Runtime Configuration Guidelines

Mac OS X

2008-07-08

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

Simultaneously published in the United States and Canada.

Contents

Tables and Listings

Introduction

Dynamic configuration is a convenient way to adjust the properties of your executable without recompiling your code. Rather than relying on hardcoded information, your application implements slightly different behaviors based on external settings. There are several ways to record these settings, ranging from user preferences to property lists stored with your bundle.

Bundles use property lists extensively to store information about the bundle and its contents. Mac OS X and iPhone OS use the information in these property lists to determine an application properties such as its icon and whether to show the status bar (for iPhone applications).

You should read this document to learn about the properties you can use to configure application behavior and specify how Mac OS X or iPhone OS handle your application.

Organization of This Document

This document contains the following articles:

- "Information Property List Files" (page 11) discusses information property list files and how you use them to configure your bundled application.
- "Property List Key Reference" (page 17) provides a reference for the keys that can go into an information property list file.
- "The Preferences System" (page 39) discusses the role and scope of user preferences and describes the use of the `defaults` tool for accessing preferences.
- "Environment Variables" (page 43) discusses the role of environment variables in configuring applications. This section also covers some of the ways you can establish environment variables for a given user session or process.
- "Guidelines for Configuring Applications" (page 45) lists the required and recommended configuration options for applications. This article also describes additional ways to configure both bundled and non-bundled applications.

Information Property List Files

An information property list file contains essential configuration information for a bundled executable. Most bundles have at least one file of this type (usually named `Info.plist`) containing most of the bundle's configuration information. Variants of this file may also be present depending on the platforms and languages supported by the bundle.

The contents of an Information property list file are organized hierarchically, with each node in the hierarchy containing an entity such as an array, dictionary, string, or other scalar type. Information property list files are typically saved as XML files in a text file format that uses the Unicode UTF-8 encoding. Although you may encounter versions of these files encoded using ASCII text and binary formats, the XML format is recommended for any new files you save. Regardless of the format, you can open property list files saved in ASCII, XML, and binary formats using the Property List Editor application.

Important: In the sections that follow, pay attention to the capitalization of files and directories that reside inside a bundle. `CFBundle` and `NSBundle` consider case when searching for resources inside a bundle directory. Case mismatches could prevent you from finding your resources at runtime.

Standard Information Property List Files

By convention, a bundle's information property list file has the name `Info.plist`. This file resides in the bundle's `Contents` directory and contains configuration information for all supported platforms. However, if you want to configure your application differently on different platforms, you can include platform-specific versions of your information property list file. These files reside in your `Contents` directory. The name of each file is of the form "`Info-<platform>.plist`." The currently supported platforms are `macos` and `macosclassic`, thus you can define the following platform-specific information property list files:

- `Info-macos.plist` contains properties specific to a Mac OS X application.
- `Info-macosclassic.plist` contains properties specific to an application run in the Classic compatibility environment.

`NSBundle` and `CFBundle` load only one information property list file from your bundle, preferring the platform-specific version over the generic version. Thus, if you provide platform-specific information property list files for your bundle, make sure each of them contains all of the necessary keys to configure the application. Otherwise, if you split the required keys between an `Info-macos.plist` file and a `Info.plist` file, the keys in the `Info.plist` file are ignored.

You can create custom property lists as needed to store the values for application-specific configuration keys. If you create custom property lists, put them in your `Contents/Resources` directory with the rest of your application-specific resources. You can include your keys in your bundle's information property list file if you want all of your keys stored in one place.

Localizing Property List Values

If your information property list file contains values that might be displayed to the user, you should provide localized values for those properties. To deliver localized values, you create an `InfoPlist.strings` file in each language-specific resource directory of your bundle. Inside this strings file, you specify the key for each property you wish to localize along with the appropriate translated value.

At runtime, when your code retrieves the value of a property, the `NSBundle` and `CFBundle` routines check to see if an `InfoPlist.strings` file of the appropriate language exists and if it contains the requested key. If it does, the routines return the value from the strings file. If the key does not exist in any language files, the routines return the default version of the key from your bundle's `Info.plist` file.

For example, the TextEdit application has several keys that are displayed in the Finder and thus should be localized. Suppose your information property list file defines the following keys:

```
<key>CFBundleDisplayName</key>
<string>TextEdit</string>
<key>NSHumanReadableCopyright</key>
<string>Copyright ¬© 1995-2002, Apple Computer, Inc.,All Rights Reserved.
</string>
```

The French localization for TextEdit then includes the following strings in the `InfoPlist.strings` file of its `Contents/Resources/French.lproj` directory:

```
CFBundleDisplayName = "TextEdit";
NSHumanReadableCopyright = "Copyright © 1995-2002 Apple Computer Inc.\nTous
droits réservés.";
```

See *Bundle Programming Guide* for more about localizing bundles.

Creating and Editing Property Lists

The simplest way to create a new property list file or edit an existing file is to use the Property List Editor application. This application comes with Xcode and is installed in the `<Xcode>/Applications/Utilities` directory (where `<Xcode>` is the root directory of your Xcode installation). When you launch the application, it automatically opens a new, empty property list for you to edit.

To create a basic property list, add a root element and one or more children. The root element is always a dictionary. The children of the root can be any of the simple types (String, Number, Boolean, Date, or Data) or one of the collection types (Array or Dictionary).

Once you save the property list, you can edit it either with the Property List Editor or with any text editor that supports UTF-8 encoding. Although you can edit the file with an XML editor, make sure your editor saves the file in the UTF-8 encoding.

Important: If you create an information property list file by hand, remember that the first letter of the filename must be capitalized. `CFBundle` and `NSBundle` consider case when searching for resource files (including the `Info.plist` file) in a bundle.

An Example of an Information Property List File

Listing 1 shows the `Info.plist` file of the Sketch sample application from Mac OS X v10.5. Sketch is a Cocoa application so it includes the `NSMainNibFile` and `NSPrincipalClass` keys to identify the location of the primary application resources. Sketch registers several supported document types (using both UTIs and the file extensions) to make it easier to work with those types of files. UTIs are supported in Mac OS X v10.5 and later and are the preferred way to register support for file types. For more information about the meaning of each key, see "Property List Key Reference" (page 17). (Note, many of the comments in this XML file were removed or edited for brevity.)

Listing 1 The Info.plist file for the Sketch demo application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>sketch2</string>
            </array>
            <key>CFBundleTypeIconFile</key>
            <string>Draw2File</string>
            <key>CFBundleTypeName</key>
            <string>Apple Sketch document</string>
            <key>CFBundleTypeRole</key>
            <string>Editor</string>
            <!-- The LSItemContentTypes key is ignored in Mac OS X v10.4 because it's
introduced in 10.5. -->
            <key>LSItemContentTypes</key>
            <array>
                <string>com.apple.sketch2</string>
            </array>
            <key>NSDocumentClass</key>
            <string>SKTDrawDocument</string>
            <!-- This key is ignored in Mac OS X 10.5 because of the presence of the
LSItemContentTypes key. -->
            <key>NSExportableAs</key>
            <array>
                <string>NSPDFPboardType</string>
                <string>NSTIFFPboardType</string>
            </array>
            <!-- The NSExportableTypes key is ignored in Mac OS X 10.4 -->
            <key>NSExportableTypes</key>
```

```
            <array>
                <string>com.adobe.pdf</string>
                <string>public.tiff</string>
            </array>

        </dict>

        <!-- These keys are for compatibility with Mac OS X v10.4. -->
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>pdf</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>NSPDFPboardType</string>
            <key>CFBundleTypeRole</key>
            <string>None</string>
        </dict>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>tiff</string>
                <string>tif</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>NSTIFFPboardType</string>
            <key>CFBundleTypeRole</key>
            <string>None</string>
        </dict>

    </array>
    <key>CFBundleExecutable</key>
    <string>Sketch</string>
    <key>CFBundleIconFile</key>
    <string>Draw2App</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.CocoaExamples.Sketch</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleShortVersionString</key>
    <string>2.0</string>
    <key>CFBundleSignature</key>
    <string>sktc</string>
    <key>CFBundleVersion</key>
    <string>46.1</string>
    <key>NSAppleScriptEnabled</key>
    <string>YES</string>
    <key>NSMainNibFile</key>
    <string>Draw2</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
    <key>OSAScriptingDefinition</key>
    <string>Sketch.sdef</string>

    <!-- UTI keys are for ignored in Mac OS X v10.4 but used in Mac OS X v10.5. -->
    <key>UTExportedTypeDeclarations</key>
```

