Preferences Programming Topics for Core Foundation

Data Management: Preference Settings

2006-10-03

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

Simultaneously published in the United States and Canada.

Contents

Tables and Listings

Introduction to Preferences Programming Topics for Core Foundation

Most applications need to store and retrieve preferences to allow for user customization of application behavior and provide a way to keep track of configuration settings across multiple launches. Frameworks and libraries can also use preferences to store configuration information. Core Foundation preferences provide a simple and standard way to maintain preferences for both types of use.

Preferences allow you to store values that are associated with a key that can later be used to "look up" the preference value when you need it. Key/value pairs are assigned a scope using a combination of username, application ID, and host (computer) name. This mechanism allows you to create preferences which apply to different classes of users. For example, using preferences you can save a preference value that applies to:

- 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
- Any user of any application on any host connected to the local network

Preferences has a high-level API which makes it very simple to store and retrieve application preferences using the default scope (current user, any host) which is appropriate for the majority of situations. There is also a low-level API which allows you to specify the exact scope of a preference value when necessary.

Preferences uses the Core Foundation property list types to store and retrieve preference values. Readers unfamiliar with property lists should consult the Core Foundation Topic *Property List Programming Topics for Core Foundation* for more information.

Organization of This Document

This topic contains conceptual information you need to understand in order to use the preferences API, and examples that demonstrate how to save and retrieve preference values. The concepts covered in this topic are:

- "Preferences Best Practices" (page 9)
- "Application IDs" (page 11)
- "Preference Domains" (page 13)

The tasks covered in this topic are:

- "Using the High-Level Preferences API" (page 15)
- "Using the Low-Level Preferences API" (page 19)

Preferences Best Practices

CFPreferences is Apple's standard API for storing and retrieving preference keys and values, allowing the calling process to concentrate on native types and key-value pair meanings rather than the mechanisms of writing to files and so on. While it provides a convenient API, it is also easy to use incorrectly. This document gives an overview of when it is appropriate to use what API, and how to synchronize preferences across process boundaries.

When To Use What API

The following general guiding principles apply to the CFPreferences API:

- You should typically use `CFPreferencesCopyAppValue` to read preference keys.
- You should use `CFPreferencesSetAppValue` to write preference keys for "current user/any host."
- If you need to write a by-host preference for the current user, use `CFPreferencesSetValue`. Make sure this is absolutely necessary.

Note that, although they are treated separately in the documentation, high-level API and low-level API are not exclusive. It may be appropriate to use high-level API in some parts of an application, and low-level API in another. For example, you can set a preference key/value pair with `CFPreferencesSetValue(key, value, app, user, host)` and then read it with `CFPreferencesCopyAppValue(key, value, app)`—indeed you probably do want to read it with the latter function since it traverses the search path.

High Level API

As much as possible, you should use `CFPreferencesCopyAppValue` to retrieve preference keys. This function traverses the search path looking for the matching key and returns the value from the most-specific domain.

Users of particular machines may also be subject to "management" through "Workgroup Manager" or the "Capabilities" option of the Accounts preference pane in System Preferences. Either of these mechanisms may force preference values on the user. These values are also picked up by the `CFPreferencesCopyAppValue` API—you should use this function to ensure your application properly responds to management of this kind.

Low Level API

If your application needs to distinguish between "the current host" and "any host" then you use the low level API. If for some reason you need to search for a key-value pair in a specific domain, you should use `CFPreferencesCopyValue`—you should not use this function as a general retrieval mechanism.

Synchronizing Preferences Across Process Boundaries

The Rule of Thumb on CFPreferences synchronization:

1. Only synchronize when absolutely necessary
2. If you have to communicate across process boundaries, use notifications with appropriate granularity, keeping 1) in mind. Typically, this means setting a flag in client processes, and only if a changed key is required should you trigger a synchronize.

Many processes in OS X write a preference key-value pair for use in another process. While it would be convenient for preference key-value pairs to auto-update in other processes, guaranteeing auto-update in all circumstances would incur a performance penalty and also make it difficult to ensure related preferences are read and written consistently. Processes should always have the choice of when to elect to accept new information into their space. For preference values, `CFPreferencesSynchronize` and `CFPreferencesAppSynchronize` are the function calls that providing the information choke-point. You should typically not, however, call these functions before every read of a preference key.

Preference File Locations and Debugging

Preferences files are stored in the system's or user's preferences directories. On Mac OS X versions 10.0 to 10.4 these are in `/Library/Preferences` and in `/Library/Preferences` in the user's home directory respectively. When debugging an application, it may sometimes be useful to inspect these files to determine that preferences have been saved correctly, however you should *never* hardcode these paths into an application. If you do need to access the directory programmatically you should use the `NSSearchPathForDirectoriesInDomains` API, although there should typically be no reason to do so.

