from Core Foundation, see the `CFBundle` reference.

For information about framework bundles, see *Framework Programming Guide*.

Organization of This Document

This document includes the following articles:

- "About Bundles" (page 9) provides background information about Mac OS X bundles and why you should use them.
- "Anatomy of a Modern Bundle" (page 11) gives you a tour of a modern bundle and provides examples of how different parts of a bundle contribute to the overall directory hierarchy.
- "Bundles and the Resource Manager" (page 19) explains the bundle support for Resource Manager-style resources and includes an example of how to load these resources from your application.
- "Searching for Bundle Resources" (page 21) explains the search algorithm used by `NSBundle` and `CFBundle` to locate resources, which is helpful for developers who need to place those resources initially.
- "Packages and the Finder" (page 23) describes the behavior of the Finder as it pertains to bundles and packages.
- "Document Packages" (page 25) provides guidelines for creating your custom document types using a package instead of a file.
- "Guidelines for Using Bundles" (page 27) provides guidelines for when and how to use bundles.
- "Creating Bundles" (page 29) provides basic instructions on how to create a new bundle or convert an older NeXT bundle to the modern bundle structure.
- "Locating and Opening Bundles" (page 31) demonstrates how to find bundles on the system.
- "Locating Resources Inside a Bundle" (page 35) explains the interfaces used to find resources inside a bundle.
- "Loading and Unloading Executable Code" (page 37) demonstrates how to call functions and access classes defined in plug-ins and other loadable bundles.

- "Getting Information From a Bundle" (page 41) demonstrates how to retrieve information about the bundle itself, including its configuration data and directory paths.
- "Working With Localized Strings" (page 43) discusses the interfaces used to retrieve localized strings from a bundle and shows you how to write code in a way that makes it easier to find localizable strings.

About Bundles

The problem of managing code and related resources is not a new one and different operating systems have taken different approaches. In the past, resources have been compiled into the executable file, compiled into resource forks, and installed in known locations, among other techniques. The problem with many of these techniques is that they tend to be more fragile and in some cases complicate the process of updating the code and resources.

Bundles provide an elegant solution to the problem of grouping related code and resources together. A bundle is a hierarchical directory structure containing executable code and resources geared for a specific purpose. Applications, frameworks, and plug-ins can all be implemented as bundles in Mac OS X, and in fact, bundles are the preferred delivery mechanism for all of these software types. Developers can also use bundles for other purposes, including the implementation of custom document types.

Important: It is important to remember the distinction between what is a bundle and what is a package. The term bundle indicates a directory with a specific hierarchical structure, whereas the term package indicates a directory that is treated as an opaque entity by the Finder. Most bundles (including applications and plug-ins) are also packages. Some bundles, such as frameworks, are not packages, however.

Advantages of Using Bundles

Bundles provide significant advantages over other application packaging schemes available on Mac OS X.

- Bundles are directory hierarchies in the file system. A bundle contains real files that can be manipulated by all file-based services and API.
- The bundle directory structure makes it easy to support multiple localizations. You can easily add new localized resources or remove unwanted ones.
- Bundles can reside on volumes of many different formats, including multiple fork formats like HFS, HFS+, and AFP, and single-fork formats like UFS, SMB, and NFS.
- Users can install, relocate, and remove bundles simply by dragging them around in the Finder.
- Bundles that are also packages, and are therefore treated as opaque files, are less susceptible to accidental user modifications, such as removal, modification, or renaming of critical resources.
- A bundle can include separate executables for different target platforms. For example, a bundled application could include separate executables for Mac OS 9 and Mac OS X.
- A bundle can support multiple chip architectures (PowerPC, Intel), library architectures (CFM/MachO), and other special executables (for example, optimized libraries for AltiVec).
- Most executable code can be bundled. Applications, frameworks (shared libraries), and plug-ins all support the bundle model.
- An application can run directly from a server. No special shared libraries, extensions, and resources need to be installed on the local system.

Types of Bundles

The type of a bundle determines its internal directory structure and the location of key resources within that structure. Mac OS X supports two basic bundle types: modern and versioned. The **modern bundle** type is the one most commonly found on the system and is used to implement applications, plug-ins, and most other executables. The **versioned bundle** type is used almost exclusively to implement frameworks and umbrella frameworks.

For information about the structure of a modern bundle, see "Anatomy of a Modern Bundle" (page 11). For information about the structure of versioned bundles, see "Anatomy of Framework Bundles" in *Framework Programming Guide*.

Programmatic Support for Bundles

Programs that refer to bundles, or are themselves bundled, can take advantage of interfaces in Core Foundation and Cocoa to access the contents of a bundle. Using these interfaces you can find bundle resources, get information about the bundle's configuration, and load executable code. Cocoa programs use an `NSBundle` object to manage bundle information. Most other programs use the `CFBundle` object and C-based functions that are part of Core Foundation.

Note: Unlike many other Core Foundation and Cocoa types, `CFBundle` and `NSBundle` are not toll-free bridged data types and cannot be used interchangeably. However, you can extract the bundle path information from either object and use it to create the other.

Anatomy of a Modern Bundle

Modern bundles reflect a structure that is different from the bundle organization that came before Mac OS X. This bundle type is used by applications and plug-ins and is the most common type of bundle.

Bundles have evolved significantly over the years but the overall goal has been the same. The bundle organization makes it easier for the application to find its resources while making it harder for users to interfere with those resources. Because the Finder treats most bundles as opaque entities, it is difficult for casual users to move or delete the resources an application might need.

Everything an application requires should be stored inside its bundle directory. This includes the following types of resources:

- executable code
- images
- sounds
- nib files and other archived user-interface definitions
- string resources
- Resource Manager–style resource files
- localized versions of your resources
- private libraries and frameworks
- plug-ins and other loadable bundles

The basic structure of a modern bundle is very simple. At the top-level of the bundle is a directory named `Contents`. This directory contains everything, including the resources, executable code, private frameworks, private plug-ins, and support files needed by the application or plug-in. While the `Contents` directory might seem superfluous, it identifies the bundle as a modern-style bundle and separates it from document and legacy bundle types.

Inside the `Contents` directory, you can add numerous other directories and files to implement your bundle. The following sections explain the types of files you can add. These sections are cumulative, that is, each section builds on the contents of the previous section to show you how to build a more full-featured bundle in stages.

