
Runtime Configuration Guidelines

Data Management: Preference Settings



2009-10-19



Apple Inc.
© 2003, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, AppleScript, Carbon, Cocoa, Mac, Mac OS, Quartz, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Finder and iPhone are trademarks of Apple Inc.

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction 7

Organization of This Document 7
See Also 7

Information Property List Files 9

The Preferences System 11

How Preferences Are Stored 11
Preference Domains 12
The defaults Utility 13

Environment Variables 15

Environment Variable Scope 15
User Session Environment Variables 15
Application-Specific Environment Variables 16

Additional Configuration Tips 17

The PkgInfo File 17
Using Launch Arguments 17

Document Revision History 19

Tables

The Preferences System 11

Table 1 Preference domains in search order 12

Additional Configuration Tips 17

Table 1 Command-line arguments for Cocoa applications 17

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 9) provides an introduction to information property list files and how they are used by the system.
- [“The Preferences System”](#) (page 11) discusses the role and scope of user preferences and describes the use of the `defaults` tool for accessing preferences.
- [“Environment Variables”](#) (page 15) 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.
- [“Additional Configuration Tips”](#) (page 17) lists the required and recommended configuration options for applications. This article also describes additional ways to configure both bundled and non-bundled applications.

See Also

For information about the keys and values you can include in an information property list file, see *Information Property List Key Reference*.

For additional information about the preferences system, see *User Defaults Programming Topics for Cocoa* or *Preferences Programming Topics for Core Foundation*.

Information Property List Files

An information property list file is a structured text file that contains essential configuration information for a bundled executable. The file itself is typically encoded using the Unicode UTF-8 encoding and the contents are structured using XML. The root XML node is a dictionary, whose contents are a set of keys and values describing different aspects of the bundle. The system uses these keys and values to obtain information about your application and how it is configured. As a result, all bundled executables (plug-ins, frameworks, and applications) are expected to have an information property list file.

By convention, the name of an information property list file is `Info.plist`. This name of this file is case sensitive and must have an initial capital letter `I`. In iPhone applications, this file resides in the top-level of the bundle directory. In Mac OS X bundles, this file resides in the bundle's `Contents` directory. Xcode typically creates this file for you automatically when you create a project of an appropriate type.

The contents of a typical `Info.plist` file convey the following information to the system:

- The user-visible name to display for the bundle
- A unique identifier string (typically in the form `com.yourcompany.appname`) that can be used to locate the bundle at runtime
- The type of the bundle (application, framework, plug-in)
- Version information
- Information about how to launch the bundle or load its contents into memory
- The preferred execution environment for the bundle
- Information about the bundle's supported document types (if any)
- For iPhone applications, information about how the application presents content initially

For information about how to create information property lists, along with keys and values that you can include in them, see *Information Property List Key Reference*.

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. You assign this value to the `CFBundleIdentifier` key in your information property list file.

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

Search order	User Scope	Application Scope	Host Scope
4	Current User	Any Application	Any Host
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.

