Preference Pane Programming Guide

User Experience: Windows & Views

2008-10-15

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

Simultaneously published in the United States and Canada.

Contents

Introduction 9

Who Should Read This Document? 9
Organization of This Document 9

Architecture of Preference Panes 11

Application Architecture 11
Design Architecture 12
Implementation 13

The Preference Application 15

System Preferences 15
Custom Preference Application 16
Target Application 17

Managing User Preferences 19

Preference Services 19
Configuration Files 20
Interprocess Communication 20
- Apple Events 21
- Distributed Objects 21
- Distributed Notifications 22
- Sockets and Ports 23
- Signals 23

Life Cycle of a Preference Pane 25

Instantiating the Preference Pane Object 25
Loading the Main View 25
- Dynamically Selecting the Main Nib File 26
- Assigning the Main View 26
- Setting Up the User Interface 27

Selecting 27
Deselecting 27
Longevity of Preference Pane Objects in System Preferences 28

Anatomy of a Preference Pane Bundle 29

Information Property Lists 29

Localizable Resources 30
Global Resources 30
Where Preference Panes Live 31

Updating Preference Panes for Snow Leopard and Beyond 33

Making Your Preference Pane 64-Bit 33
Using Garbage Collection 33
Supporting Sudden Termination 34

Preventing Name Conflicts 35

Uniqueness Algorithm 35
Categories 36

Wrapping Long Labels 37

Using Preference Services 39

Communicating With the Target Application 41

Using Distributed Objects 41
Using Distributed Notifications 41

Implementing a Preference Pane Help Menu 45

Adding Global Help Menu Items 45
Adding Dynamic Help Menu Items 46

Creating a Preference Pane Bundle 47

Create the Project 47
Create the Nib File 47
Create the Preference Pane Files 48
Update the Build Settings 48
Build and Install the Preference Pane 49

Implementing a Simple Preference Pane 51

Build the User Interface 51
Update the Header File 51
Implement the initWithBundle: Method 52
Implement the mainViewDidLoad Method 52
Implement the checkboxClicked: Method 53
Implement the didUnselect Method 53

Using Preference Panes in Other Applications 55

Embedding a Single Pane 55
Managing a Collection of Panes 56

Document Revision History 59

Figures, Tables, and Listings

Architecture of Preference Panes 11

Figure 1 Plug-in architecture of preference panes 12
Figure 2 Model-View-Controller design of preference panes 12

The Preference Application 15

Figure 1 Using a tabbed view to categorize related options 16

Managing User Preferences 19

Figure 1 Distributed object architecture 21
Figure 2 Distributed Notification Model 22
Table 1 Preference domains in precedence order 19

Life Cycle of a Preference Pane 25

Figure 1 Execution flow of loadMainView 26
Table 1 Return values of shouldUnselect 28

Anatomy of a Preference Pane Bundle 29

Figure 1 Contents of a preference pane bundle 29

Updating Preference Panes for Snow Leopard and Beyond 33

Figure 1 Figure 34

Implementing a Preference Pane Help Menu 45

Listing 1 Info.plst entry for the SMPL001 Help menu item 45
Listing 2 English Localizable.strings entry for the SMPL001 Help menu item 46
Listing 3 French Localizable.strings entry for the SMPL001 Help menu item 46
Listing 4 Dynamic help menu for a tab view 46

Introduction

Note: This document was formerly titled "Preference Panes."

Preference panes are dynamically loaded plug-ins that provide a graphical user interface to the system's or an application's user preferences. Preference panes can be presented to the user using the central System Preferences application, using a specialized preferences application, or as the Preferences menu item in an application's application menu. In System Preferences, each icon in its Show All view represents an individual preference pane plug-in. You can develop preference panes for use by System Preferences or by your own application.

The most common situation for using preference panes is an application that lacks its own user interface (or has a very restricted user interface such as the Mac OS X Login application) but needs to be configurable. Possible cases include a server application that always runs in the background or an application that makes its services available to other applications through the Services menu. To allow configuration of these applications, you must provide a user interface in a separate application. You should not require the user to hand-edit configuration files or execute the application from the command line with special arguments. Instead, create one or more preference pane plug-ins that contain the user interface and the code that can read and write the preference settings. Then, either supply your own "Setup" application or, if appropriate, use System Preferences to display the preference panes.

Who Should Read This Document?

You should read this document if you are a Cocoa developer who wants to provide a custom preference pane, accessible from the System Preferences applications, to your users. You should have a working knowledge of Cocoa programming with the Application Kit before attempting preference pane programming.

Organization of This Document

This document describes how to create and manage a preference pane, how to have System Preferences load your own preference pane, and how to load a preference pane in your own application.

Here are the concepts covered:

- "Architecture of Preference Panes" (page 11) describes the plug-in architecture of preference panes and how they interact with applications and the system.
- "The Preference Application" (page 15) describes the ways the preference pane can be presented to the user: System Preferences, a specialized preferences application, or inside the main application.
- "Managing User Preferences" (page 19) describes several ways the preference pane can interact with the system for manipulating preferences.

- "Life Cycle of a Preference Pane" (page 25) describes how the application interacts with the preference pane.
- "Anatomy of a Preference Pane Bundle" (page 29) describes the structure of a preference pane bundle.
- "Updating Preference Panes for Snow Leopard and Beyond" (page 33) describes the requirements for preference panes for Mac OS X version 10.6 and later—64-bit code, garbage collection, and support for sudden termination.