Adding Configuration Information

For the Finder to recognize an executable bundle as such, you need to include an `Info.plist` file (also known as an **information property list** file). This file contains XML property-list data that identifies the configuration of your bundle. For a minimal bundle, this file would contain very little information, most likely just the name and identifier of the bundle. For more complex bundles, the `Info.plist` file includes much more information, such as the following:

- The bundle name
- The bundle identifier code
- The bundle version
- The bundle signature (similar to creator types on Mac OS 9)
- The location of the bundle executable and entry point
- Information about how the bundle should be handled by Launch Services
- Information about the document types recognized by the bundle
- The services exported by the bundle

The exact information you put into your `Info.plist` file is dependent on your bundle's needs and can be localized as necessary. For more information on this file, see *Runtime Configuration Guidelines*.

Important: Bundle resources are located using a case-sensitive search. Therefore, the name of your information property list file must start with a capital "I".

Adding Executable Code

The most important thing to add to an executable bundle is your executable binary. This is the file compiled from your source code and linked into a form that can be executed on the target computer. Binaries that run natively on Mac OS X go in the `Contents/MacOS` directory inside your bundle. If you have a helper application or other binary code that runs only the Classic compatibility environment, you can put it into a `MacOSClassic` directory instead.

Listing 1 shows a bundle whose main executable runs natively in Mac OS X but that uses a helper tool that runs only in the Classic compatibility environment.

Listing 1 A bundle with executable code

```
- MyBundle/
    Contents/
        MacOSClassic/
            Helper Tool
        MacOS/
            MyApp
        Info.plist
```

Some bundles include an alias to the main executable at the same level as the `Contents` directory. This is a legacy feature to make it easier for users on Mac OS 9 systems to find and launch the application. It is not needed for applications that run only on Mac OS X.

Adding Non-Localized Resources

Resources in your bundle belong in a separate `Resources` directory just inside the `Contents` directory. This is where you put image, sound, nib, strings, and icon files among others. The top-level `Resources` directory contains resources that apply to all localized versions of your application. You can also put global resources in a custom subdirectory. If you have language-specific resources, put them into language-specific subdirectories, as described in "Adding Localized Resources" (page 13).

Note: You should always place resources in the `Resources` directory as opposed to embedding them in the resource fork of your executable file. The use of resource forks is not recommended as it precludes the use of your bundle on non-HFS file systems. Resource forks are also not supported by Mach-O executables.

Listing 2 shows a bundle that includes some resources in the native language of the program. The files in the `WaterSounds` directory are considered global resources just like `Hand.tiff`, `Horse.jpg`, and `MyBundle.icns`.

Listing 2 A bundle with global resources

```
- MyBundle/
    MyApp /* alias to Contents/MacOSClassic/MyApp */
    Contents/
        MacOSClassic/
            Helper Tool
            MyApp
        MacOS/
            Helper Tool
            MyApp
        Info.plist
        Resources/
            Hand.tiff
            Horse.jpg
            MyBundle.icns
            WaterSounds/
                Water1.aiff
                Water2.aiff
```

One special resource that belongs in your top-level `Resources` directory is your application icon file. By convention, this file takes the name of the bundle and an extension of `.icns`; the image format can be any supported type, but if no extension is specified, the system assumes `.icns`.

Adding Localized Resources

Within the `Resources` directory, you can create one or more additional subdirectories to store language-specific resources. The name of each directory is based on the language and region of the translation followed by the `.lproj` extension. For all languages, you can use either the ISO 639 or ISO 3166 standards for specifying language directories. The ISO 639 format is as follows:

language`.lproj`

The ISO 3166 format is as follows:

language_region`.lproj`

For both formats, the language and region designators are two-letter ISO codes. For each language, you must include a language-specific resource directory and may include one or more region-specific directories as well. For example, for the English language, you would put all of your English resources in an `en.lproj` directory and could include a copy of some resources specific to the United States in a `en_US.lproj` directory. For backwards compatibility, `NSBundle` and `CFBundle` also support human-readable names for several common languages. For more information, see "Language Designations" in *Internationalization Programming Topics*.

Each of your language-specific resource directories should contain a copy of the same resource files. The names of the files must all be the same; only the content of the files differs based on the language. When your application asks for a resource file, Mac OS X uses the current language preferences to return the file from the appropriate directory.

Listing 3 shows a bundle that includes localized resources for multiple foreign languages. Notice that the region-specific directories do not contain a full complement of resources. If a region-specific version of a resource is not found, the bundle interfaces search in the language-specific directory (in this case `en.lproj`) for the resource. The language-specific directory *must* contain a complete copy of the language-specific resources.

Listing 3 A bundle with localized resources

```
- MyBundle/
    MyApp /* alias to Contents/MacOSClassic/MyApp */
    Contents/
        MacOSClassic/
            Helper Tool
            MyApp
        MacOS/
            Helper Tool
            MyApp
        Info.plist
        Resources/
            Hand.tiff
            Horse.jpg
            MyBundle.icns
            WaterSounds/
                Water1.aiff
                Water2.aiff
            en_GB.lproj/
                MyApp.nib
                bird.tiff
                Localizable.strings
            en_US.lproj/
                MyApp.nib
                Localizable.strings
            en.lproj/
                MyApp.nib
                bird.tiff
                Bye.txt
                house.jpg
                InfoPlist.strings
                Localizable.strings
                CitySounds/
                    city1.aiff
                    city2.aiff
```

```
Japanese.lproj/
    MyApp.nib
    bird.tiff
    Bye.txt
    house.jpg
    InfoPlist.strings
    Localizable.strings
    CitySounds/
        city1.aiff
        city2.aiff
```

For more information about internationalization of your software and localization of its resources, see *Internationalization Programming Topics*.

Adding Platform-Specific Resources

In addition to localized resources, you can also add resources that apply to a specific operating system supported by the bundle. Platform-specific resources are created by adding specific identifiers to existing resource names. Mac OS X defines the identifiers `macosclassic` for Mac OS 9 resources and `macos` for Mac OS X resources.

Note: Support for platform-specific resources exists primarily to maintain compatibility with legacy applications. New applications should have only Mac OS X resources and therefore should not need platform-specific versions.