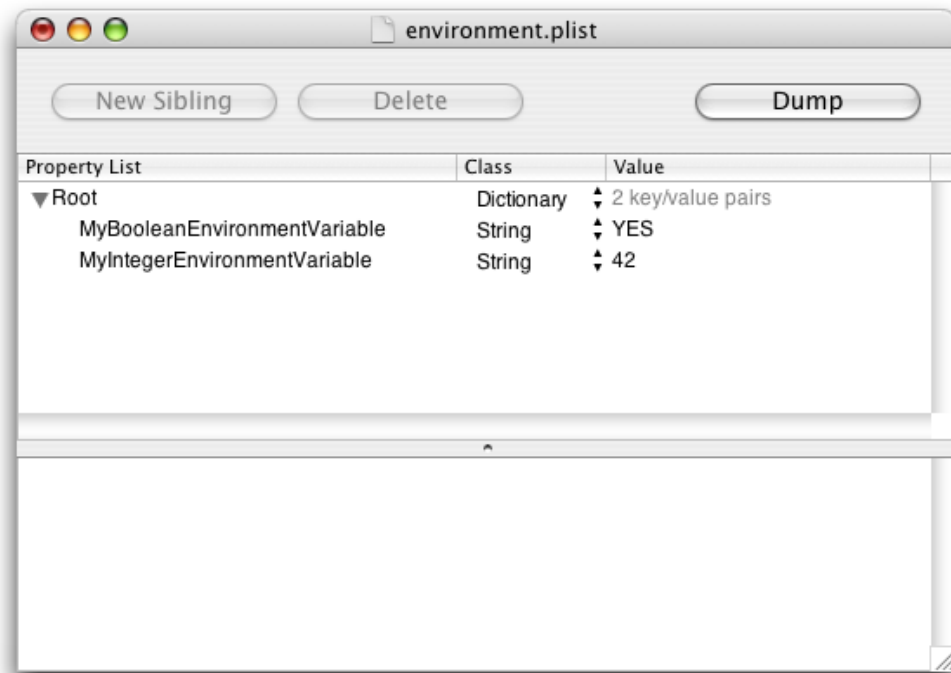
Sessions can be inherited. For example, when a user logs in, the system creates a user session and defines a standard set of environment variables. Any processes launched by the user during the session inherit the user environment variables. However, this inheritance is a read-only relationship. Any 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 (which you must create) 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 .

Additional Configuration Tips

This chapter describes some miscellaneous techniques for configuring your application.

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 `'ttxx'`, the file would contain the ASCII string `"APPLttxx"`.

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
<code>-NSOpen$fileName$</code>	Opens the specified file after the application finishes launching. Uses the <code>application: openFile: method</code> of the application's delegate to open the file.
<code>-NSOpenTemp$fileName$</code>	Opens the specified file as a temp file after the application finishes launching. Uses the <code>application: openTempFile: method</code> of the application's delegate to open the file.
<code>-NSPrint$fileName$</code>	Prints the specified file after the application finishes launching. Uses the <code>application: printFile: method</code> of the application's delegate to print the file.
<code>-NSShowAllDrawing<YES></code>	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.
<code>-NSTraceEvents<YES></code>	Displays a running log of events received by the application.

Document Revision History

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

Date	Notes
2009-10-19	Removed deprecated information from the document.
	Moved the reference information for <code>Info.plist</code> keys to <i>Information Property List Key Reference</i> .
2009-09-09	Clarified that while a <code>CFBundleIdentifier</code> is similar to a UTI, it is not actually a UTI, as the allowed character set is more restricted.
2009-08-14	Added links to Cocoa Core Competencies.
2009-05-15	Added information about keys introduced in Mac OS X v10.6.
	Documented <code>QuartzGLEnable</code> key.
	Updated information on version strings to specify that the full three-digit string is required, e.g. 10.4.0.
2008-07-08	Updated multiplatform information.
	Added <code>LSRequiresiPhoneOS</code> , <code>UIRequiresPersistentWiFi</code> , <code>UIStatusBarStyle</code> , <code>UIStatusBarHidden</code> , <code>UIInterfaceOrientation</code> , <code>LSFileQuarantineEnabled</code> , <code>LSHandlerRank</code> keys.
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 <code>CFBundleGetInfoString</code> key and clarified details about the <code>NSAppleScriptEnabled</code> key.
	Added details on the new purpose of the <code>CFBundleGetInfoString</code> key.
	Clarified the possible types of the <code>NSAppleScriptEnabled</code> key.
2006-09-05	Added definition of <code>NSPersistentStoreTypeKey</code> .
2006-07-24	Updated description of the <code>CFBundleVersion</code> and <code>CFBundleShortVersionString</code> keys.
	Undocumented the <code>CFBundleGetInfoString</code> key.
	Made minor editorial changes.
2006-04-04	Updated description of <code>CFBundleIdentifier</code> key.

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