```
    <array>
        <dict>
            <key>UTTypeDescription</key>
            <string>Apple Sketch document</string>
            <key>UTTypeConformsTo</key>
            <array>
                <string>public.data</string>
            </array>
            <key>UTTypeIconFile</key>
            <string>Draw2File</string>
            <key>UTTypeIdentifier</key>
            <string>com.apple.sketch2</string>
            <key>UTTypeTagSpecification</key>
            <dict>
                <key>public.filename-extension</key>
                <array>
                    <string>sketch2</string>
                </array>
            </dict>
        </dict>
    </array>

</dict>
</plist>
```

Putting Info.plist Files in a Flat Executable

Even if your program does not use the bundle structure, it should still include an information property-list file to identify key pieces of information to the system. For unbundled CFM executables, you can place the contents of the program's `Info.plist` file in a `'plst'` resource. For unbundled Mach-O executables, you can create an `__info_plist` section in the executable's `__TEXT` segment and put the contents of your information property-list file there. To create an `__info_plist` section, you would create an `Info.plist` file as you would for a bundled program and then add the following linker options to your makefile or Xcode project:

```
-sectcreate __TEXT __info_plist Info.plist
```

To retrieve the `Info.plist` information, your unbundled program can use many of the `CFBundle` functions for accessing bundle properties. Although your program is not bundled, you can still get the "main bundle" and pass that object to any functions you call.

Property List Key Reference

The following sections contain detailed information about the usage and behavior of property-list keys available for use in an information property list file.

Key Summary

Table 1 contains an alphabetical list of the keys you can use in an information property list file, along with a brief description and the platforms to which they apply (Mac OS X or iPhone). For detailed descriptions of these keys, see "Key Descriptions " (page 22).

Table 1 Summary of standard keys

Key	Summary	Platforms
APInstallerURL	A URL-based path to the files you want to install. See "APInstallerURL" (page 22) for details.	Mac OS X
APFiles	An array of dictionaries describing the files or directories that can be installed. See "APFiles" (page 22) for details.	Mac OS X
ATSApplicationFontsPath	The path to a single font file or directory of font files in the bundle's `Resources` directory. See "ATSApplicationFontsPath" (page 23) for details.	Mac OS X
CFAppleHelpAnchor	The bundle's initial HTML help file. See "CFAppleHelpAnchor" (page 23) for details.	Mac OS X
CFBundleAllowMixedLocalizations	Used by Foundation tools to retrieve localized resources from frameworks. See "CFBundleAllowMixedLocalizations" (page 23) for details.	Mac OS X, iPhone OS
CFBundleDevelopmentRegion	The native region for the bundle. Usually corresponds to the native language of the author. See "CFBundleDevelopmentRegion" (page 23) for details.	Mac OS X, iPhone OS
CFBundleDisplayName	The actual name of the bundle. See "CFBundleDisplayName" (page 23) for details.	Mac OS X, iPhone OS
CFBundleDocumentTypes	An array of dictionaries describing the document types supported by the bundle. See "CFBundleDocumentTypes" (page 24) for details.	Mac OS X

Key	Summary	Platforms
CFBundleExecutable	Name of the bundle executable file. See "CFBundleExecutable" (page 26) for details.	Mac OS X, iPhone OS
CFBundleGetInfoString	Brief description of the bundle. See "CFBundleGetInfoString" (page 26) for details.	Mac OS X
CFBundleHelpBookFolder	The name of the folder containing the bundle's help files. See "CFBundleHelpBookFolder" (page 26) for details.	Mac OS X
CFBundleHelpBookName	The name of the help file to display when Help Viewer is launched for the bundle. See "CFBundleHelpBookName" (page 26) for details.	Mac OS X
CFBundleIconFile	File name for icon image file. See "CFBundleIconFile" (page 26) for details.	Mac OS X, iPhone OS
CFBundleIdentifier	An identifier string that specifies the application type of the bundle. This string should be a uniform type identifier (UTI), for example `com.mycompany.MyApp`. See "CFBundleIdentifier" (page 27) for details.	Mac OS X, iPhone OS
CFBundleInfoDictionaryVersion	Version information for the `Info.plist` format. See "CFBundleInfoDictionaryVersion" (page 27) for details.	Mac OS X, iPhone OS
CFBundleLocalizations	Contains localization information for an application that handles its own localized resources. See "CFBundleLocalizations" (page 27) for details.	Mac OS X, iPhone OS
CFBundleName	The short display name of the bundle. See "CFBundleName" (page 27) for details.	Mac OS X, iPhone OS
CFBundlePackageType	The four-letter code identifying the bundle type. See "CFBundlePackageType" (page 27) for details.	Mac OS X
CFBundleShortVersionString	The release-version-number string for the bundle. See "CFBundleShortVersionString" (page 28) for details.	Mac OS X, iPhone OS
CFBundleSignature	The four-letter code identifying the bundle creator. See "CFBundleSignature" (page 28) for details.	Mac OS X
CFBundleURLTypes	An array of dictionaries describing the URL schemes supported by the bundle. See "CFBundleURLTypes" (page 28) for details.	Mac OS X, iPhone OS
CFBundleVersion	The build-version-number string for the bundle. See "CFBundleVersion" (page 29) for details.	Mac OS X, iPhone OS

Key	Summary	Platforms
CFPlugInDynamicRegistration	If YES, register the plug-in dynamically; otherwise, register it statically. See "CFPlugInDynamicRegistration" (page 29) for details.	Mac OS X
CFPlugInDynamicRegistrationFunction	The name of the custom, dynamic registration function. See "CFPlugInDynamicRegisterFunction" (page 29) for details.	Mac OS X
CFPlugInFactories	For static registration, this dictionary contains a list of UUIDs with matching function names. See "CFPlugInFactories" (page 29) for details.	Mac OS X
CFPlugInTypes	For static registration, the list of UUIDs "CFPlugInTypes" (page 29) for details.	Mac OS X
CFPlugInUnloadFunction	The name of the custom function to call when it's time to unload the plug-in code from memory. See "CFPlugInUnloadFunction" (page 30) for details.	Mac OS X
CSResourcesFileMapped	If true, Core Services routines map the bundle's resource files into memory instead of reading them. See "CSResourcesFileMapped" (page 30) for details.	Mac OS X
LSArchitecturePriority	Contains an array of strings identifying the supported code architectures and their preferred execution priority. See "LSArchitecturePriority" (page 30) for details.	Mac OS X
LSBackgroundOnly	Specifies whether the application runs only in the background. (Mach-O applications only). See "LSBackgroundOnly" (page 30) for details.	Mac OS X
LSEnvironment	Contains a list of key/value pairs, representing environment variables and their values. See "LSEnvironment" (page 31) for details.	Mac OS X
LSFileQuarantineEnabled	Specifies whether the files this application creates are quarantined by default. See "LSFileQuarantineEnabled" (page 31).	Mac OS X
LSGetAppDiedEvents	Specifies whether the application is notified when a child process dies. See "LSGetAppDiedEvents" (page 31) for details.	Mac OS X
LSHasLocalizedDisplayName	Specifies whether the application supports a localized display name. See "LSHasLocalizedDisplayName" (page 31) for details.	Mac OS X

Key	Summary	Platforms
LSMinimumSystemVersion	Specifies the minimum version of Mac OS X required for the application to run. See "LSMinimumSystemVersion" (page 31) for details.	Mac OS X
LSMinimumSystemVersionByArchitecture	Specifies the minimum version of Mac OS X required to run a given platform architecure. See "LSMinimumSystemVersionByArchitecture" (page 32) for details.	Mac OS X
LSMultipleInstancesProhibited	Specifies whether one user or multiple users can launch an application simultaneously. See "LSMultipleInstancesProhibited" (page 32) for details.	Mac OS X
LSRequiresIPhoneOS	Specifies whether the application is an iPhone application. See "LSRequiresIPhoneOS" (page 32) for details.	Mac OS X, iPhone OS
LSRequiresNativeExecution	Specifies whether the application must run natively on Intel-based Macintosh computers, as opposed to under Rosetta emulation. See "LSRequiresNativeExecution" (page 32) for details.	Mac OS X
LSUIElement	Specifies whether the application is an agent application, that is, an application that should not appear in the Dock or Force Quit window. See "LSUIElement" (page 33) for details.	Mac OS X
LSUIPresentationMode	Sets the visibility of system UI elements when the application launches. See "LSUIPresentationMode" (page 33) for details.	Mac OS X
LSVisibleInClassic	Specifies whether an agent application or background-only application is visible to other applications in the Classic environment. See "LSVisibleInClassic" (page 33) for details.	Mac OS X
NSAppleScriptEnabled	Specifies whether AppleScript is enabled. See "NSAppleScriptEnabled" (page 34) for details.	Mac OS X
NSHumanReadableCopyright	Specifies the copyright notice for the bundle. See "NSHumanReadableCopyright" (page 34) for details.	Mac OS X
NSJavaNeeded	Specifies whether the program requires a running Java VM. See "NSJavaNeeded" (page 34) for details.	Mac OS X
NSJavaPath	An array of paths to classes whose components are preceded by `NSJavaRoot`. See "NSJavaPath" (page 34) for details.	Mac OS X

Key	Summary	Platforms
NSJavaRoot	The root directory containing the java classes. See "NSJavaRoot" (page 34) for details.	Mac OS X
NSMainNibFile	The name of an application's main nib file. See "NSMainNibFile" (page 34) for details.	Mac OS X, iPhone OS
NSPersistentStoreTypeKey	The type of Core Data persistent store associated with a persistent document type. See "NSPersistentStoreTypeKey" (page 35) for details.	Mac OS X
NSPrefPaneIconFile	The name of an image file resource used to represent a preference pane in the System Preferences application. See "NSPrefPaneIconFile" (page 35) for details.	Mac OS X
NSPrefPaneIconLabel	The name of a preference pane displayed beneath the preference pane icon in the System Preferences application. See "NSPrefPaneIconLabel" (page 35) for details.	Mac OS X
NSPrincipalClass	The name of the bundle's main class. See "NSPrincipalClass" (page 35) for details.	Mac OS X
NSServices	An array of dictionaries specifying the services provided by an application. See "NSServices" (page 35) for details.	Mac OS X
UIInterfaceOrientation	Specifies the initial orientation of the application's user interface. See "UIInterfaceOrientation" (page 36) for details.	iPhone OS
UIPrerenderedIcon	Specifies whether the iPhone OS applies sheen and bevel effects to the application icon. See "UIPrerenderedIcon" (page 36) for details.	iPhone OS
UIRequiresPersistentWiFi	Specifies whether this application requires a Wi-Fi connection. See "UIRequiresPersistentWiFi" (page 37) for details.	iPhone OS
UIStatusBarHidden	Specifies whether the status bar is initially hidden when the application launches. See "UIStatusBarHidden" (page 37) for details.	iPhone OS
UIStatusBarStyle	Specifies the style of the status bar as the application launches. See "UIStatusBarStyle" (page 37) for details.	iPhone OS
UTExportedTypeDeclarations	An array of dictionaries specifying the UTI-based types supported (and owned) by the application. See "UTExportedTypeDeclarations" (page 37) for details.	Mac OS X

Key	Summary	Platforms
UTImportedTypeDeclarations	An array of dictionaries specifying the UTI-based types supported (but not owned) by the application. See "UTImportedTypeDeclarations" (page 38) for details.	Mac OS X

Key Descriptions