Here are the tasks covered:

- "Preventing Name Conflicts" (page 35) recommends a technique to prevent name conflicts between the global symbols in your preference pane and either the application or other preference panes.
- "Wrapping Long Labels" (page 37) describes how to modify the `.plist` file so that long labels are split across two lines.
- "Using Preference Services" (page 39) describes the methods available for reading and writing preferences to a preference file.
- "Communicating With the Target Application" (page 41) provides examples for notifying a separate target application of preference changes.
- "Creating a Preference Pane Bundle" (page 47) walks you through the steps of creating a skeletal preference pane bundle in Xcode and Interface Builder.
- "Implementing a Simple Preference Pane" (page 51) walks you through the source code of a simple preference pane, showing an example of how to implement a working preference pane.
- "Using Preference Panes in Other Applications" (page 55) describes the responsibilities of an application that loads a preference pane.

Architecture of Preference Panes

This section provides an overview of the preference pane architecture. It describes the various components involved in using preference panes and how they fit together, the design principle within the plug-in, and finally some implementation details.

Application Architecture

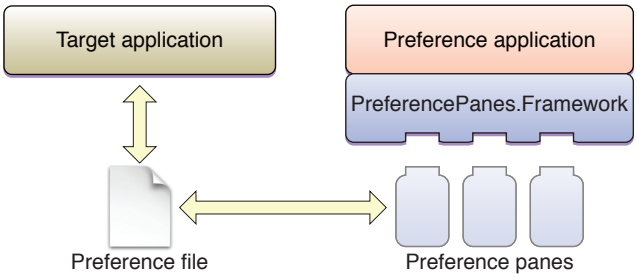
There are three logical components to the preference pane architecture: the application that loads the preference pane plug-ins (the preference application), the Preference Panes framework (`PreferencePanes.framework`), and the preference pane plug-ins themselves. The responsibilities of each are as follows:

- The preference application serves as the container of the preference pane: It loads the preference pane plug-in and provides the window in which the preference pane is displayed. When the plug-in is loaded, the application creates an instance of the plug-in's principle class, a subclass of NSPreferencePane. Through NSPreferencePane's interface, the application notifies the preference pane when the pane is displayed and again when it is being removed from the screen.
- The Preference Panes framework acts as the interface between the preference application and the preference pane plug-in. The framework provides the NSPreferencePane class, which is subclassed by the plug-in. The application uses the methods defined by NSPreferencePane to communicate with the plug-in. The default implementation of NSPreferencePane is able to load a default nib file and provide the application with a view containing the user interface.
- The preference pane plug-in provides the user interface for modifying the preferences and interacts with the system or another application to record the changes. The plug-in exports a principle class that is a subclass of the NSPreferencePane class. An instance of this subclass is created by the preference application. This instance, the preference pane object, initializes the user interface with the current preference settings, receives action messages from the interface when the user makes changes, and then records the changes when the user is finished.

In managing the preference settings, the preference pane object usually interacts with an additional component: the object to which the preferences apply. This target can be part of the operating system or one or more separate applications; the interaction can be by direct communication between the preference pane object and target or by indirect communication though the use of a preference file.

The plug-in architecture of preference panes is illustrated, showing a case of indirect communication with the target application, in Figure 1.

Figure 1 Plug-in architecture of preference panes

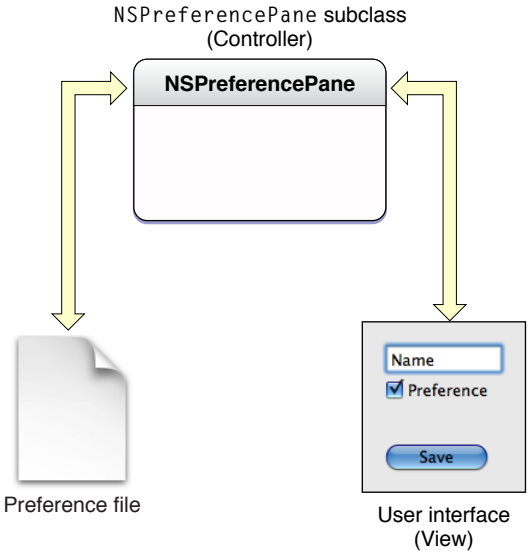


Design Architecture

Preference panes are built using a Model-View-Controller (MVC) design wherein the user interface (View) and data model (Model) are separated from one another with all communication going through a third object (Controller). Cocoa applications, as well as the Cocoa frameworks, are frequently implemented using MVC, which allows for greater flexibility and object reuse.

Figure 2 (page 12) shows the MVC design as it applies to preference panes. The `NSPreferencePane` subclass (the preference pane object) assumes the central role as Controller. It is the intermediary between the user interface defined within a nib file and the preference file, which holds the user's preferences. Through target-action connections between the user interface (the View) and the preference pane object, the user interface sends messages to the preference pane object as the user performs actions. The preference application can also be considered part of the View, since it provides the window for displaying the preference pane and notifies the preference pane object when the user selects and deselects the preference pane. The preference pane object translates these user actions into modified preference values and updates the values in the preference file (the Model).

Figure 2 Model-View-Controller design of preference panes



Implementation

The Preference Panes framework is an Objective-C framework built on top of Cocoa. As such, it can be used only by a Cocoa application, and the user interface you create for the preference pane module must also be implemented using Cocoa. You cannot create a Carbon-based, or Java-based, preference pane with this framework. In contrast, the application to which the preferences apply can be implemented in any language using any framework, provided it is able to communicate with the preference pane.