To construct a platform-specific resource name, insert a hyphen followed by the appropriate identifier immediately before the filename extension of the resource. For example, if you have a resource named `Fish.jpg` its Mac OS 9 name would be `Fish-macosclassic.jpg` and its Mac OS X name would be `Fish-macos.jpg`. If your bundle requests the resource `Fish.jpg`, it would get back a path to `Fish-macosclassic.jpg` on Mac OS 9 or `Fish-macos.jpg` on Mac OS X. If no platform-specific version existed, your bundle would get back a path to `Fish.jpg`.

Important: If you include platform-specific versions of a resource, you must also include the platform-generic version in the same directory.

Platform-specific resources can reside in the global `Resources` directory or in any of the language-specific subdirectories in your bundle. Listing 4 shows a bundle that includes platform-specific variants of the `house.jpg` file.

Listing 4 A bundle with platform-specific resources

```
- MyBundle/
        MyApp /* alias to Contents/MacOSClassic/MyApp */
        Contents/
            MacOSClassic/
                Helper Tool
                MyApp
            MacOS/
                Helper Tool
                MyApp
```

```
Info.plist
Resources/
    Hand.tiff
    Horse.jpg
    MyBundle.icns
    WaterSounds/
        Water1.aiff
        Water2.aiff
    en_GB.lproj
        MyApp.nib
        bird.tiff
        Localizable.strings
    en_US.lproj/
        MyApp.nib
        Localizable.strings
    en.lproj
        MyApp.nib
        bird.tiff
        Bye.txt
        house.jpg
        house-macos.jpg
        house-macosclassic.jpg
        InfoPlist.strings
        Localizable.strings
        CitySounds/
            city1.aiff
            city2.aiff
    Japanese.lproj/
        MyApp.nib
        bird.tiff
        Bye.txt
        house.jpg
        house-macos.jpg
        house-macosclassic.jpg
        InfoPlist.strings
        Localizable.strings
        CitySounds/
            city1.aiff
            city2.aiff
```

Adding Additional Support Files

While executable and resource files are standard in most bundles, there are several other special directories that are not needed by most bundles. These directories include the following:

- A `Frameworks` directory contains any private libraries and frameworks used by the executable.
- A `PlugIns` directory contains loadable bundles that extend the basic features of the executable.
- A `SharedSupport` directory contains additional non-critical resources that do not impact the ability of the application to run. For example, this directory might include things like document templates, clip art, and tutorials.

All of these directories are inextricably bound to the application. This is particularly true for frameworks. The dynamic shared libraries of embedded frameworks are revision-locked and cannot be superseded by any other, even newer, versions that may be available to the operating system.

Listing 5 shows the bundle structure for a fully-localized application that supports these directories.

Listing 5 The bundle layout of a complex application

```
- MyBundle/
    MyApp /* alias to Contents/MacOSClassic/MyApp */
    Contents/
        MacOSClassic/
            Helper Tool
            MyApp
        MacOS/
            Helper Tool
            MyApp
        Info.plist
        Resources/
            Hand.tiff
            Horse.jpg
            MyBundle.icns
            WaterSounds/
                Water1.aiff
                Water2.aiff
            en_GB.lproj
                MyApp.nib
                bird.tiff
                Localizable.strings
            en_US.lproj/
                MyApp.nib
                Localizable.strings
            en.lproj
                MyApp.nib
                bird.tiff
                Bye.txt
                house.jpg
                house-macos.jpg
                house-macosclassic.jpg
                InfoPlist.strings
                Localizable.strings
                CitySounds/
                    city1.aiff
                    city2.aiff
            Japanese.lproj/
                MyApp.nib
                bird.tiff
                Bye.txt
                house.jpg
                house-macos.jpg
                house-macosclassic.jpg
                InfoPlist.strings
                Localizable.strings
                CitySounds/
                    city1.aiff
                    city2.aiff
        Frameworks/
        PlugIns/
```

```
SharedSupport/
```

For information on how to embed frameworks in your application bundle, see *Framework Programming Guide*.

Bundles and the Resource Manager

A bundle can contain any number of files with Resource Manager–style resources in their data forks. These resource files—which, by convention, have an extension of `.rsrc` —are treated as bundle resources just as any other kind of file under the `Resources` directory.

If your application requires specific resources at launch time, you can place those resources in one of two places and have Bundle Services load them for you automatically. For nonlocalized resources, place a file in your top-level `Resources` directory whose name is the same as the name of your executable but with a `.rsrc` extension. For localized resources, place a `Localized.rsrc` file in each of your language-specific directories.

For example, suppose your application is called `MyApp` and contains localizations for English, German, and Japanese. If you had both localized and nonlocalized resources you needed to load at launch time, your `Resources` directory would contain the following additional files:

```
MyApp.app/
    Contents/
        Resources/
            MyApp.rsrc
            English.lproj/
                Localized.rsrc
            German.lproj/
                Localized.rsrc
            Japanese.lproj/
                Localized.rsrc
```

The automatic loading of resources applies only to the listed files in your application bundle. If you load resources from frameworks or other loadable bundles, you must load those resources manually using the `CFBundleOpenBundleResourceMap` or `CFBundleOpenBundleResourceFiles` functions. These functions open the resource files and return reference numbers that you can pass to Resource Manager functions.

The following example shows you how to load an arbitrary resource from your bundle. The first step is to get a `CFURL` and then convert it to a `FSRef`. Once you have an `FSRef`, you can pass it to an appropriate Resource Manager call.

```
CFBundleRef myBundle;
CFURLRef tempURL;
FSRef myResource;

myBundle = CFBundleGetMainBundle();
tempURL = CFBundleCopyResourceURL (myBundle, CFSTR("MyResource"),
        CFSTR("rsrc"), NULL);

if (CFURLGetFSRef (tempURL, &myResource))
{
    // Open the resource using Resource Manager calls
}
```

Searching for Bundle Resources

As long as you use the `CFBundle` and `NSBundle` programming interfaces, your bundle code need never concern itself with how resources are retrieved from a bundle. Both `CFBundle` and `NSBundle` automatically retrieve the appropriate language-specific resource based on the available user settings and bundle information. However, you still have to put all those language-specific resources into your bundle, so knowing how they are retrieved is important.