Mac OS X provides a set of core keys for specifying information about a bundle. Some of these keys are given default values by the Xcode application depending on the type of project you create.

APInstallerURL

`APInstallerURL` (`String`) identifies the base path to the files you want to install. You must specify this path using the form `file://localhost/path/`. All installed files must reside within this directory.

APFiles

`APFiles` (`Array`) specifies a file or directory you want to install. You specify this key as a dictionary, the contents of which contains information about the file or directory you want to install. To specify multiple items, nest the `APFiles` key inside itself to specify files inside of a directory. Table 2 lists the keys for specifying information about a single file or directory.

Table 2 Keys for APFiles dictionary

Key	Type	Description
APFileDescriptionKey	`String`	A short description of the item to display in the Finder's Info window
APDisplayedAsContainer	`String`	If "Yes" the item is shown with a folder icon in the Info panel; otherwise, it is shown with a document icon
APFileDestinationPath	`String`	Where to install the component as a path relative to the application bundle
APFileName	`String`	The name of the file or directory
APFileSourcePath	`String`	The path to the component in the application package relative to the `APInstallerURL` path.
APInstallAction	`String`	The action to take with the component: "Copy" or "Open"

ATSApplicationFontsPath

`ATSApplicationFontsPath` (`String`) identifies the location of a font file or directory of fonts in the bundle's `Resources` directory. If present, Mac OS X activates the fonts at the specified path for use by the bundled application. The fonts are activated only for the bundled application and not for the system as a whole. The path itself should be specified as a relative directory of the bundle's Resources directory. For example, if a directory of fonts was at the path `/Applications/MyApp.app/Contents/Resources/Stuff/MyFonts/`, you should specify the string `Stuff/MyFonts/` for the value of this key.

CFAppleHelpAnchor

`CFAppleHelpAnchor` (`String`) identifies the name of the bundle's initial HTML help file, minus the `.html` or `.htm` extension. This file must be located in the bundle's localized resource directories or, if the help is not localized, directly under the `Resources` directory.

CFBundleAllowMixedLocalizations

`CFBundleAllowMixedLocalizations` (`Boolean`) specifies whether the bundle supports the retrieval of localized strings from frameworks. This key is used primarily by Foundation tools that link to other system frameworks and want to retrieve localized resources from those frameworks.

CFBundleDevelopmentRegion

`CFBundleDevelopmentRegion` (`String`) specifies the native region for the bundle. This key contains a string value that usually corresponds to the native language of the person who wrote the bundle. The language specified by this value is used as the default language if a resource cannot be located for the user's preferred region or language.

CFBundleDisplayName

`CFBundleDisplayName` (`String`) specifies the display name of the bundle. If you support localized names for your bundle, include this key in both your information property list file and in the `InfoPlist.strings` files of your language subdirectories. If you localize this key, you should also include a localized version of the `CFBundleName` key.

If you do not intend to localize your bundle, do not include this key in your `Info.plist` file. Inclusion of this key does not affect the display of the bundle name but does incur a performance penalty to search for localized versions of this key.

Before displaying a localized name for your bundle, the Finder compares the value of this key against the actual name of your bundle in the file system. If the two names match, the Finder proceeds to display the localized name from the appropriate `InfoPlist.strings` file of your bundle. If the names do not match, the Finder displays the file-system name.

See *File System Overview* for more information on display names.

CFBundleDocumentTypes

`CFBundleDocumentTypes` (`DictionaryArray`) associates a document type with your application using a set of dictionaries. Each dictionary is called a type-definition dictionary and contains keys used to define the document type. Table 3 lists the keys that are supported in these dictionaries:

Table 3 Keys for type-definition dictionaries

Key	Type	Description
CFBundleTypeExtensions	`Array`	This key contains an array of strings. Each string contains a filename extension (minus the leading period) to map to this document type. To open documents with any extension, specify an extension with a single asterisk "*". (In Mac OS X v10.4, this key is ignored if the `LSItemContentTypes` key is present.) Deprecated in Mac OS X v10.5.
CFBundleTypeIconFile	`String`	This key specifies the name of the icon file to associate with this document type. If you omit the extension, the system looks for your file with the extension `.icns`.
CFBundleTypeMIMETypes	`Array`	Contains an array of strings. Each string contains the MIME type name you want to map to this document type. (In Mac OS X v10.4, this key is ignored if the `LSItemContentTypes` key is present.) Deprecated in Mac OS X v10.5.
CFBundleTypeName	`String`	This key contains the abstract name for the document type and is used to refer to the type. This key is required and can be localized by including it in an `InfoPlist.strings` files. This value is the main way to refer to a document type. If you are concerned about this key being unique, you should consider using a uniform type identifier (UTI) for this string instead. If the type is a common Clipboard type supported by the system, you can use one of the standard types listed in the `NSPasteboard` class description. See *Uniform Type Identifiers Overview* for details on UTIs.
CFBundleTypeOSTypes	`Array`	This key contains an array of strings. Each string contains a four-letter type code that maps to this document type. To open documents of any type, include four asterisk characters (`****`) as the type code. These codes are equivalent to the legacy type codes used by Mac OS 9. (In Mac OS X v10.4, this key is ignored if the `LSItemContentTypes` key is present.) Deprecated in Mac OS X v10.5.
CFBundleTypeRole	`String`	This key specifies the application's role with respect to the type. The value can be `Editor`, `Viewer`, `Shell`, or `None`. See "Document Configuration" (page 47) for descriptions of these values. This key is required.

Key	Type	Description
LSItemContentTypes	`Array`	This key contains an array of strings. Each string contains a UTI defining a supported file type. The UTI string must be spelled out explicitly, as opposed to using one of the constants defined by Launch Services. For example, to support PNG files, you would include the string "`public.png`" in the array. When using this key, also add the `NSExportableTypes` key with the appropriate entries. In Mac OS X v10.4 and later, this key (when present) takes precedence these type identifier keys: `CFBundleTypeExtensions`, `CFBundleTypeMIMETypes`, `CFBundleTypeOSTypes`.
LSHandlerRank	`String`	Determines how Launch Services ranks this application among the applications that declare themselves editors or viewers of files of this type. The possible values are: `Owner` (this application is the creator of files of this type), `Alternate` (this application is a secondary viewer of files of this type), `None` (this application must never be used to open files of this type, but it accepts drops of files of this type), `Default` (default; this application doesn't accept drops of files of this type). Launch Services uses the value of `LSHandlerRank` to determine the application to use to open files of this type. The order of precedence is: `Owner`, `Alternate`, `None`. This key is available in Mac OS X v10.5 and later.
LSTypeIsPackage	`Boolean`	Specifies whether the document is distributed as a bundle. If set to true, the bundle directory is treated as a file. (In Mac OS X v10.4 and later, this key is ignored if the `LSItemContentTypes` key is present.)
NSDocumentClass	`String`	This key specifies the name of the `NSDocument` subclass used to instantiate instances of this document. This key is used by Cocoa applications only.
NSExportableAs	`Array`	This key specifies an array strings. Each string contains the name of another document type, that is, the value of a `NSBundleTypeName` property. This value represents another data format to which this document can export its content. This key is used by Cocoa applications only. Deprecated in Mac OS X v10.5.
NSExportableTypes	`Array`	This key specifies an array strings. Each string contains the name of another document type, that is, the value of a `NSBundleTypeName` property. This value represents another data format to which this document can export its content. This key is used by Cocoa applications only.

When registering document types, you must specify at least one of the keys `LSItemContentTypes`, `CFBundleTypeExtensions`, `CFBundleTypeMIMETypes`, or `CFBundleTypeOSTypes` along with the appropriate data. If you do not specify at least one of these keys, no document types are bound to the type-name specifier. You may use all three keys when binding your document type, if you so choose. In Mac OS X v10.4 and later, if you specify the `LSItemContentTypes` key, the other keys are ignored.

CFBundleExecutable

`CFBundleExecutable` (`String`) identifies the name of the bundle's main executable file. For an application, this is the application executable. For a loadable bundle, it is the binary that will be loaded dynamically by the bundle. For a framework, it is the shared library for the framework. Xcode automatically adds this key to the information property list file of appropriate projects.

For frameworks, the value of this key is required to be the same as the framework name, minus the `.framework` extension. If the keys are not the same, the target system may incur some launch-performance penalties. for launch-performance reasons. The value should not include any extension on the name.

Important: You must include a valid `CFBundleExecutable` key in your bundle's information property list file. Mac OS X uses this key to locate the bundle's executable or shared library in cases where the user renames the application or bundle directory.

CFBundleGetInfoString

`CFBundleGetInfoString` (`String`) provides a brief description of the bundle. For an application bundle, this key provides a short description of the application or the current release that the build or release version number cannot convey; for example, the date of the release. This key can be localized.

CFBundleHelpBookFolder

`CFBundleHelpBookFolder` (`String`) identifies the folder containing the bundle's help files. Help is usually localized to a specific language, so the folder specified by this key represents the folder name inside the `.lproj` directory for the selected language.

CFBundleHelpBookName

`CFBundleHelpBookName` (`String`) identifies the main help page for your application. This key identifies the name of the Help page, which may not correspond to the name of the HTML file. The Help page name is specified in the `CONTENT` attribute of the help file's `META` tag.

CFBundleIconFile

`CFBundleIconFile` (`String`) identifies the file containing the icon for the bundle. The filename you specify does not need to include the extension, although it may. The Finder looks for the icon file in the `Resources` directory of the bundle.

If your bundle uses a custom icon, you must specify this property. If you do not specify this property, the Finder (and other applications) display your bundle with a default icon.