Note that preferences you set up in the registration domain (see Defaults Domains in *User Defaults Programming Topics*) are *not* stored in the preferences file. Put another way, the preferences file stores only values that are different from those in the registration domain, so you should not expect to see "default defaults" in the preferences file after you run your application.

Managed Preferences

Mac OS X 10.2 introduced the concept of "managed preferences." The function `CFPreferencesAppValueIsForced` determines whether or not a given key has been imposed on the user. For managed keys, you should disable any user interface that allows the user to modify the value for the key.

Application IDs

Preferences store preference data on disk in files named using an application ID that you provide. To ensure that there are no naming conflicts, it's a good idea to define and set a bundle identifier for your application and use it as the application ID for preferences. 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 `com.apple.Finder` or `com.foo.ImageImport`. Using this scheme minimizes the possibility of collision, and leaves you responsible for managing the identifier namespace under your corporate domain. See *Bundle Programming Guide* for more information on bundles and bundle IDs.

Preference Domains

When creating a new preference or searching for an existing one, Core Foundation uses the notion of "Preference Domains" to specify the scope and location of the preference. A preference domain consists of three pieces of information, an application ID, a host name, and a user name. Table 1 (page 13) shows all of the preference domains, listed in the order that they are searched when attempting to locate a preference value.

Table 1 Preference Domains in search order

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
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

When using the high-level preferences functions `CFPreferencesSetAppValue`, and `CFPreferencesCopyAppValue`, you need only specify the application ID. The first function, `CFPreferencesSetAppValue`, places the preference value into the "Current User" and "Any Host" domain for the application, meaning that the standard location for application preferences is domain number two as listed in Table 1 (page 13). The other function, `CFPreferencesCopyAppValue`, searches through all the domains in order until a value is found. See "Using the High-Level Preferences API" (page 15) for information on using these functions.

Domain Qualifiers

If you need to specify an exact domain for your preference values you can use the low-level preferences functions `CFPreferencesSetValue`, and `CFPreferencesCopyValue`. These functions allow you to specify all three of the domain qualifiers when setting or searching for preferences. When using these functions you cannot pass arbitrary host and user names; you must instead use the appropriate "Any" or "Current" constants given in the list below. For the application domain qualifier, you can either pass the application ID or one of the "Any" or "Current" application constants given in the list below. See "Using the Low-Level Preferences API" (page 19) for information on using these functions.

`kCFPreferencesAnyApplication`	Indicates a preference that applies to any application.

`kCFPreferencesCurrentApplication`	Indicates a preference that applies only to the current application.
`kCFPreferencesAnyHost`	Indicates a preference that applies to any host.
`kCFPreferencesCurrentHost`	Indicates a preference that applies only to the current host.
`kCFPreferencesAnyUser`	Indicates a preference that applies to any user.
`kCFPreferencesCurrentUser`	Indicates a preference that applies only to the current user.

Using the High-Level Preferences API

The functions `CFPreferencesSetAppValue` and `CFPreferencesCopyAppValue` are the most straightforward way for an application to create and retrieve a preference that is specific to the current user and application. The preference data is written to the default domain (Current User, Current Application, Any Host) and so it will be available on all machines that this user can log into. These functions should never be called with `kCFPreferencesAnyApplication`, only a true application ID or `kCFPreferencesCurrentApplication`.

Saving a Simple Preference

Preferences are stored as key/value pairs. The key must be a CFString object, but the value can be any Core Foundation property list value (see *Property List Programming Topics for Core Foundation*), including the container types. For example, you might have a key called `defaultWindowWidth` which defines the width in pixels of any new windows that your application creates. Its value would most likely be of type CFNumber. You might also decide to combine window width and height into a single preference called `defaultWindowSize` and make its value be a CFArray object containing two CFNumber objects.

The code in Listing 1 (page 15) demonstrates how to create a simple preference for the application "MyTextEditor". The example sets the default text color for the application to blue.

Listing 1 Writing a simple default

```
CFStringRef textColorKey = CFSTR("defaultTextColor");
CFStringRef colorBLUE = CFSTR("BLUE");

// Set up the preference.
CFPreferencesSetAppValue(textColorKey, colorBLUE,
        kCFPreferencesCurrentApplication);

// Write out the preference data.
CFPreferencesAppSynchronize(kCFPreferencesCurrentApplication);
```

Notice that `CFPreferencesSetAppValue` by itself is not sufficient to create the new preference. A call to `CFPreferencesAppSynchronize` is required to actually save the value. If you are writing multiple preferences, it is more efficient to sync only once after the last value has been set than to sync after each individual value is set. For example, if you implement a preference panel you might only synchronize when the user presses an "OK" button. In other cases you might not want to sync at all until the application quits—although note that, of course, if the application crashes all unsaved preferences settings will be lost.