The bundle programming interfaces follow a specific search algorithm to locate resources within the bundle. Global resources have the highest priority, followed by region- and language-specific resources. When considering region- and language-specific resources, the algorithm takes into account both the settings for the current user and development region information in the bundle's `Info.plist` file. The bundle interfaces use a case-sensitive search to locate the appropriate resource files. The following list shows the order in which resources are searched:

1. Global resources
2. Region-specific resources (based on the user's region preferences)
3. Language-specific resources (based on the user's language preferences)
4. Development language of the bundle (as specified by the `CFBundleDevelopmentRegionKey` in the bundle's `Info.plist` file.)

If a resource is found, the bundle interfaces then check to see if there is a platform-specific version that should be returned instead. If one is available, it is returned; otherwise, the original platform-generic resource is returned.

Note: See "Adding Localized Resources" (page 13) for an example that shows where to put global and language-specific resources reside within a bundle.

Because global resources take precedence over language-specific resources, there should *never* be both a global and localized version of a given resource. If a global version of a resource exists, language-specific versions of the same resource are never returned. The reason for this precedence is performance. If localized resources were searched first, the bundle routines might search needlessly in several localized resource directories before discovering the global resource.

Also notice that if your bundle includes any platform-specific resources, you *must* also include a complete set of platform-generic resources as well. Again, the reason is performance. You would typically create a generic resource that applies to most platforms and then offer customized versions only if it was relevant. See "Adding Platform-Specific Resources" (page 15) for information and examples of how to create platform-specific resources.

Packages and the Finder

The Finder treats packaged directories differently than other directories. Instead of displaying the contents of the packaged directory, the Finder treats it as if it were a single file. Hiding the directory's contents prevents casual users from making changes that might damage the package contents. For example, rearranging or deleting resources from an application package might prevent the application from running correctly.

Most bundles are also packages. In particular, applications and plug-ins are typically delivered as packages and thus appear to the user as a single file. Some bundles, such as frameworks, are not delivered as packages. In the case of frameworks, this is done so that the user can browse the contents of the framework; in particular, it lets the user look at the framework header files.

Even though packages are treated as opaque files by default, it is still possible for users to view and modify their contents. On the contextual menu for package directories is a Show Package Contents command. Selecting this command displays a new Finder window set to the top level of the package directory. The user can navigate the package's directory structure and make changes as if it were a regular directory hierarchy.

The Finder identifies packages by any of the following mechanisms:

- The directory has a known extension: `.app`, `.bundle`, `.framework`, `.plugin`, `.kext`, and so on.
- The directory has its bundle bit set.
- The directory has a known structure type indicating it is a modern or versioned bundle.

Once identified, the Finder may also modify the name of a package in any of the following ways:

- If the package is an application, the Finder hides the `.app` extension in most cases.
- If the package supports localized display names and the user has not manually changed the package name, the Finder displays the name that matches the user's current language settings.

The Finder hides the `.app` extension most of the time. However, if an application name contains another extension, the Finder shows the `.app`. extension to prevent confusion. For example, if you rename the `Chess` application by adding a `.mov` extension to the end of its name, the Finder displays the resulting bundle name as `Chess.mov.app` to prevent users from thinking `Chess.mov` is a QuickTime file.

Document Packages

If your document file formats are getting too complex to manage because of several different types of data, you might consider adopting a package format for your documents. Document packages give the illusion of a single document to users but provide you with more flexibility in how you store the document data internally. Especially if you use several different types of standard data formats, such as JPEG, GIF, or XML, document packages make accessing that data much easier.

Defining Your Document Directory Structure

Apple does not prescribe any specific structure for document packages. The contents and organization of the package directory are left to you. You are encouraged, however, to create either a flat directory structure or use the framework bundle structure, which involves placing your files in a top-level `Resources` subdirectory. (For more information about the bundle structure of frameworks, see *Framework Programming Guide*.)

Registering Your Document Type

To register a document as a package, you must modify the document type information in your application's information property list (`Info.plist`) file. The `CFBundleDocumentTypes` key stores information about the document types your application supports. For each document package type, include the `LSTypeIsPackage` key with an appropriate value. The presence of this key tells the Finder and Launch Services to treat directories with the given file extension as a package. See "Property List Key Reference" in *Runtime Configuration Guidelines* for more information about `Info.plist` keys.

Document packages should always have an extension to identify them—even though that extension may be hidden by the user. The extension allows the Finder to identify your document directory and treat it as a package. You should never associate a document package with a MIME type or 4-byte OS type.

Accessing Your Document Contents

There are several ways to access the contents of a document package. Because a document package is a directory, you can access the document's contents using any appropriate file-system routines. If you use a bundle structure for your document, you can also use the `NSBundle` or `CFBundle` routines. Use of a bundle structure is especially appropriate for documents that store multiple localizations.

If your document package uses a flat directory structure or contains a fixed set of content files, you might find using file-system routines faster and easier than using `NSBundle` or `CFBundle`. If the contents of your document can fluctuate, you should consider using a bundle structure and `NSBundle` or `CFBundle` to simplify the dynamic discovery of files inside your document.

If you are creating a Cocoa application, you must also remember to customize the way your `NSDocument` subclass loads the contents of the document package. The traditional technique of using the `loadDataRepresentation:ofType:` and `dataRepresentationOfType:` methods to read and write data are intended for a single file document. To handle a document package, you must use the `readFromFile:ofType:` and `writeToFile:ofType:` methods or use an `NSFileWrapper` object instead. For information about reading and writing document data from your `NSDocument` subclass, see *Document-Based Applications Overview*.

Guidelines for Using Bundles

Bundles are the preferred packaging mechanism for most types of software in Mac OS X. The bundle structure lets you group executable code and the resources to support that code in one place and in an organized way. The following guidelines offer some additional advice on how to use bundles:

- Store all resources required by an application to run inside the application bundle. This includes all images, strings files, localizable resources and plug-ins. See "Anatomy of a Modern Bundle" (page 11) for more information.
- If you plan to load C++ code from a bundle, you might want to mark the symbols you plan to load as `extern "C"`. Neither `NSBundle` or `CFBundle` know about C++ name mangling conventions, so marking your symbols this way can make it much easier to identify them later.
- You cannot use the `NSBundle` class to load CFM-based code. If you need to load CFM-based code, you must use the functions of `CFBundle` or `CFPlugin`. You may load CFM-based plugins from a Mach-O executable using this technique.
- You cannot use `CFBundle` to load Objective-C or Java code. For Cocoa or Java code, you should always use `NSBundle`.
- Always include an information-property list in your bundle. Make sure you include the keys recommended for your bundle type, as discussed in *Runtime Configuration Guidelines*.