CFBundleIdentifier

`CFBundleIdentifier` (`String`) identifies the type of the bundle. This identifier should be a uniform type identifier (UTI) string, for example `com.mycompany.MyApp`. This key does not uniquely identify a specific bundle in the file system, as multiple copies of an application with the same or different version may exist. See *Uniform Type Identifiers Overview* for details on UTIs.

The preferences system uses this string to identify the application for which a given preference applies. Launch Services uses the bundle identifier to locate an application capable of opening a particular file, using the first application it finds with the given identifier.

CFBundleInfoDictionaryVersion

`CFBundleInfoDictionaryVersion` (`String`) identifies the current version of the property list structure. This key exists to support future versioning of the information property list file format. Xcode generates this key automatically when you build a bundle and you should not change it manually. The value for this key is currently 6.0.

CFBundleLocalizations

`CFBundleLocalizations` (`Array`) identifies the localizations handled manually by your application. If your executable is unbundled or does not use the existing bundle localization mechanism, you can include this key to specify the localizations your application does handle.

Each entry in this property's array is a string identifying the language name or ISO language designator of the supported localization. See "Language Designations" in *Internationalization Programming Topics* in Internationalization Documentation for information on how to specify language designators.

CFBundleName

`CFBundleName` (`String`) identifies the short name of the bundle. This name should be less than 16 characters long and be suitable for displaying in the menu bar and the application's Info window. You can include this key in the `InfoPlist.strings` file of an appropriate `.lproj` subdirectory to provide localized values for it. If you localize this key, you should also include the key "CFBundleDisplayName" (page 23).

CFBundlePackageType

`CFBundlePackageType` (`String`) identifies the type of the bundle and is analogous to the Mac OS 9 file type code. The value for this key consists of a four-letter code. The type code for applications is `APPL`; for frameworks, it is `FMWK`; for loadable bundles, it is `BNDL`. For loadable bundles, you can also choose a type code that is more specific than `BNDL` if you want.

CFBundleShortVersionString

`CFBundleShortVersionString` (`String`) specifies the **release version number** of the bundle, which identifies a released iteration of the application. The release version number is a string comprised of three period-separated integers. The first integer represents major revisions to the application, such as revisions that implement new features or major changes. The second integer denotes revisions that implement less prominent features. The third integer represents maintenance releases.

The value for this key differs from the value for "CFBundleVersion" (page 29), which identifies an iteration (released or unreleased) of the application.

CFBundleSignature

`CFBundleSignature String`) identifies the creator of the bundle and is analogous to the Mac OS 9 file creator code. The value for this key is a string containing a four-letter code that is specific to the bundle. For example, the signature for the TextEdit application is `ttxt`.

CFBundleURLTypes

`CFBundleURLTypes DictionaryArray` describes the URL schemes (`http`, `ftp`, and so on) supported by the application. The purpose of this key is similar to that of "CFBundleDocumentTypes" (page 24), but it describes URL schemes instead of document types. Each dictionary entry corresponds to a single URL scheme. Table 4 lists the keys to use in each dictionary entry.

Table 4 Keys for CFBundleURLTypes dictionaries

Key	Type	Description
CFBundleTypeRole	`String`	This key specifies the application's role with respect to the URL type. The value can be `Editor`, `Viewer`, `Shell`, or `None`. See "Document Configuration" (page 47) for descriptions of these values. This key is required.
CFBundleURLIconFile	`String`	This key contains the name of the icon image file (minus the extension) to be used for this URL type.
CFBundleURLName	`String`	This key contains the abstract name for this URL type. This is the main way to refer to a particular type. To ensure uniqueness, it is recommended that you use a Java-package style identifier. This name is also used as a key in the `InfoPlist.strings` file to provide the human-readable version of the type name.
CFBundleURLSchemes	`Array`	This key contains an array of strings, each of which identifies a URL scheme handled by this type. Examples of URL schemes include `http`, `ftp`, and so on.

CFBundleVersion

`CFBundleVersion` (`String`) specifies the **build version number** of the bundle, which identifies an iteration (released or unreleased) of the bundle. This is a monotonically increased string, comprised of one or more period-separated integers. This key is not localizable.

CFPlugInDynamicRegistration

`CFPlugInDynamicRegistration` (`String`) specifies whether how host loads this plug-in. If the value is `YES`, the host attempts to load this plug-in using its dynamic registration function. If the value is `NO`, the host uses the static registration information included in the "CFPlugInFactories" (page 29), and "CFPlugInTypes" (page 29) keys.

See "Plug-in Registration" in *Plug-ins* for information about registering plug-ins.

CFPlugInDynamicRegisterFunction

`CFPlugInDynamicRegisterFunction` (`String`) identifies the function to use when dynamically registering a plug-in. Specify this key if you want to specify one of your own functions instead of implement the default `CFPlugInDynamicRegister` function.

See "Plug-in Registration" in *Plug-ins* for information about registering plug-ins.

CFPlugInFactories

`CFPlugInFactories` (`Dictionary`) is used for static plug-in registration. It contains a dictionary identifying the interfaces supported by the plug-in. Each key in the dictionary is a universally unique ID (UUID) representing the supported interface. The value for the key is a string with the name of the plug-in factory function to call.

See "Plug-in Registration" in *Plug-ins* for information about registering plug-ins.

CFPlugInTypes

`CFPlugInTypes` (`Dictionary`) is used for static plug-in registration. It contains a dictionary identifying one or more groups of interfaces supported by the plug-in. Each key in the dictionary is a universally unique ID (UUID) representing the group of interfaces. The value for the key is an array of strings, each of which contains the UUID for a specific interface in the group. The UUIDs in the array corresponds to entries in the "CFPlugInFactories" (page 29) dictionary.

See "Plug-in Registration" in *Plug-ins* for information about registering plug-ins.

CFPlugInUnloadFunction

`CFPlugInUnloadFunction` `String`) specifies the name of the function to call when it is time to unload the plug-in code from memory. This function gives the plug-in an opportunity to clean up any data structures it allocated.

See "Plug-in Registration" in *Plug-ins* for information about registering plug-ins.

CSResourcesFileMapped

`CSResourcesFileMapped` `Boolean`) specifies whether to map this application's resource files into memory. Otherwise, they are read into memory normally. File mapping can improve performance in situations where you are frequently accessing a small number of resources. However, resources are mapped into memory read-only and cannot be modified.

LSArchitecturePriority

`LSArchitecturePriority` (`StringArray`) identifies the architectures this application supports. The order of the strings in this array dictate the preferred execution priority for the architectures. The possible strings for this array are listed in Table 5.

Table 5 Execution architecture identifiers

String	Description
`i386`	The 32-bit Intel architecture.
`ppc`	The 32-bit PowerPC architecture.
`x86_64`	The 64-bit Intel architecture.
`ppc64`	The 64-bit PowerPC architecture.

if a PowerPC architecture appears before either of the Intel architectures, Mac OS X runs the executable under Rosetta emulation on Intel-based Macintosh computers regardless by default. To force Mac OS X to use the current platform's native architecture, include the "LSRequiresNativeExecution" (page 32) key in your information property list.

LSBackgroundOnly

`LSBackgroundOnly` (`Boolean`) specifies whether this application runs only in the background. If this key exists and is set to "1", Launch Services runs the application in the background only. You can use this key to create faceless background applications. You should also use this key if your application uses higher-level frameworks that connect to the window server, but are not intended to be visible to users. Background applications must be compiled as Mach-O executables. This option is not available for CFM applications.

LSEnvironment

`LSEnvironment` `Dictionary`) defines environment variables to be set before launching this application. The names of the environment variables are the keys of the dictionary, with the values being the corresponding environment variable value. Both keys and values must be strings.

These environment variables are set only for applications launched through Launch Services. If you run your executable directly from the command line, these environment variables are not set.

LSFileQuarantineEnabled

`LSFileQuarantineEnabled` (`Boolean`) specifies whether files this application creates are quarantined by default.

Value	Description
`true`	Files created by this application are quarantined by default.
`false`	(Default) Files created by this application are not quarantined by default.

See

This key is available in Mac OS X v10.5 and later.

LSGetAppDiedEvents

`LSGetAppDiedEvents` (`Boolean`) indicates whether the operation system notifies this application when when one of its child process terminates. If you set the value of this key to `YES`, the system sends your application an `kAEApplicationDied` Apple event for each child process as it terminates.

LSHasLocalizedDisplayName

`LSHasLocalizedDisplayName` (`String`) specifies whether the Finder displays the name of this application as a localized string. When set to "1", the Finder displays the name of your application as a localized string. Include this key only if you include localized versions of the key "CFBundleDisplayName" (page 23) in your language-specific `InfoPList.strings` files.

Including this key significantly improves the performance of localized filename display. If your bundle supports localized display names, you should include this key in your information property list file.

LSMinimumSystemVersion

`LSMinimumSystemVersion` (`String`) indicates the minimum version of Mac OS X required for this application to run. This string is usually of the form *n.n.n* where *n* is a number. The first number is the major version number of the system. The second and third numbers are minor revision numbers. For example, to support Mac OS X v10.1 and later, you would set the value of this key to "10.1.5".

If the minimum system version is not available, Mac OS X tries to display an alert panel notifying the user of that fact, although it may not always be able to do so.

LSMinimumSystemVersionByArchitecture