For storing preferences, Mac OS X has a built-in preference system, Core Foundation Preference Services, that provides all applications (Cocoa, Carbon, or BSD) the ability to easily read and write preference information to standardized XML-based files. When direct communication between applications is required, you can choose from a variety of methods from low-level signals and sockets to high-level Apple events, most of which are available to both Cocoa and Carbon applications.

The Preference Application

The preference application loads the preference pane plug-in and presents the preference pane to the user. Depending on the type and number of preferences you need to manage, you have several choices for the preference application. You can have the preference pane be displayed by System Preferences, by a custom preference application, or by the target application itself in response to the Preferences menu item.

System Preferences

System Preferences is the standard location for presenting system-level preferences. The preference panes shipped with Mac OS X include panes affecting hardware (such as the Sound, Mouse, and Display panes), software integrated into the system (such as the Dock and Screen Saver panes), and behavior applicable to every application (such as the International and General panes).

When your preferences apply to the system or to the user's environment as a whole, make the preference pane available to System Preferences. This may include panes for the following situations:

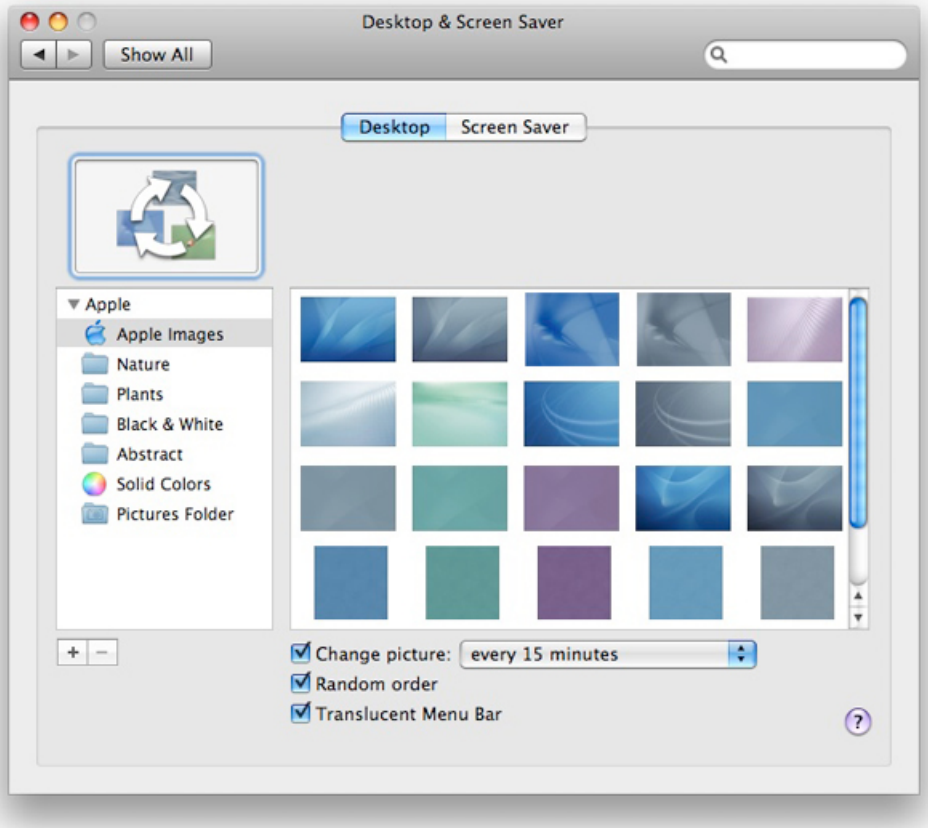
- additional input devices such as tablets, multi-function mice, and microphones
- configurable internal hardware such as processor upgrade cards
- light-weight faceless server applications such as a file server
- system-wide utilities such as keyboard macros

Unless your preference pane clearly belongs in System Preferences, use a custom preference application instead.

System Preferences searches for preference panes in four separate locations. Depending on where you install your preference pane bundle, it is available only to an individual user, to an individual computer, or to all computers and users on a network (see "Where Preference Panes Live" (page 31)).

The System Preferences window has a fixed width (668 pixels) but resizes itself vertically to fit the size of the current preference pane. Your preference pane should fit on the smallest supported screen resolution in Mac OS X: 800 x 600. If all your preferences cannot fit reasonably well within this size, you can use a tabbed view to divide the preferences into subsets as shown in Figure 1. If you find yourself creating more than a few tabs to hold all your preferences, you should create a custom preference application, instead. Do not split related preferences between multiple panes.

Figure 1 Using a tabbed view to categorize related options



Preference panes are self-contained modules. They cannot interact with nor extend other preference panes. In particular you cannot create a preference pane that adds another tab to one of Apple's standard preference panes such as the pane shown in Figure 1.

Custom Preference Application

If your preferences cannot be presented to the user from within the target application, due to the lack of a suitable user interface, and they do not provide system-level functionality appropriate for System Preferences, present the preferences in a custom preference application. In particular, use a custom preference application if any of the following is true:

- there are a large number of preferences (for example, for a Web server)
- the preferences apply to a background application that is not providing a service to the system or other applications (for example, distributed computing applications)
- more interaction is required than basic mouse clicking and short typing (for example, for training voice-recognition software)

Because System Preferences controls the window in which your preference pane is displayed, your layout options are restricted, especially when you have more than a few preferences. By using a custom preference application, you get greater freedom in designing the user interface. Rather than having a single preference pane with its preferences possibly split into tabbed views of a fixed height and width, preferences can be split between multiple preference panes with custom icons and unique sizes. Each pane can be customized to present the best possible interface for its contents.