Reading a Simple Preference.

The simplest way to locate and retrieve a preference value is to use the `CFPreferencesCopyAppValue` function. This call searches through the various preference domains in order until it finds the key you have specified. If a preference has been set in a less-specific domain—"Any Application", for example —its value will be retrieved with this call if a more specific version cannot be found. Listing 2 (page 16) shows how to retrieve the text color preference saved in Listing 1 (page 15).

Listing 2 Reading a simple default

```
CFStringRef textColorKey = CFSTR("defaultTextColor");
CFStringRef textColor;

// Read the preference.
textColor = (CFStringRef)CFPreferencesCopyAppValue(textColorKey,
        kCFPreferencesCurrentApplication);
// When finished with value, you must release it
// CFRelease(textColor);
```

Note that all values returned from preferences are immutable, even if you have just set the value using a mutable object.

Updating a Simple Preference

An example of simple preference updating is a game that searches for a high score preference each time a round is completed. If there is no high score preference, the application writes the current score as the high score. If a high score preference exists, it is compared with the new score and updated if the new score is higher. Listing 3 (page 16) demonstrates this process.

Listing 3 Updating a preference

```
CFStringRef highScoreKey = CFSTR("High Score");
CFNumberRef tempScore;
int highScore;

// Look for the preference.
tempScore = (CFNumberRef)CFPreferencesCopyAppValue(highScoreKey,
        kCFPreferencesCurrentApplication);

// If the preference exists, update it. If not, create it.
if (tempScore)
{
    // Numbers come out of preferences as CFNumber objects.
    if (!CFNumberGetValue(tempScore, kCFNumberIntType, &highScore)) {
        highScore = 0;
    }
    CFRelease(tempScore);

    printf("The old high score was %d.", highScore);
}
else
{
```

```
    // No previous value.
    printf("There is no old high score.");
    highScore = 0;
}

highScore += 5;

// Create the CFNumber to pass to the preference API.
tempScore = CFNumberCreate(NULL, kCFNumberIntType, &highScore);

// Set the preference value, or update it if it already exists.
CFPreferencesSetAppValue(highScoreKey, tempScore,
        kCFPreferencesCurrentApplication);

// Release the CFNumber.
CFRelease(tempScore);

// Write out the preferences.
CFPreferencesAppSynchronize(kCFPreferencesCurrentApplication);
```

The technique shown in Listing 3 (page 16) generalizes to context of multiple preferences where an application tries to locate a set of preferences for display to the user in a graphical preference panel. If no preferences exist, default values are used. If existing preference values are found, they are used to initialize the preference panel for display to the user. After the user makes changes and pushes the “OK” button, you can set the changed preference values and write them to storage.

Using the Low-Level Preferences API

There are some cases where using the high-level API is not appropriate. If you are building some sort of "helper tool" that runs on behalf of another application, or an application that stores preferences for other applications, you will need to use the low-level preferences API to write to the other application's preferences. Listing 1 (page 19) shows you how to do this.

Listing 1 Writing a value to another application's preferences.

```
CFStringRef appID = CFSTR("com.apple.anotherapp");
CFStringRef defaultTextColorKey = CFSTR("defaultTextColor");
CFStringRef colorBLUE = CFSTR("BLUE");

// Set up the preference.
CFPreferencesSetValue(defaultTextColorKey,
                colorBLUE,
                appID,
                kCFPreferencesCurrentUser,
                kCFPreferencesAnyHost);

// Write out the preference data.
CFPreferencesSynchronize(appID,
                kCFPreferencesCurrentUser,
                kCFPreferencesAnyHost);
```

Note that this example writes to another application's preferences. There's no way to get the bundle ID directly from the other application, so it's necessary to hardcode the application ID.

Document Revision History

This table describes the changes to *Preferences Programming Topics for Core Foundation*.

Date	Notes
2006-10-03	Added a section about preferences file locations and access for debugging.
2006-02-07	Corrected minor typographic errors. Changed name from "Preferences".
2005-11-09	Made minor correction to code listings (added typecast to return value from CFPreferencesCopyAppValue).
2004-08-31	Added Best Practices section. Added note about immutability of returned values.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.