`LSMinimumSystemVersionByArchitecture` (`Dictionary`) specifies the earliest Mac OS X version for a set of architectures. This key contains a dictionary of key-value pairs. Each key corresponds to one of the architectures associated with the "LSExecutableArchitectures" (page 30) key. The value for each key is the minimum version of Mac OS X required for the application to run under that architecture. This string is usually of the form *n.n.n* where *n* is a number. The first number is the major version number of the system. The second and third numbers are minor revision numbers. For example, to support Mac OS X v10.4.9 and later, you would set the value of this key to "10.4.9".

If the current system version is less than the required minimum version, Launch Services does not attempt to use the corresponding architecture. This key applies only to the selection of an execution architecture and can be used in conjunction with the "LSMinimumSystemVersion" (page 31) key, which specifies the overall minimum system version requirement for the application.

LSMultipleInstancesProhibited

`LSMultipleInstancesProhibited` (`Boolean`) indicates whether an application is prohibited from running simultaneously in multiple user sessions. If true, the application runs in only one user session at a time. You can use this key to prevent resource conflicts that might arise by sharing an application across multiple user sessions. For example, you might want to prevent users from accessing a custom USB device when it is already in use by a different user.

Launch Services returns an appropriate error code if the target application cannot be launched. If a user in another session is running the application, Launch Services returns a `kLSMultipleSessionsNotSupportedErr` error. If you attempt to launch a separate instance of an application in the current session, it returns `kLSMultipleInstancesProhibitedErr`.

LSRequiresIPhoneOS

`LSRequiresIPhoneOS` (`Boolean`) specifies whether the application can run only on iPhone OS. If this key is set to `YES`, Launch Services allows the application to launch only when the host platform is iPhone OS.

LSRequiresNativeExecution

`LSRequiresNativeExecution` (`Boolean`) specifies whether to launch the application using the subbinary for the current architecture. If this key is set to `YES`, Launch Services always runs the application using the binary compiled for the current architecture. You can use this key to prevent a universal binary from being run under Rosetta emulation on an Intel-based Macintosh computer. For more information about configuring the execution architectures, see "LSExecutableArchitectures" (page 30).

LSUIElement

`LSUIElement` (`String`). If this key is set to "1", Launch Services runs the application as an agent application. Agent applications do not appear in the Dock or in the Force Quit window. Although they typically run as background applications, they can come to the foreground to present a user interface if desired. A click on a window belonging to an agent application brings that application forward to handle events.

The Dock and loginwindow are two applications that run as agent applications.

LSUIPresentationMode

`LSUIPresentationMode` (`Number`) identifies the initial user-interface mode for the application. You would use this in applications that may need to take over portions of the screen that contain UI elements such as the Dock and menu bar. Most modes affect only UI elements that appear in the content area of the screen, that is, the area of the screen that does not include the menu bar. However, you can request that all UI elements be hidden as well.

This key is applicable to both Carbon and Cocoa applications and can be one of the following values:

Value	Description
0	Normal mode. In this mode, all standard system UI elements are visible. This is the default value.
1	Content suppressed mode. In this mode, system UI elements in the content area of the screen are hidden. UI elements may show themselves automatically in response to mouse movements or other user activity. For example, the Dock may show itself when the mouse moves into the Dock's auto-show region.
2	Content hidden mode. In this mode, system UI elements in the content area of the screen are hidden and do not automatically show themselves in response to mouse movements or user activity.
3	All hidden mode. In this mode, all UI elements are hidden, including the menu bar. Elements do not automatically show themselves in response to mouse movements or user activity.
4	All suppressed mode. In this mode, all UI elements are hidden, including the menu bar. UI elements may show themselves automatically in response to mouse movements or other user activity. This option is available only in Mac OS X 10.3 and later.

LSVisibleInClassic

`LSVisibleInClassic` (`String`). If this key is set to "1", any agent applications or background-only applications with this key appears as background-only processes to the Classic environment. Agent applications and background-only applications without this key do not appear as running processes to Classic at all. Unless your process needs to communicate explicitly with a Classic application, you do not need to include this key.

NSAppleScriptEnabled

`NSAppleScriptEnabled` (`Boolean` or `String`). This key identifies whether the application is scriptable. Set the value of this key to `true` (when typed as `Boolean`) or "YES" (when typed as `String`) if your application supports AppleScript.

NSHumanReadableCopyright

`NSHumanReadableCopyright` (`String`). This key contains a string with the copyright notice for the bundle; for example, `© 2008, My Company`. You can load this string and display it in an About dialog box. This key can be localized by including it in your `InfoPlist.strings` files. This key replaces the obsolete `CFBundleGetInfoString` key.

NSJavaNeeded

`NSJavaNeeded` (`Boolean` or `String`). This key specifies whether the Java VM must be loaded and started up prior to executing the bundle code. This key is required only for Cocoa Java applications to tell the system to launch the Java environment. If you are writing a pure Java application, do not include this key.

You can also specify a string type with the value "YES" instead of a Boolean value if desired.

Deprecated in Mac OS X v10.5.

NSJavaPath

`NSJavaPath` (`Array`). This key contains an array of paths. Each path points to a Java class. The path can be either an absolute path or a relative path from the location specified by the key "NSJavaRoot" (page 34). The development environment (or, specifically, its jamfiles) automatically maintains the values in the array.

Deprecated in Mac OS X v10.5.

NSJavaRoot

`NSJavaRoot` (`String`). This key contains a string identifying a directory. This directory represents the root directory of the application's Java class files.

NSMainNibFile

`NSMainNibFile` (`String`). This key contains a string with the name of the application's main nib file (minus the `.nib` extension). A nib file is an Interface Builder archive containing the description of a user interface along with any connections between the objects of that interface. The main nib file is automatically loaded when an application is launched.

NSPersistentStoreTypeKey

`NSPersistentStoreTypeKey` (`String`). This key contains a string that specifies the type of Core Data persistent store associated with a document type (see "CFBundleDocumentTypes" (page 24)). See `NSPersistentStoreCoordinator_Store_Types` for possible values.

NSPrefPaneIconFile

`NSPrefPaneIconFile` (`String`). This key contains a string with the name of an image file (including extension) containing the preference pane's icon. This key should only be used by preference pane bundles. The image file should contain an icon 32 by 32 pixels in size. If this key is omitted, the System Preferences application looks for the image file using the `CFBundleIconFile` key instead.

NSPrefPaneIconLabel

`NSPrefPaneIconLabel` (`String`). This key contains a string with the name of a preference pane. This string is displayed below the preference pane's icon in the System Preferences application. You can split long names onto two lines by including a newline character ('\n') in the string. If this key is omitted, the System Preferences application gets the name from the `CFBundleName` key.

This key can be localized and included in the `InfoPlist.strings` files of a bundle.

NSPrincipalClass

`NSPrincipalClass` (`String`). This key contains a string with the name of a bundle's principal class. This key is used to identify the entry point for dynamically loaded code, such as plug-ins and other dynamically-loaded bundles. The principal class of a bundle typically controls all other classes in the bundle and mediates between those classes and any classes outside the bundle. The class identified by this value can be retrieved using the `principalClass` method of `NSBundle`. For Cocoa applications, the value for this key is `NSApplication` by default.

NSServices

`NSServices` (`Array`). This key contains an array of dictionaries specifying the services provided by the application. Table 6 lists the keys for specifying a service:

Table 6 Keys for NSServices dictionaries

Key	Type	Description
NSPortName	`String`	This key specifies the name of the port your application monitors for incoming service requests. Its value depends on how the service provider application is registered. In most cases, this is the application name. For more information, see *System Services*.

Key	Type	Description
NSMessage	`String`	This key specifies the name of the instance method to invoke for the service. In Objective-C, the instance method must be of the form `messageName: userData:error:`. In Java, the instance method must be of the form `messageName(NSPasteBoard,String)`.
NSSendTypes	`Array`	This key specifies an array of data type names that can be read by the service. The `NSPasteboard` class description lists several common data types. You must include this key, the `NSReturnTypes` key, or both.
NSReturnTypes	`Array`	This key specifies an array of data type names that can be returned by the service. The `NSPasteboard` class description lists several common data types. You must include this key, the `NSSendTypes` key, or both.
NSMenuItem	`Dictionary`	This key contains a dictionary that specifies the text to add to the Services menu. The only key in the dictionary is called `default` and its value is the menu item text. This value must be unique. You can use a slash character "`/`" to specify a submenu. For example, `Mail/Send` would appear in the Services menu as a menu named Mail with an item named Send.
NSKeyEquivalent	`Dictionary`	This key is optional and contains a dictionary with the keyboard equivalent used to invoke the service menu command. Similar to `NSMenuItem`, the only key in the dictionary is called `default` and its value is a single character. Users invoke this keyboard equivalent by pressing the Command and Shift key modifiers along with the character.
NSUserData	`String`	This key is an optional string that contains a value of your choice.
NSTimeout	`String`	This key is an optional numerical string that indicates the number of milliseconds Services should wait for a response from the application providing a service when a response is required.

UIInterfaceOrientation

`UIInterfaceOrientation` (`String`) specifies the initial orientation of the application's user interface.

This value is based on the `Interface Orientation Constants` constants declared in the `UIApplication.h` header file. The default style is `UIInterfaceOrientationPortrait`.

UIPrerenderedIcon

`UIPrerenderedIcon` (`Boolean`) specifies whether the iPhone OS applies sheen and bevel effects to the application icon.

Value	Description
`YES`	iPhone OS doesn't apply sheen and bel effects to the application icon.

Value	Description