If you have very few preferences and do not need to manage multiple preference panes, you can forego use of the Preference Panes framework altogether. You can more easily create a regular nib file containing the entire interface and have the application load it directly.

Target Application

If the preferences apply to an individual application with its own user interface, or several applications that share common preferences, use the application itself, loading the preference pane in response to the Preferences menu item. For a solitary application, store the preference pane bundles within the `Resources` directory of the application's package. For a suite of applications, store the bundles in your own subdirectory in `/Library/Application Support`.

If the preferences apply to an individual application, you are calling the preference pane in response to the user choosing Preferences from the menu, and you have only a few preferences that can all fit into a single preference pane, the Preference Panes framework provides little benefit over a regular nib file and a custom class.

When managing a collection of preferences, though, the framework provides a prebuilt architecture with several benefits. For one, the plug-in architecture takes advantage of lazy loading. The code and resources consumed by the preference pane do not need to be allocated until the user selects the preference pane. Its modular design also allows for greater reuse. For example, if you write multiple applications that have a common preference, such as a default font, you can have a single preference pane used by all applications. Finally, the programming interface is designed to support preference panes being displayed and then hidden as the user selects from a collection of preference panes. You do not need to design and create your own implementation for this.

Managing User Preferences

The Preference Panes framework defines the interface through which an application interacts with the preference pane object, but the application is responsible only for displaying the preference pane's user interface. The preference pane object is responsible for handling the preferences themselves. This section describes techniques by which the preference pane can store and communicate preferences to the target application.

Because preference panes are subject to sudden termination by `SIGKILL` when the user shuts down the device, preferences should be sent to the target application or saved to a preference file whenever the user makes a change, if practical. You should not wait for the pane to be dismissed to save the preferences. If you have complex sets of preferences that need to saved as a group, save the changes interactively to a temporary file, and provide an Apply button to copy the settings to the preferences file and/or target application, so that the device can be shut down without your pane either losing data or blocking shut down.

Preference Services

At the heart of the Mac OS X user preference system is Core Foundation Preference Services. This collection of routines defines a set of domains according to the user name, host name, and application ID to which a given preference value applies. Each component of the domain is specified by a CFString. Predefined constant strings are available to easily select either the "Current" instance of a component (such as the current application or current user) or a shared component available to "Any" instance.

`kCFPreferencesCurrentUser`	`kCFPreferencesAnyUser`
`kCFPreferencesCurrentApplication`	`kCFPreferencesAnyApplication`
`kCFPreferencesCurrentHost`	`kCFPreferencesAnyHost`

These constants combine to form the eight preference domains shown in Table 1 (page 19).

Table 1 Preference domains in precedence order

User name	Application ID	Host name
Current	Current	Current
Current	Current	Any
Current	Any	Current
Current	Any	Any
Any	Current	Current

User name	Application ID	Host name
Any	Current	Any
Any	Any	Current
Any	Any	Any

By providing your own string, you can access domains other than those defined by the current context. Access to the preferences of other users, though, requires special privileges.

Preferences located in domains higher on the list in Table 1 (page 19) (in other words, the more specific domains) take precedence over those located in lower domains (the more general domains).

The NSUserDefaults class, which is part of the Cocoa Foundation framework, is used by Cocoa applications to manage their user preferences. Because it writes preferences to the second domain listed in Table 1 (page 19), using the "Current" application domain, it cannot be used by a preference pane within a preference application. It would write any changed preferences to the preference application's preference file, instead of to the target application's file. If the preference pane is embedded in the target application, NSUserDefaults works, but it then breaks the modular and reusable design of the preference pane.

For details on how to read and write preferences to a preference file, see "Using Preference Services" (page 39). For more information about Preferences, see Preferences.

Configuration Files

When dealing with a cross-platform target application, such as standard BSD daemons, Core Foundation Preference Services cannot be used. Instead, you need to manipulate custom configuration files. For BSD applications, the file is normally a plain text file, but the specific format varies from program to program. Your preference pane needs to include its own code to read, parse, modify, and save these files rather than use Preference Services.

Interprocess Communication

Storing the new preferences on disk is not sufficient when the change applies to a running application or an integrated part of the operating system; you need to inform the target of the change if it is to take effect immediately instead of waiting for a restart. You achieve this by sending a message from the preference pane to the target application. In some cases the message may be a simple function call to an operating system routine.

In Mac OS X there is an abundance of ways to communicate with other processes; each layer and framework has its own preferred methods. Although preference panes are written using the Cocoa Objective-C framework, you are not restricted to its particular implementation of messaging—use what is best for the situation. When the target application is not under your control, your choice is limited to those methods understood by the target.

You could forego saving the preferences in the preference pane altogether if the changes are communicated directly to an omnipresent part of the system that manages its preferences itself. The preference pane could merely send the updated preferences to the target and the target would take responsibility for storing them in a persistent location (typically a file on the disk).

If the target is not guaranteed to always be running, the preference pane needs to update the preference file itself. Use the following communication methods only for auxiliary information or to notify the target, if it is currently running, that its preferences have changed.