Creating Bundles

Because bundles are simply file and directory hierarchies in the file system, it is possible to construct them by hand by placing files manually using the organization specified in "Anatomy of a Modern Bundle" (page 11). It is far easier, however, to let Xcode take care of the bundle structure for you. When building a bundle project type, Xcode performs the following default operations to create your bundle:

- Copies the contents of the Resources group into your bundle's `Resources` directory
- Creates an information property list (`Info.plist`) file from information you provide in the Inspector window of your target
- Copies any compiled code into the platform-specific subdirectory of the `Contents` directory

Xcode currently supports the creation of modern and versioned bundles, either of which can be accessed from your code using `CFBundle` or `NSBundle`. However, if you have bundles left over from development on the NeXT operating system, you can convert those bundles to the modern structure using the command-line utility `/Developer/Makefiles/pb_makefiles/convertBundle`.

If your application requires localization, you can create strings files using the `genstrings` development tool. This tool parses your source files looking for occurrences of specific macro and function calls and uses them to create annotated entries in a strings file. After translation, the strings files are then placed into the appropriate `.lproj` directories. See *Internationalization Programming Topics* for more information on preparing and loading strings.

The final step is for you to localize, and if necessary, create platform-specific versions of your other application resources and place them in the language-specific `.lproj` directories of your bundle.

Locating and Opening Bundles

Before you can access a bundle's resources, you must first obtain an appropriate `NSBundle` or `CFBundle` object. The following sections outline the different ways you create these objects from your code.

Opening the Main Bundle

The main bundle is the bundle that contains your running code. This is the most commonly used bundle for any program and is what you use to load your strings, images, and other resource files. Because it is a meta bundle, the main bundle is the easiest to retrieve. To get it, call either `CFBundleGetMainBundle` function or use the `mainBundle` class method of `NSBundle`. Listing 1 shows an example of loading the main bundle from a Carbon application.

Listing 1 Locating the main bundle from Core Foundation

```
CFBundleRef mainBundle;

// Get the main bundle for the app
mainBundle = CFBundleGetMainBundle();
```

Listing 2 shows this same code example written in Objective-C and using the `NSBundle` class.

Listing 2 Locating the main bundle from Cocoa

```
NSBundle* mainBundle;

// Get the main bundle for the app.
mainBundle = [NSBundle mainBundle];
```

When getting the main bundle it is still a good idea to make sure the value you get back represents a valid bundle. In particular, you might get a `NULL` value for the main bundle in the following situations:

- If a program is not bundled, attempting to get the main bundle might return a `NULL` value. The bundle code may try to create a main bundle for you to represent your program's contents but in some exceptional cases, it cannot and returns `NULL` .
- If the agent launching the program does not specify the full path to the program's executable in the `argv` parameters, the main bundle value might be `NULL`. Bundles rely on either the path to the executable being in `argv[0]` or the presence of the executable's path in the `PATH` environment variable. If neither of these is present, the bundle routines might not be able to find the main bundle directory. Programs launched by `xinetd` often experience this problem when `xinetd` changes the current directory to `/`.

Locating Bundles by Path

If you want to access a bundle other than your main bundle, one way to create an appropriate bundle object is with a path to the bundle. The `CFBundleCreate` function in Core Foundation takes a `CFURLRef` specifying the path to the bundle and returns a corresponding `CFBundle` data type. Similarly, you can use the `bundleWithPath:` method of `NSBundle` to create a bundle object from an `NSString`.

Listing 3 shows an example of how to create a `CFBundle` from a string specifying the path. The main trick is to convert the string to a `CFURLRef` object so that it can be passed to the `CFBundleCreate` function.

Listing 3 Locating a Core Foundation bundle using its path

```
CFURLRef bundleURL;
CFBundleRef myBundle;

// Make a CFURLRef from the CFString representation of the
// bundle's path.
bundleURL = CFURLCreateWithFileSystemPath(
                kCFAllocatorDefault,
                CFSTR("/Local/Library/MyBundle.bundle"),
                kCFURLPOSIXPathStyle,
                true );

// Make a bundle instance using the URLRef.
myBundle = CFBundleCreate( kCFAllocatorDefault, bundleURL );

// Any CF objects returned from functions with "create" or
// "copy" in their names must be released by us!
CFRelease( bundleURL );
CFRelease( myBundle );
```

The preceding example can be rewritten for Cocoa using the code shown in Listing 4.

Listing 4 Locating a Cocoa bundle using its path

```
NSBundle* myBundle;

// Get the main bundle for the app.
myBundle = [NSBundle bundleWithPath:@"/Local/Library/MyBundle.bundle"];
```

Locating Bundles in Known Directories

Even if you do not know the exact path to a bundle, there are still situations where you can search for it by name. One example is if your application contains several embedded plug-ins in a `PlugIns` directory. Because this directory is inside of your application bundle, you can use the `CFBundle` functions for locating resources to get the path to each plug-in.

To load a set of plug-ins, use the `CFBundleCreateBundlesFromDirectory` function to create new `CFBundle` objects for all of the plug-ins in a given directory. Listing 5 is similar to the previous example, but this time the code retrieves `CFBundle` objects for all of the plug-ins in the application's `PlugIns` directory.

Listing 5 Obtaining bundle references for a set of plug-ins

```
CFBundleRef mainBundle = CFBundleGetMainBundle();
CFURLRef plugInsURL;
CFArrayRef bundleArray;

// Get the URL to the application's PlugIns directory.
plugInsURL = CFBundleCopyBuiltInPlugInsURL(mainBundle);

// Get the bundle objects for the application's plug-ins.
bundleArray = CFBundleCreateBundlesFromDirectory( kCFAllocatorDefault,
                    plugInsURL, NULL );

// Release the CF objects
CFRelease( plugInsURL );
CFRelease( bundleArray );
```

Locating Bundles by Identifier

Bundle identifiers make it possible to locate already loaded bundles at runtime. This is a useful way for your code to locate its own bundle at runtime. Storing a bundle identifier can be more efficient than storing a reference to the bundle itself. When you need to access the bundle again, you can use the identifier to retrieve the `CFBundle` object.