NO	(Default) iPhone OS applies sheen and bel effects to the application icon.

UIRequiresPersistentWiFi

UIRequiresPersistentWiFi (Boolean) specifies whether the application requires a Wi-Fi connection. iPhone OS maintains the active Wi-Fi connection open while the application is running.

Value	Description
YES	iPhone OS opens a Wi-Fi connection when this application is launched and keeps it open while the application is running. Use with Wi-Fi–based applications.
NO	(Default) iPhone OS closes the active Wi-Fi connection after 30 minutes.

UIStatusBarHidden

UIStatusBarHidden (Boolean) specifies whether the status bar is initially hidden when the application launches.

Value	Description
YES	Hides the status bar.
NO	Shows the status bar.

UIStatusBarStyle

UIStatusBarStyle (String) specifies the style of the status bar as the application launches.

This value is based on the Status Bar Style Constants constants declared in UIApplication.h header file. The default style is UIStatusBarStyleGray.

UTExportedTypeDeclarations

UTExportedTypeDeclarations (DictionaryArray) declares the uniform type identifiers (UTIs) owned and exported by the application. You use this key to declare your application's custom data formats and associate them with UTIs. Exporting a list of UTIs is the preferred way to register your custom file types; however, Launch Services recognizes this key and its contents only in Mac OS X v10.5 and later. This key is ignored on versions of Mac OS X prior to version 10.5.

The value for the UTExportedTypeDeclarations key is an array of dictionaries. Each dictionary contains a set of key-value pairs identifying the attributes of the type declaration. Table 7 lists the keys you can include in this dictionary along with the typical values they contain. These keys can also be included in array of dictionaries associated with the "UTImportedTypeDeclarations" (page 38) key.

Table 7 UTI property list keys

Key	Type	Description
`UTTypeConformsTo`	`Array`	(Required) Contains an array of strings. Each string identifies a UTI to which this type conforms. These keys represent the parent categories to which your custom file format belongs. For example, a JPEG file type conforms to the `public.image` and `public.data` types. For a list of high-level types, see System-Declared Uniform Type Identifiers in *Uniform Type Identifiers Overview*.
`UTTypeDescription`	`String`	A user-readable description of this type. The string associated with this key may be localized in your bundle's `InfoPlist.strings` files.
`UTTypeIconFile`	`String`	The name of the bundle icon resource to associate with this UTI. You should include this key only for types that your application exports.
`UTTypeIdentifier`	`String`	(Required) The UTI you want to assign to the type. This string uses the reverse-DNS format, whereby more generic types come first. For example, a custom format for your company would have the form `com.<yourcompany>.<type>.<subtype>`.
`UTTypeReferenceURL`	`String`	The URL for a reference document that describes this type.
`UTTypeTag-Specification`	`Dictionary`	(Required) A dictionary defining one or more equivalent type identifiers. The key-value pairs listed in this dictionary identify the filename extensions, MIME types, OSType codes, and pasteboard types that correspond to this type. For example, to specify filename extensions, you would use the key `public.filename-extension` and associate it with an array of strings containing the actual extensions. For more information about the keys for this dictionary, see System-Declared Uniform Type Identifiers in *Uniform Type Identifiers Overview*.

For more information about UTIs and their use, see *Uniform Type Identifiers Overview*.

UTImportedTypeDeclarations

`UTImportedTypeDeclarations` (`Array`) declares the uniform type identifiers (UTIs) inherently supported (but not owned) by the application. You use this key to declare any supported types that your application recognizes and wants to ensure are recognized by Launch Services, regardless of whether the application that owns them is present. For example, you could use this key to specify a file format that is defined by another company but which your program can read and export.

The value for the `UTExportedTypeDeclarations` key is an array of dictionaries and uses the same keys as those for the "UTExportedTypeDeclarations" (page 37) key. For a list of these keys, see Table 7 (page 38).

For more information about UTIs and their use, see *Uniform Type Identifiers Overview*.

The Preferences System

Preferences are application or system options that allow users to customize their working environment. Most applications read in some form of user preferences. For example, a document-based application may store preferences for the default font, automatic save options, or page setup information. Preferences are not limited to applications, however. You can read and write preference information, including user preferences, from any frameworks or libraries you define.

The preferences system of Mac OS X includes built-in support for preserving and restoring user settings across sessions. Both Carbon and Cocoa applications can use Core Foundation's Preference Services for reading and writing preference information. Cocoa applications can also use the `NSUserDefaults` class to read user preferences.

Important: The assumption with user preferences is that they are not critical; if they are lost, the application should be able to recreate the default set of preferences. You should not store an application's initial configuration data as a preference. Initial configuration data *is* critical and should be stored in a property list inside the application package.

The preferences system associates preference values with a key, which you use to retrieve the preference value later. User preferences have a scope based on a combination of the user login ID, application ID, and host (computer) name. This mechanism allows you to create preferences that apply at different levels. For example, you can save a preference value that applies to any of the following entities:

- the current user of your application on the current host
- all users of your application on a specific host connected to the local network
- the current user of your application on any host connected to the local network (the usual category for user preferences)
- any user of any application on any host connected to the local network

Applications should store only those preferences that represent information captured from the user. Storing the same set of default preferences for each user is an inefficient way to manage your application's preferences. Preferences are stored in property list files that must be parsed to read in the preference information. A more efficient way to manage preferences is to store a set of default preferences internally and then apply any user-customized preferences on top of the default set.

How Preferences Are Stored

The preferences system stores preference data in files located in the `Library/Preferences` folder in the appropriate file-system domain. For example, if the preference applies to a single user, the file is written to the `Library/Preferences` folder in the user's home directory. If the preference applies to all users on a network, it goes in `/Network/Library/Preferences`.

The name of each file in `Library/Preferences` is comprised of the application's bundle identifier followed by the `.plist` extension. For example, the bundle identifier for the TextEdit application is `com.apple.TextEdit` so its preferences file name is `com.apple.TextEdit.plist`.

To ensure that there are no naming conflicts, Apple strongly recommends that bundle identifiers take the same form as Java package names—your company's unique domain name followed by the application or library name. For example, the Finder uses the identifier `com.apple.finder`. This scheme minimizes the possibility of name collision and leaves you the freedom to manage the identifier name space under your corporate domain. See the property-list key "CFBundleIdentifier" (page 27) for more information.

Problems might ensue if an application tries to write preferences to a location other than `Library/Preferences` in the appropriate file-system domain. For one thing, the preferences APIs aren't designed for this difference. But more importantly, preferences stored in unexpected locations are excluded from the preferences search list and so might not be noticed by other applications, frameworks, or system services.

In Mac OS X version 10.3 and earlier, preferences were saved in the XML property list format. In Mac OS X version 10.4 and later, preferences are saved in the binary plist format. You can convert a file from one format to another using the `plutil(1)` tool (for example so that you can examine the plist in XML form), but you should not rely on the format of the file. You should refrain from editing preference files manually. Entering incorrect information or malformed data could cause problems when your application tries to read the file later. The correct way to extract information from preference domains in your application is through the preferences APIs.

Preference Domains

When your application searches for an existing preference value, the preferences system uses the current preference domain to limit the scope of the search. Similarly, when your application writes out new preferences, the values are scoped to the current domain.

Preference domains are identified by three pieces of information: a user ID, an application identifier, and a host name. In most cases, you would specify preferences for the current user and application. However, you might also decide to store application-level preferences. To do that, you would use the functions in the Core Foundation Preferences Utilities to specify exactly which domain you wanted to use. For information on how to use these routines, see *Preferences Programming Topics for Core Foundation*.

Table 1 shows all of the preference domains. The routines for retrieving preferences search through the preference domains in the order shown here until they find the requested key. Thus, if a preference is not found in a more user-specific and application-specific domain, the routines search the more global domains for the information.

Table 1 Preference domains in search order

Search order	User Scope	Application Scope	Host Scope
1	Current User	Current Application	Current Host
2	Current User	Current Application	Any Host
3	Current User	Any Application	Current Host
4	Current User	Any Application	Any Host

Search order	User Scope	Application Scope	Host Scope
5	Any User	Current Application	Current Host
6	Any User	Current Application	Any Host
7	Any User	Any Application	Current Host
8	Any User	Any Application	Any Host

The defaults Utility

The preferences system of Mac OS X includes a command-line utility named `defaults` for reading, writing, and removing preferences (also known as user defaults) from the application domain or other domains. The `defaults` utility is invaluable as an aid for debugging applications. Many preferences are accessible through an application's Preference dialog (or the equivalent), but preferences such as the position of a window aren't always available. For those preferences, you can view them with the `defaults` utility.

To run the utility, launch the Terminal application and, in a BSD shell, enter `defaults` plus command options describing what you want. For a terse description of syntax and arguments, run the `defaults` command by itself. For a more complete description, read the man page for `defaults` or run the command with the `usage` argument:

```
$ defaults usage
```

You should avoid changing values using the `defaults` tool while the target application is running. If you make such a change, the application is unlikely to see the change and more likely to overwrite the new value you just specified.

Environment Variables

Environment variables are another way to configure your application dynamically. Many applications and systems use environment variables to store important information, such as the location of executable programs and header files. The variable consists of a key string with the name of the variable and a value string.

To get the value of an environment variable, your application must call the `getenv` function that is part of the standard system library (`stdlib.h`). You pass this function a string containing the name of the variable you want and it returns the value, or NULL if no variable of that name was found. Your application can then use the variable as it sees fit.

Environment Variable Scope

Environment variables are scoped to the process that created them and to any children of that process. The Terminal application treats each window as its own separate process for the sake of managing environment variables. Thus, if you create a Terminal window and define some environment variables, any programs you execute from that window inherit those variables. However, you cannot access the variables defined in the first window from a second Terminal window, and vice versa.

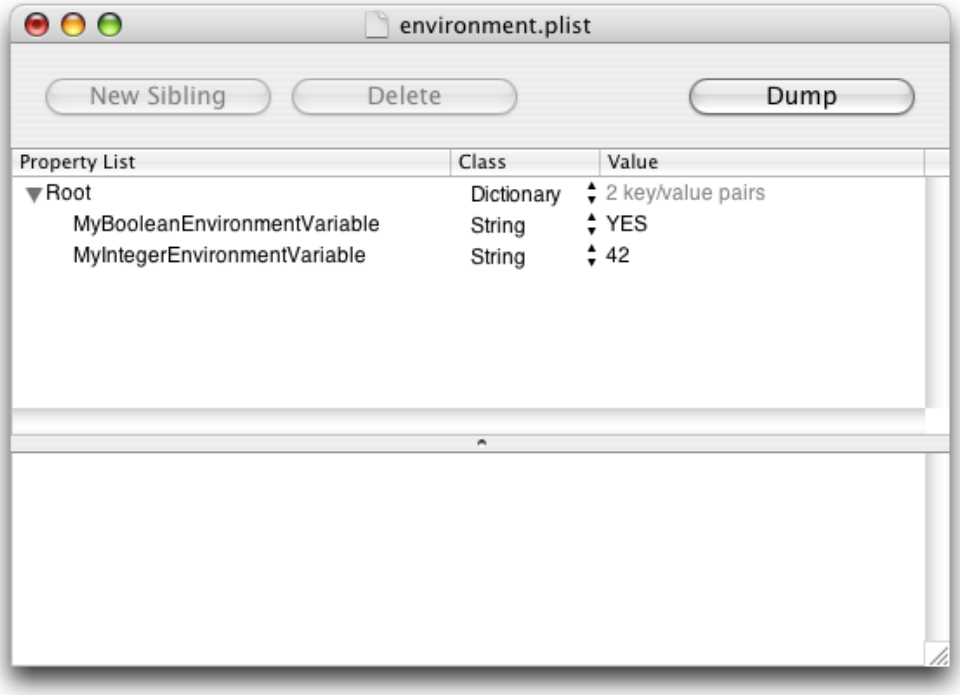
Sessions can be inherited. For example, when a user logs in, the system creates a user session and defines a standard set of environment variables. Any processes launched by the user during the session inherit the user environment variables. However, this inheritance is a read-only relationship. Any changes made to the variable by a process remain local to that process and are not inherited by other processes.

User Session Environment Variables

Mac OS X supports the definition of environment variables in the scope of the current user session. On login, the loginwindow application looks for a special property list file with the name `environment.plist`. This file must be located in a directory called `.MacOSX` at the root of the user's home directory. The path to this file is as follows:

```
~/.MacOSX/environment.plist
```

If an `environment.plist` file exists, loginwindow looks for keys that are children of the root element. For each of these keys, loginwindow registers an environment variable of the same name and assigns it the value of the key. This file supports only the definition of environment variables. You cannot use this file to execute other forms of script code. The format of the file is the same XML format as other property list files, with each key in the file containing a string value. For example, in the Property List Editor application (located in `<Xcode>/Applications/Utilities`, where `<Xcode>` is your Xcode installation directory), such a property-list file might look like the following:



Application-Specific Environment Variables

There are two ways to make environment variables available to an application. The first is to define the variables in a Terminal session and then launch the application from the same session. When launched from Terminal, the application inherits the session settings, including any environment variables defined there.

The second way to associate environment variables with an application is to include the `LSEnvironment` key in the application's information property list file. The `LSEnvironment` key lets you specify an arbitrary number of key/value pairs representing environment variables and their values. Because it requires modifying the application's information property list file, use of this key is best for options that do not change too frequently. For more information on using this key, see "Property List Key Reference" (page 17).

Guidelines for Configuring Applications

The primary way to configure an application is with an information property list file. In this file goes the information needed to register the application with the Finder and Launch Services. The following sections show you which keys to use when configuring an application and some legacy techniques for configuring an application.

Information Property List Files

An application bundle should always contain an information property list file with keys to identify the application to the Finder and Launch Services. The sections that follow describe the required and recommended keys you should include. For a complete description of these keys, see "Property List Key Reference" (page 17).

Required Properties

The following sections list the keys applications should include in their information property list files.

Application Keys

At a minimum, all applications should contain the following keys in their information property list file:

- `CFBundleDisplayName`
- `CFBundleIdentifier`
- `CFBundleName`
- `CFBundlePackageType`
- `CFBundleShortVersionString`
- `CFBundleSignature`
- `CFBundleVersion`
- `LSHasLocalizedDisplayName`
- `NSHumanReadableCopyright`
- `NSAppleScriptEnabled`

These keys identify your application to the system and provide some basic information about the services it provides. Cocoa applications should also include the following keys to identify key resources in the bundle:

- `NSMainNibFile`

- NSPrincipalClass

Note: If you are building a Cocoa application using an Xcode template, the NSMainNibFile and NSPrincipalClass keys are typically already set in the template project.

Document Keys

If your application associates itself with one or more document types, you should include a CFBundleDocumentTypes key to identify those types. The entry for each document type should contain the following keys:

- CFBundleTypeIconFile
- CFBundleTypeName
- CFBundleTypeRole

In addition to these keys, it must contain at least one of the following keys:

- LSItemContentTypes
- CFBundleTypeExtensions
- CFBundleTypeMIMETypes
- CFBundleTypeOSTypes

The LSItemContentTypes key takes precedence over other keys present when the application runs in Mac OS X v10.4 and later. You can continue to include the other keys for compatibility with older versions of the system, however.

Recommended Properties

If you are building a universal binary, you should generally specify the preferred executable architectures you support. Although the native architecture for the current platform is preferred, you may need to run your application under a different architecture to support legacy plug-ins.

To specify which environment you want your application to run in, include the following key in your information property list file:

- LSExecutableArchitectures
- LSRequiresNativeExecution

The LSRequiresNativeExecution key is recommended only if you want to ensure that your universal binary does not run under Rosetta because a PowerPC architecture is preferred over Intel-based architectures. For more information, see "LSExecutableArchitectures" (page 30).

Localized Properties

The following list contains the keys that are appropriate to include in your language-specific `InfoPlist.strings` files:

- `CFBundleDisplayName`
- `CFBundleName`
- `CFBundleShortVersionString`
- `NSHumanReadableCopyright`
- `CFBundleGetInfoString`

Document Configuration

Information property list files let you define the role your application plays for its supported document and Clipboard (pasteboard) types. This role defines the relationship between your application and the associated type. Your application can take one of the following roles for any given type:

- **Editor**. The application can read, manipulate, and save the type.
- **Viewer**. The application can read and present data of that type.
- **Shell**. The application provides runtime services for other processes—for example, a Java applet viewer. The name of the document is the name of the hosted process (instead of the name of the application), and a new process is created for each document opened.
- **None**. The application does not understand the data, but is just declaring information about the type (for example, the Finder declaring an icon for fonts).

The role you choose applies to all of the concrete formats associated with the document or Clipboard type. For example, the Safari application associates itself as a viewer for documents with the ".html", ".htm", "shtml, or "jhtml" filename extensions. Each of these extensions represents a concrete type of document that falls into the overall category of HTML documents. This same document can also support MIME types and 4-byte OS types used to identify files in Mac OS 9.

The PkgInfo File

The `PkgInfo` file is an alternate way to specify the type and creator codes of your application or bundle. This file is not required, but can improve performance for code that accesses this information. Regardless of whether you provide this file, you should always include type and creator information in your information property list file using the `CFBundlePackageType` and `CFBundleSignature` keys, respectively.

The contents of the `PkgInfo` file are the 4-byte package type followed by the 4-byte signature of your application. Thus, for the TextEdit application, whose type is `'APPL'` and whose signature is `'ttxt'`, the file would contain the ASCII string "APPLttxt".

Using a 'plst' Resource

It is possible to incorporate configuration information into a single-file, CFM-based application. However, if you want your application to run natively in Mac OS X—as opposed to running in the Classic compatibility environment—you must provide a `'plst'` resource. The `'plst'` resource allows the Finder to handle your application and document types properly. If your application does not contain this resource, the Finder automatically runs your application in the Classic compatibility environment.

To create a `'plst'` resource, add a new instance with ID 0 to your application resource fork. The content of this resource is the raw XML text from what would be your `Info.plist` file if your application were bundled. The encoding of this XML data should be UTF-8.

See "Putting Info.plist Files in a Flat Executable" (page 15) for information on including an information-property list file in an unbundled Mach-O executable.

Using Launch Arguments

If you have a Cocoa application, you can override many user defaults settings by specifying them on the command line. In addition, Cocoa recognizes a few additional arguments for opening and printing files. Table 1 lists some of the more commonly used command-line arguments for Cocoa applications.

Table 1 Command-line arguments for Cocoa applications

Argument	Description
`-NSOpen`*fileName*	Opens the specified file after the application finishes launching. Uses the `application: openFile:` method of the application's delegate to open the file.
`-NSOpenTemp`*fileName*	Opens the specified file as a temp file after the application finishes launching. Uses the `application: openTempFile:` method of the application's delegate to open the file.
`-NSPrint`*fileName*	Prints the specified file after the application finishes launching. Uses the `application: printFile:` method of the application's delegate to print the file.
`-NSShowAllDrawing<YES>`	Shows areas that are about to be drawn in yellow so that you can see which parts of your views are being updated. This is similar to the feature that is available through the Quartz Debug application but operates only on the specified application.
`-NSTraceEvents<YES>`	Displays a running log of events received by the application.

Document Revision History

This table describes the changes to *Runtime Configuration Guidelines*.

Date	Notes
2008-07-08	Updated multiplatform information.
	Added `LSRequiresIPhoneOS`, `UIRequiresPersistentWiFi`, `UIStatusBarStyle`, `UIStatusBarHidden`, `UIInterfaceOrientation`, `LSFileQuarantineEnabled`, `LSHandlerRankkeys`.
2007-04-18	Updated property list keys to include UTI-based keys. Updated configuration guidelines to include Intel-based keys.
2006-11-07	Reintroduced the CFBundleGetInfoString key and clarified details about the NSAppleScriptEnabled key.
	Added details on the new purpose of the `CFBundleGetInfoString` key.
	Clarified the possible types of the `NSAppleScriptEnabled` key.
2006-09-05	Added definition of NSPersistentStoreTypeKey.
2006-07-24	Updated description of the CFBundleVersion and CFBundleShortVersionString keys.
	Undocumented the `CFBundleGetInfoString` key.
	Made minor editorial changes.
2006-04-04	Updated description of CFBundleIdentifier key.
2005-11-09	Modified example for LSMinimumSystemVersion key.
2005-08-11	Updated description of NSPrincipalClass key. Added information about how to put Info.plist data into flat executables. Added environment.plist illustration.
2005-04-29	Updated for Mac OS X v10.4.
2005-02-03	Added CFBundleAllowMixedLocalizations key. Removed CFBundleGetInfoHTML key, which was included erroneously and is not supported.
2004-08-31	Added notes about the correct capitalization of files and directories in a bundle.
2004-04-15	Minor bug fixes.
2004-01-08	Minor bug fixes.
2003-12-02	Minor bug fixes.

Date	Notes
2003-08-07	First version of *Runtime Configuration*. Some of the information in this topic previously appeared in *System Overview*.