Apple Events

At the highest level of interprocess communication are Apple events, the long-time standard for interapplication communication in Mac OS. An Apple event is a high-level message that an application can send to itself, other applications on the same computer, or applications on a remote computer. Apple event objects have a well-defined data structure with support for extensible, hierarchical data types. A well-defined set of Apple events can provide support for a rich scripting interface through AppleScript.

Apple events are the preferred method for applications to communicate with each other in Mac OS X. They are available to both Carbon and Cocoa applications through the Apple Event Manager.

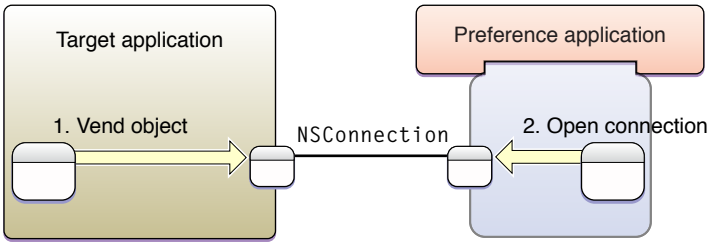
Apple events are beyond the scope of this document. For detailed information on using Apple events, see documentation on the Apple Event Manager in Carbon.

Distributed Objects

The Objective-C language runtime supports an interprocess messaging solution called "distributed objects." This mechanism enables a Cocoa application to call an object in a different Cocoa application. Calls may be synchronous, meaning the sending process is blocked while waiting for a reply from the receiver, or asynchronous, meaning no reply is expected and the sender is not blocked.

The receiving application "vends," or makes public, an object to which other applications can connect. Invoking one of the vended object's methods then takes place as if the object existed in your own application—the syntax does not change. The runtime system handles the necessary transmission of data between the applications.

Figure 1 Distributed object architecture



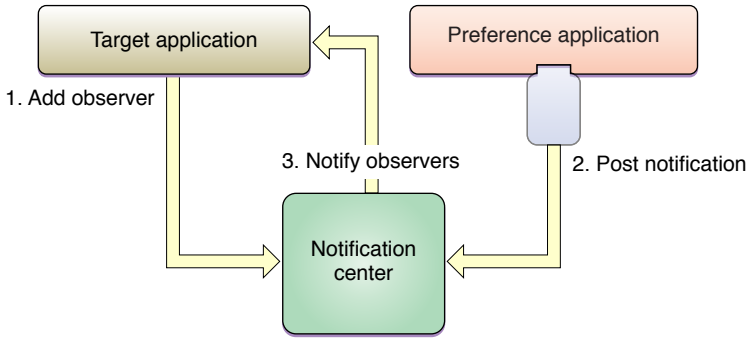
If your preference pane is to be used to control a faceless Cocoa application, this is a very simple technique for interapplication communication. Since it allows two-way communication, you can provide greater interaction between the user and the application. The preference application can obtain the current settings directly from the target application instead of a preference file. It can then get immediate feedback on whether the user's modifications are accepted by the target.

For details on how to use distributed objects in a preference pane, see "Using Distributed Objects" (page 41). For more information on distributed objects, see *Inside Mac OS X: The Objective-C Programming Language* and the NSConnection and NSDistantObject class descriptions in the Foundation Framework Reference.

Distributed Notifications

An alternative to the direct and bidirectional communication of distributed objects and Apple events is a one-way distributed notification. A distributed notification is a message posted by any application to a per-machine notification center, which in turn broadcasts the message to any applications interested in receiving it. Included with the notification is an identifier of the sender, and, optionally, a dictionary containing additional information. The receiver of the notification cannot communicate any information back to the sender. Figure 2 (page 22) illustrates this architecture.

Figure 2 Distributed Notification Model



The distributed notification mechanism is accessible through the Core Foundation CFNotificationCenter object and through the Cocoa NSDistributedNotificationCenter class. A Cocoa application, such as your preference application, can use either interface. A Carbon application, perhaps the target application, can use only the Core Foundation interface. A notification posted using one interface can be received by either.

Note: Core Foundation and Cocoa notifications are unrelated to the Notification Manager, which is part of Carbon.

A benefit of the distributed notification model is the model's one-to-many capabilities. If you have a suite of tools that share a common set of preferences, each running tool can register for and receive the same notifications for preference changes. Distributed notifications are also sent asynchronously. Your preference pane can post the notification and return immediately; you do not need to wait for the target application to receive the notification and finish processing it.

Further, the target application does not need to be running. If no application is listening for the notification, nothing happens. Because of this, distributed notifications are especially useful for notifying an application of a modified preference file. The preference pane can modify the preference file and post a notification about the change without being dependent on whether someone is listening to it.

Distributed notifications are a system-wide resource shared by all applications. To avoid name conflicts, select notification names that are certain to be unique to your application. See "Preventing Name Conflicts" (page 35) for details.

For details on how to use distributed notifications in a preference pane, see "Using Distributed Notifications" (page 41). For more information on distributed notifications, see the NSDistributedNotificationCenter class description in the Foundation Framework Reference.

Sockets and Ports

Cross-platform applications cannot make use of the above Mac OS X–specific techniques for interprocess communication. However, Mac OS X supports BSD sockets, a standard communication method on BSD platforms. You can make use of the standard POSIX socket APIs or take advantage of higher-level abstractions in the Cocoa class NSSocketPort or a Core Foundation CFSocket object. The production and parsing of the raw data stream sent over the sockets are the responsibilities of the preference pane object and the target application.