Each bundle you create should have a bundle identifier in its information property-list (`Info.plist`) file. The `CFBundleIdentifier` key contains the bundle identifier string, which traditionally uses Java-style package naming conventions. For example, a Finder plug-in from Apple might use the string `com.apple.Finder.MyGetInfoPlugin` as its bundle identifier. Including the domain name of your company in the string helps avoid collisions with bundle developers in other companies.

Listing 6 shows how to retrieve a bundle using its bundle identifier. Remember that a bundle identifier can *only* be used to locate an existing `CFBundle` instance (including the main bundle and bundles for all statically linked frameworks). If your bundle has been opened and its code is running, then you can locate it using its bundle identifier.

Listing 6 Locating a bundle using its identifier

```
CFBundleRef requestedBundle;

 // Look for a bundle using its identifier
 requestedBundle = CFBundleGetBundleWithIdentifier(
        CFSTR("com.apple.Finder.MyGetInfoPlugIn") );
```

You can also locate bundles by their identifier in Cocoa. The `NSBundle` class defines the class method `bundleWithIdentifier:` to find and return an existing bundle.

Searching for Related Bundles

If you are writing a Cocoa application, you can obtain a list of bundles related to the application by calling the `allBundles` and `allFrameworks` class methods of NSBundle. These methods create an array of `NSBundle` objects corresponding to the bundles or frameworks currently in use by your application. You can use these methods as convenience functions rather than maintain a collection of loaded bundles yourself.

The `bundleForClass:` class method is another way get related bundle information in a Cocoa application. This method returns the bundle in which a particular class is defined. Again, this method is mostly for convenience so that you do not have to retain a pointer to an `NSBundle` object that you may use only occasionally.

Locating Resources Inside a Bundle

One of the main reasons to get a bundle object is so that you can load resources from your bundle. Both `CFBundle` and `NSBundle` provide interfaces for finding specific resource types in your bundle. These interfaces return a path to the resource that you can then use to load it.

Resource types are typically specified using filename extensions, not the file type information used in previous versions of Mac OS; therefore, you must make sure your files are named properly. For Core Foundation programs, the most commonly used functions for finding resources are `CFBundleCopyResourceURL` and `CFBundleCopyResourceURLsOfType`. These functions let you retrieve resources using name, type, and directory information. Similarly, Cocoa applications typically use the `pathForResource:ofType:` method and `pathForResourcesOfType:InDirectory:` methods to retrieve resources.

Important: `CFBundle` and `NSBundle` consider case when searching for resource files in the bundle directory. This case-sensitive search occurs even on file systems (such as HFS+) that are not case sensitive when it comes to file names.

Even if you do not have a bundle object, you can still load resources from bundles whose paths you know. Both Core Foundation and Cocoa provide API for searching bundles with only a path to the bundle. However, it is important to remember that searching a bundle for multiple resources is always faster using a bundle object. The bundle objects cache search information as they go, so subsequent searches are usually faster.

The Core Foundation and Cocoa API take into account localized versions of resources when determining which paths to return. For information on how these interfaces determine which files to return, see "Searching for Bundle Resources" (page 21).

Suppose you have placed an image called `Seagull.jpg` in your application's main bundle. Listing 1 shows you how to search for this image by name and type using the Core Foundation function `CFBundleCopyResourceURL`. In this case, the code searches for the file named "Seagull" with the file type (filename extension) of "jpg". This example searches for the resource starting at the top level of the bundle's `Resources` directory.

Listing 1 Locating resources inside a bundle by name and type

```
CFURLRef    seagullURL;

// Look for a resource in the main bundle by name and type.
seagullURL = CFBundleCopyResourceURL( mainBundle,
                CFSTR("Seagull"),
                CFSTR("jpg"),
                NULL );
```

Suppose that instead of searching for one image file, you wanted to get the names of all image files in a directory called `BirdImages`. You could load all of the JPEGs in the directory using the function `CFBundleCopyResourceURLsOfType`, as shown in Listing 2.

Listing 2 Locating multiple resources by type

```
CFArrayRef  birdURLs;

// Find all of the JPEG images in a given directory.
birdURLs = CFBundleCopyResourceURLsOfType( mainBundle,
                CFSTR("jpg"),
                CFSTR("BirdImages") );
```

Note: You can search for resources that do not have a filename extension. To get the path to such a resource, specify the complete name of the resource and specify `NULL` for the resource type.

Loading and Unloading Executable Code

Most applications use only the code from their own main executable file. However, if you are developing an application that supports plug-ins or other types of loadable bundles, you need to know how to load the code from these separate bundles dynamically. Both `CFBundle` and `NSBundle` provide facilities for loading code from a bundle. Depending on your needs, you may use one or both of these objects in your coding.

The key to loading code from an external bundle is finding an appropriate entry point into the bundle's executable file. As with other plug-in schemes, this requires some coordination between the application developer and plug-in developer. You can publish a custom API for bundle's to implement or define a formal plug-in interface. In either case, once you have an appropriate bundle or plug-in, you need a way to access the functions or classes implemented by the external code.

Note: Another option for loading Mach-O code directly is to use the `NSModule` loading routines. However, these routines typically require more work to use and are less preferable than `CFBundle` or `NSBundle`. For more information, see *Mac OS X ABI Mach-O File Format Reference* in Mac OS X Documentation or see the `NSModule` man pages.

Loading Functions

If you are working in C, C++, or even in Objective-C, you can publish your interface as a set of C-based symbols, such as function pointers and global variables. Using the `CFBundle` interfaces, you can load references to those symbols from a bundle's executable file.

You can retrieve symbols using any of several `CFBundle` interfaces. To retrieve function pointers, call either `CFBundleGetFunctionPointerForName` or `CFBundleGetFunctionPointersForNames`. To retrieve a pointer to a global variable, call `CFBundleGetDataPointerForName` or `CFBundleGetDataPointersForNames`. For example, suppose a loadable bundle defines the function shown in Listing 1.