Signals

BSD signals are also available in Mac OS X. Signals are software interrupts that can be sent to a specific application. By default, the signal terminates the receiving application, but the application can override this by installing a signal handler that runs when a particular signal is received. The only information passed by the signal is a single integer identifying the signal.

Traditional BSD services (such as `inetd`) frequently use the predefined signal `SIGHUP` (hangup signal) to reset themselves. When modifying the preferences of one of these services, write the updated settings to the application's preference file and send the `SIGHUP` signal to the application. In response, the application can reread its preferences.

Life Cycle of a Preference Pane

Normally, the user interacts with a preference pane via the System Preferences application. It is the responsibility of the System Preferences application to load the preference pane bundle, create an instance of the principle class, and message the principle class object appropriately at various times during its life.

Preference panes can be used in other applications as well. For example, the Mac OS X Setup Assistant embeds the Date & Time preference pane in one of its windows.

Throughout this description, we'll refer to the container application, whether it be System Preferences, the Setup Assistant, or your own preference application, as simply "the application."

Instantiating the Preference Pane Object

The life of a preference pane begins when the application instantiates an `NSBundle` object for the preference pane package. The application then asks the `NSBundle` for its principle class and creates an instance of the principle class using the `initWithBundle:` method, passing in the `NSBundle` object as the argument.

The `initWithBundle:` method is the designated initializer for the `NSPreferencePane` class. Subclasses of `NSPreferencePane` that wish to perform their own initialization should override the `initWithBundle:` method, taking care to call the superclass's implementation first. For example:

```
- (id)initWithBundle:(NSBundle *)bundle
{
    if ( ( self = [super initWithBundle:bundle] ) != nil )
    {
        // add subclass-specific initialization here
    }
    return self;
}
```

At this point, the user interface elements of the preference pane (its main nib file and its main view) have not been loaded or initialized. Any initialization that depends on outlet connections to user interface elements in the main nib file should be deferred to the `mainViewDidLoad` method described below.

If your preference pane supports AppleScript commands, it should be prepared to respond to them at this point.

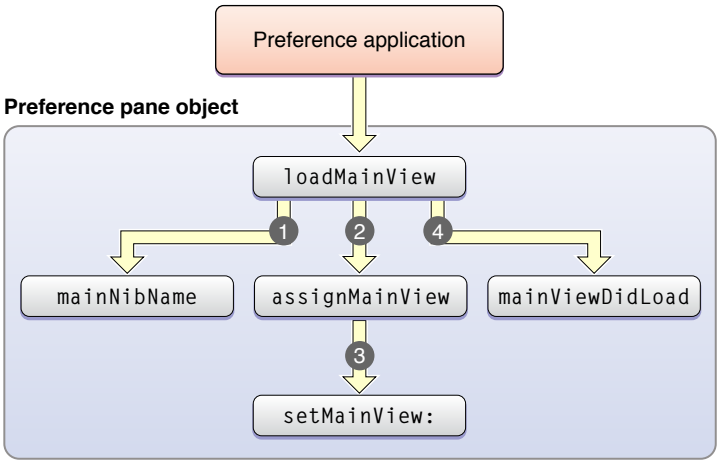
Loading the Main View

When the preference pane's user interface needs to be displayed for the first time, the application sends the `loadMainView` message to the preference pane object. The default implementation of `loadMainView` performs the following actions:

1. Determines the name of the main nib file by calling the preference pane object's `mainNibName` method.
2. Loads that nib file, passing in the preference pane object as the nib file's owner.
3. Invokes the preference pane object's `assignMainView` method to find and assign the main view.
4. Invokes the preference pane object's `mainViewDidLoad` method.
5. Returns the main view.

The sequence of methods invoked while loading the main view is illustrated in Figure 1 (page 26).

Figure 1 Execution flow of loadMainView



A preference pane subclass should rarely need to override the `loadMainView` method. One case in which an override is necessary is if a preference pane subclass needs to use a non–nib–based technique to load the main view, such as programatically creating the main view. In this case, the subclass's implementation of `loadMainView` must call `setMainView:` passing in the main view as the argument. This ensures that future calls to `mainView` will return the correct view.

Dynamically Selecting the Main Nib File

The default implementation of `mainNibName` returns the value of the `NSMainNibFile` key in the bundle's property list. If the key does not exist, the default value of `@"Main"` is returned. A NSPreferencePane subclass can override the `mainNibName` method if it needs to dynamically select the main nib file to use.

Assigning the Main View

The default implementation of `loadMainView` invokes the `assignMainView` method to find and assign the main view in the main nib file. The default implementation of `assignMainView` assigns the content view of `_window` to the `_mainView` outlet and retains the view. It then removes the content view from `_window`, releases `_window`, and sets `_window` to `nil`.

Most preference panes should not need to override the `assignMainView` method. The default implementation of `assignMainView` allows a preference pane developer to create the user interface for the preference pane in a window and connect the `_window` outlet to the window. If a preference pane has multiple main views and needs to select which main view to use at runtime, it can override the `assignMainView` method.

Note: This method was developed at a time when Interface Builder did not support views unless they were in windows.

Setting Up the User Interface

The preference pane object receives a `mainViewDidLoad` message after its main nib file has been loaded and the main view has been assigned. The default implementation of `mainViewDidLoad` in the NSPreferencePane class does nothing. A NSPreferencePane subclass can override this method if it needs to initialize the state of the view's graphical elements.

Selecting

Before a preference pane's main view is displayed in the application's window, the application sends the preference pane object a `willSelect` message. Immediately after the view is displayed, the application sends the preference pane a `didSelect` message.

The default implementations of these methods do nothing. An NSPreferencePane subclass should override these methods if it needs to perform some action either immediately before or immediately after a preference pane is selected.

Deselecting

The application attempts to deselect the currently selected preference pane when one of the following actions occur:

- the user attempts to switch to another view in the preference window
- the user attempts to close the preference window
- the user attempts to quit the application

The application attempts to deselect a preference pane by sending it the `shouldUnselect` message. The method returns one of the values from Table 1 (page 28), indicating whether the preference pane is willing to be deselected. The default implementation of `shouldUnselect` in the NSPreferencePane class returns `NSUnselectNow`. This tells the application that it is OK to deselect the preference pane immediately.

Important: Work that needs to be done should not be implemented as part of `willUnselect` or `didUnselect` and left to be executed when the preference pane is dismissed, as these methods will not be called when the user shuts down the device while sudden termination is enabled. 64-bit preference panes have sudden termination enabled by default, and 32-bit preference panes are encouraged to enable sudden termination.

Table 1 Return values of shouldUnselect

`NSUnselectCancel`	Cancel the deselection
`NSUnselectNow`	Continue the deselection
`NSUnselectLater`	Delay the deselection until the preference pane invokes `replyToShouldUnselect:`

A preference pane can override the `shouldUnselect` method if it needs to cancel or delay the deselection. Typically, this occurs if the preference pane needs to confirm saving changes with the user (as with the Network preference pane). If the mechanism of confirming the deselection is synchronous (such as with an application-modal alert or sheet), the `shouldUnselect` method should make the synchronous call and then return `NSUnselectCancel` or `NSUnselectNow`. For example:

```
- (NSPreferencePaneUnselectReply)shouldUnselect
{
    int result = NSRunAlertPanel( ... );

    if ( result == NSAlertDefaultReturn )
        return NSUnselectNow;
    return NSUnselectCancel;
}
```

If the mechanism of confirming the deselection is asynchronous (such as with a window-modal sheet), the `shouldUnselect` method should return `NSUnselectLater`. When a pane returns `NSUnselectLater`, it must call `replyToShouldUnselect:` once the pane decides whether or not it can be deselected. The `replyToShouldUnselect:` method takes one parameter, a Boolean value, that indicates whether or not the application should deselect the pane. A value of `YES` means the application should deselect the pane. A value of `NO` means the application should cancel the deselection.

Once the deselection is confirmed, the application sends the preference pane a `willUnselect` message immediately before the action that causes the deselection is performed. The application sends the preference pane a `didUnselect` message immediately after the action that caused the deselection is performed. When quitting the application, the `willUnselect` and `didUnselect` messages are both sent before the application quits.

Longevity of Preference Pane Objects in System Preferences

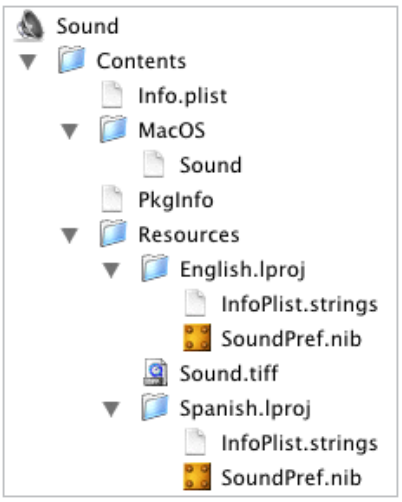
For performance reasons, the System Preferences application keeps preference pane objects around once they have been instantiated. They are not deallocated when the preference pane is deselected. They are only deallocated when the System Preferences application terminates.

Anatomy of a Preference Pane Bundle

A preference pane plug-in is packaged on disk as a bundle with the `.prefPane` extension. Like all bundle packages, a preference pane consists of an executable (in the Mac OS X native Mach-O format), an information property list (`Info.plist`), and localizable and global (nonlocalized) resources.

The structure of a sample preference pane bundle is shown in Figure 1 (page 29).

Figure 1 Contents of a preference pane bundle



When created within Xcode, the basic structure and the files are created for you. "Creating a Preference Pane Bundle" (page 47) describes the steps required to produce a working preference pane bundle. The following sections describe the individual elements of the bundle.

Information Property Lists

Every bundle contains a dictionary, the `Info.plist` file, that defines certain properties of the bundle, such as the names of important resources. Preference pane bundles should provide values for the following keys in the information property list:

Key	Description
`CFBundleIdentifier`	The unique identifier string for the bundle. Every bundle should have a unique `CFBundleIdentifier` prefixed by the reverse domain name of the organization. For example, "`com.apple.preference.sound`".