Listing 1 An example function for a loadable bundle

```
// Add one to the incoming value and return it.
long addOne(short number)
{
    return ( (long)number + 1 );
}
```

Given a `CFBundle` object for the loadable bundle, you would need to search for the desired function before you could use it in your code. Listing 2 shows a code fragment that illustrates this process. In this example, the `myBundle` variable is a `CFBundle` object pointing to the bundle.

Listing 2 Finding a function in a loadable bundle

```
// Function pointer.
```

```
AddOneFunctionPtr addOne = NULL;

// Value returned from the loaded function.
long result;

// Get a pointer to the function.
addOne = (void*)CFBundleGetFunctionPointerForName(
            myBundle, CFSTR("addOne") );

    // If the function was found, call it with a test value.
if (addOne)
{
    // This should add 1 to whatever was passed in
    result = addOne ( 23 );
}
```

Loading Objective-C Classes

If you are writing a Cocoa application, you can load the code for an entire class using the methods of `NSBundle`. The NSBundle methods for loading a class are aimed squarely for Objective-C developers and cannot be used to load classes written in C++ or other object-oriented languages.

If a loadable bundle defines a principal class, you can load it using the `principalClass` method of `NSBundle`. The `principalClass` method uses the `NSPrincipalClass` key of the bundle's `Info.plist` file to identify and load the desired class. Using the principal class alleviates the need to agree on specific naming conventions for external classes, instead letting you focus on the behavior of those interfaces. For example, you might use an instance of the principal class as a factory for creating other relevant objects.

If you want to load an arbitrary class from a loadable bundle, call the `classNamed:` method of `NSBundle`. This method searches the bundle for a class matching the name you specify. If the class exists in the bundle, the method returns the corresponding `Class` object, which you can then use to create instances of the class.

Listing 3 shows you a sample method for loading a bundle's principal class.

Listing 3 Loading the principal class of a bundle

```
- (void)loadBundle:(NSString*)bundlePath
{
    Class exampleClass;
    id newInstance;
    NSBundle *bundleToLoad = [NSBundle bundleWithPath:bundlePath];
    if (exampleClass = [bundleToLoad principalClass])
    {
        newInstance = [[exampleClass alloc] init];
        // [newInstance doSomething];
    }
}
```

For more information about `NSBundle` methods, see the `NSBundle` class description in the Foundation reference.

Unloading Bundles

You cannot currently unload the contents of an `NSBundle` object. You can unload the contents of a `CFBundle` object using `CFBundleUnloadExecutable`. If your bundle may be unloaded, you need to ensure that string constants are handled correctly by setting an appropriate compiler flag.

When you compile a bundle with a minimum deployment target of Mac OS X 10.2 (or later), the compiler automatically switches to generating “truly-constant” strings in response to `CFSTR("...")`. This can also be achieved by compiling with the flag `-fconstant-cfstrings`. Constant strings have many benefits and should be used when possible, however if you reference constant strings after the executable containing them is unloaded, the references will be invalid and will cause a crash. This might happen even if the strings have been retained, for example, as a result of being put in data structures, retained directly, and, in some cases, even copied. Rather than trying to make sure all such references are cleaned up at unload time (and some references might be created within the libraries, making them hard to track), it is best to compile unloadable bundles with the flag `-fno-constant-cfstrings`.

Getting Information From a Bundle

Both `CFBundle` and `NSBundle` contain interfaces for retrieving information about the bundle itself. From an appropriate bundle object, you can retrieve path information for the bundle as well as configuration data from its information property list. Remember though that the bundle provides information as read-only data. For information on how to modify the value of a property list from Core Foundation, see *Property List Programming Topics for Core Foundation*.

Getting Path Information

With a valid bundle object, you can retrieve the path to the bundle as well as paths to many of its subdirectories. Modern bundles can contain many specific subdirectories (see "Anatomy of a Modern Bundle" (page 11)). Most of these directories contain plug-ins and other executable code or they contain various types of resource files used by the application. Using the available interfaces to retrieve directory paths insulates your code from having to know the exact structure of the bundle.

Core Foundation defines functions for retrieving several different internal bundle directories. To get the path of the bundle itself, you can use the `CFBundleCopyBundleURL` function. Core Foundation always returns bundle paths in a `CFURLRef` object. You can use this object to extract a `CFStringRef` that you can then pass to other Core Foundation routines. For a complete list of path-based functions, see the *CFBundle Reference*.

`NSBundle` also contains similar methods for retrieving paths to a bundle's internal directories. It also contains a `bundlePath` method for getting the path to the bundle itself. However, `NSBundle` returns path information in an `NSString` object, which you can pass directly to most other `NSBundle` methods. For more information, see the `NSBundle` class description.

Getting Configuration Data

One file that every bundle should contain is an information property list (`Info.plist`) file. This file is an XML-based text file that contains specific types of key-value pairs. These key-value pairs specify information about the bundle, such as its ID string, version number, development region, type, and other important properties. (See *Runtime Configuration Guidelines* for the list of keys you can include in this file.) Bundles may also include other types of configuration data, mostly organized in XML-based property lists.

Core Foundation offers functions for retrieving several specific pieces of data from a bundle's information property list file, including the bundle's ID string, version, and development region. You can retrieve the localized value for a key using the `CFBundleGetValueForInfoDictionaryKey` function. You can also retrieve the entire dictionary of non-localized keys using `CFBundleGetInfoDictionary`. See *CFBundle Reference* for a complete list of functions.

`NSBundle` provides the `objectForInfoDictionaryKey:` and `infoDictionary` methods for retrieving information property list data. The `objectForInfoDictionaryKey:` method returns the localized value for a key and is the preferred method to call. The `infoDictionary` method returns an `NSDictionary` with all of the keys from the property list; however, it does not return any localized values for these keys. For more information, see the `NSBundle` class description.

Note: Because they take localized values into account, `CFBundleGetValueForInfoDictionaryKey` and `objectForInfoDictionaryKey:` are the preferred interfaces for retrieving keys.

Listing 1 demonstrates how to retrieve the bundle's version number from the information property list using Core Foundation functions. Though the value in the information property list may be written as a string, for example "2.1.0b7", the value is returned as an unsigned long integer similar to the value in a `vers` resource on Mac OS 9.

Listing 1 Obtaining the bundle's version

```
// This is the 'vers' resource style value for 1.0.0
#define kMyBundleVersion1 0x01008000

UInt32  bundleVersion;

// Look for the bundle's version number.
bundleVersion = CFBundleGetVersionNumber( mainBundle );

// Check the bundle version for compatibility with the app.
if (bundleVersion < kMyBundleVersion1)
    return (kErrorFatalBundleTooOld);
```

Listing 2 shows you how to retrieve arbitrary values from the information property list using `CFBundleGetInfoDictionary`. Because the resulting information property list is an instance of the standard Core Foundation type `CFDictionaryRef`, you can use the dictionary lookup routines from `CFDictionary.h` to find and retrieve your properties.

Listing 2 Retrieving information from a bundle's information property list

```
CFDictionaryRef bundleInfoDict;
CFStringRef     myPropertyString;

// Get an instance of the non-localized keys.
bundleInfoDict = CFBundleGetInfoDictionary( myBundle );

// If we succeeded, look for our property.
if ( bundleInfoDict != NULL ) {
    myPropertyString = CFDictionaryGetValue( bundleInfoDict,
                CFSTR("MyPropertyKey") );
}
```

It is also possible to obtain an instance of a bundle's information dictionary without a bundle object. To do this you use either the Core Foundation function `CFBundleCopyInfoDictionaryInDirectory` or the Cocoa `NSDictionary` class. This can be useful for searching the information property lists of a set of bundles without first creating bundle objects.

Working With Localized Strings

Bundles are able to retrieve localized strings to best suit the user's preferences. Whereas they return a path for other resources, bundles know how to lookup and return strings from within strings resource files.

Both `CFBundle` and `NSBundle` define a single interface for retrieving strings. For `CFBundle`, it is the `CFBundleCopyLocalizedString` function. For `NSBundle`, it is the `localizedStringForKey:value:table:` method. However, both Cocoa and Core Foundation also define convenience macros for retrieving strings from known locations.

There are several advantages to using the convenience macros instead of the corresponding `CFBundle` and `NSBundle` API. First, the macros are easier to use for certain common cases. Second, they are recognized by `genstrings` command-line tool, which automatically creates strings files based on the contents of your source code. (The `CFBundle` and `NSBundle` API are not recognized by `genstrings`.) Third, the macros let you specify a comment string argument to aid the translator. Comments are ignored by the compiler, but `genstrings` uses the information to annotate the generated strings files.

Core Foundation defines the following convenience macros:

- `CFCopyLocalizedString`
- `CFCopyLocalizedStringFromTable`
- `CFCopyLocalizedStringFromTableInBundle`
- `CFCopyLocalizedStringWithDefaultValue`

The Foundation framework in Cocoa defines the following convenience macros:

- `NSLocalizedString`
- `NSLocalizedStringFromTable`
- `NSLocalizedStringFromTableInBundle`
- `NSLocalizedStringWithDefaultValue`

Listing 1 demonstrates the proper usage of the Core Foundation convenience macros. The first argument of each macro is both the text to translate and the key to use when looking up the string. This string appears in the native language of the author of the program. Subsequent macros let you specify the specific strings file to search and the specific bundle to search. The final macro also lets you specify a default translation for the string if no other version is found. (Note, that the corresponding Cocoa macros use essentially the same syntax but with different data types.)

Listing 1 Using the LocalizedString macros

```
CFStringRef localString;

localString = CFCopyLocalizedString(
    CFSTR("String text to translate"),
    CFSTR("Comment to help translators."));
```

```
    localString = CFCopyLocalizedStringFromTable(
        CFSTR("String text to translate"),
        CFSTR("MyStrings"), // strings file to search
        CFSTR("Comment to help translators."));

    localString = CFCopyLocalizedStringFromTableInBundle(
        CFSTR("String text to translate"),
        CFSTR("MyStrings"), // strings file to search
        myBundle, // bundle to search
        CFSTR("Comment to help translators."));

    localString = CFCopyLocalizedStringWithDefaultValue(
        CFSTR("String text to translate"),
        CFSTR("MyStrings"), // strings file to search
        myBundle, // bundle to search
        CFSTR("Default translation if string not found"),
        CFSTR("Comment to help translators."));
```

When you run the `genstrings` tool on source code using the above macros, it creates one or more string files for the referenced keys. (See the man page for `genstrings` for instructions on running this tool.) For instance, suppose your code contains the following usage of the macro:

```
localString = CFCopyLocalizedStringFromTable(
            CFSTR("Yes"),
            CFSTR("MyStrings"),
            CFSTR("Label for an affirmative answer") );
```

After running `genstrings`, you would have a file called `MyStrings.strings`. with the following data in it:

```
"Yes" = "Yes"; /* Label for an affirmative answer */
```

You would then copy this strings file into each of your localized resource directories and translate each entry to the appropriate language. For example, translating the preceding listing from `MyStrings.strings` placed into French would yield the following entry:

```
"Yes" = "Oui"; /* Label for an affirmative answer */
```

For additional information on working with strings files, see “Extracting Localizable Strings From Your Code” and “Loading Localized Strings” in *Internationalization Programming Topics*.

Document Revision History

This table describes the changes to *Bundle Programming Guide*.

Date	Notes
2005-11-09	Updated information on how to create document bundles. Updated guidance on how to build a bundle manually.
	Clarified the distinction between bundles and packages.
2005-07-07	Updated the list of conditions under which a main bundle might be NULL. Fixed typos.
	Changed title from *Bundles*.
	Added a link to *Framework Programming Guide*.
2005-03-03	Corrected typos.
2004-08-31	Added notes about the correct capitalization of files and directories in a bundle.
	Added section about unloadable bundles.
2004-03-26	Merged content from Mac OS X Bundles into this document.
	Update content to reflect both Cocoa and Core Foundation interfaces.
	Added guidelines for using bundles.
	Removed information about the anatomy of framework bundles. That information is now covered in *Framework Programming Guide*.
	Fixed minor bugs.
2003-10-22	Updated links for *Internationalizing Your Software*.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.