Key	Description
`NSMainNibFile`	The name of the main nib file. If this key is omitted, the default preference pane implementation assumes a value of "Main". The value must not include the `.nib` extension. For example, "`SoundPref`".
`NSPrefPaneIconFile`	The name of an image file resource used in the Show All view and favorites area of the System Preferences application to represent the preference pane. The icon should be 32 x 32 pixels in size. If this key is omitted, System Preferences looks for the `CFBundleIconFile` key. The value must include the extension. For example, "`Sound.tiff`".
`NSPrefPaneIconLabel`	The name of the preference pane displayed by System Preferences beneath the pane's icon and in the Pane menu. You can include a newline character in the string ("`\n`") to split a long name between two lines. If this key is omitted, System Preferences looks for the `CFBundleName` key. `NSPrefPaneIconLabel` should be localized via the `InfoPlist.strings` file. For example, "`Sound`" and "`Sonido`".
`NSPrincipalClass`	The name of the main controller class of the preference pane. This class must be defined in the Mach-O binary of the bundle and it must be a subclass of NSPreferencePane. To avoid symbol name collisions, the name of the class must be prefixed by a specially munged version of the bundle identifier (see "Preventing Name Conflicts" (page 35) for details). For example, "`ComApplePreferenceSoundPref`".

Localizable Resources

A bundle's resources can be localized to different languages and regions. Generally, these are resources that present text to the user, such as menu names and labels in windows. The resource files are stored in separate subdirectories in the `Contents/Resources` directory of the bundle. The directories are named after the language, such as `English.lproj` or `Spanish.lproj`. When your preference pane accesses a localized resource, such as a nib file containing a window, the operating system selects the version according to the user's language preferences.

The simplest way for a preference pane to define its user interface is through a main nib file. This nib file should be a localized resource. The name of the main nib file can be anything, but it must match the value of the `NSMainNibFile` key in the bundle's property list.

Like application bundles, preference pane bundles should include a localized `InfoPlist.strings` resource. This file contains individual strings the user sees, but cannot be stored within the nib file. This file should contain an entry for the `NSPrefPaneIconLabel` property whose value is the localized display name of the preference pane.

Global Resources

Not all resources need to be localized. Images without textual content can be used for all languages. These global resources are stored in the `Contents/Resources` directory.

The preference pane icon file (usually an `.icns` or `.tiff` file) is the 32 x 32 pixel icon used in the System Preferences application to represent the preference pane in the Show All view and the favorites area. The name of the preference pane icon file is specified by the `NSPrefPaneIconFile` key in the bundle's property list. Typically, this is a global (nonlocalized) resource. However, if the icon contains locale-specific information (such as text), it can be made localized.

Where Preference Panes Live

Preference pane bundles for System Preferences live in the `PreferencePanes` family of library directories. This family of directories consists of these directories:

Directory	Description
`/System/Library/PreferencePanes`	Mac OS X built-in preference panes
`/Network/Library/PreferencePanes`	Third-party preference panes available to all users on the network
`/Library/PreferencePanes`	Third-party preference panes available to all users on the computer
`~/Library/PreferencePanes`	Third-party preference panes available only to the current user

System Preferences searches these directories in the reverse order that they are listed here. If multiple preference panes are found with identical bundle identifiers (`CFBundleIdentifier` key value), only the first preference pane found is displayed.

When creating a custom preference application or if you use preference panes to implement the Preferences menu item of the target application, store the preference pane bundles inside the application's bundle in the `Resources` directory. If the preference pane needs to be shared by a suite of applications, store the preference pane bundles in a subdirectory in `/Library/Application Support`.

Updating Preference Panes for Snow Leopard and Beyond

Snow Leopard introduces several system-wide features that preference panes should support. Preference panes should be 64-bit, utilize garbage collection, and support sudden termination. For transition purposes, Snow Leopard will support 32-bit preference panes from developers outside of Apple. For 32-bit preference panes, garbage collection is not required, and support for sudden termination is "opt-in." For 64-bit preference panes, garbage collection is a required feature and sudden termination is enabled by default. It is strongly recommended that all new preference panes be 64-bit, as support for 32-bit preference panes is not guaranteed in the future.

Making Your Preference Pane 64-Bit

Starting with Snow Leopard, preference panes should be 64-bit programs. In the future, only 64-bit versions of preference panes will be supported. For transitional purposes, however, Snow Leopard supports 32-bit preference panes as well.

Because you probably want your preference pane to work under earlier versions of the Mac OS, and because Snow Leopard can be run on 32-bit machines, you probably want to release your preference pane as a dual binary for Snow Leopard, in both 32-bit and 64-bit versions.

If you are providing a dual binary, your preference pane is more like a framework than a stand-alone application, in that it can be called by 32-bit or 64-bit applications.

See *64-Bit Transition Guide for Cocoa* for a description of how to write your preference pane as a 64-it program, and how to provide a dual binary that can be called from 32-bit or 64-bit applications.

Using Garbage Collection

All 64-bit preference panes are expected to use garbage collection. Using garbage collection will, in most cases, simplify your code and reduce the likelihood of memory leaks.

In Snow Leopard, the System Preferences application will run 64-bit preference panes with garbage collection enabled, and 32-bit panes with garbage collection disabled.

See *Garbage Collection Programming Guide* for a description of how to implement garbage collection in Cocoa programs and for the compiler directives to produce working code for dual binaries. You still need to retain and release objects in your 32-bit code, but with the proper compiler directives, the compiler will ignore these statements in the 64-bit version.

Supporting Sudden Termination

To make shutting down the Mac faster and more convenient, Snow Leopard introduces sudden termination. When the user shuts down the device, applications that support sudden termination are simply killed, instead of being told that they are being quit and allowed to complete unfinished tasks.

It is important that you disable and re-enable sudden termination in parts of your code that have unfinished work that must be completed before your program terminates. In particular, the methods `willUnselect` and `didUnselect` should not be relied on to complete work at shutdown.

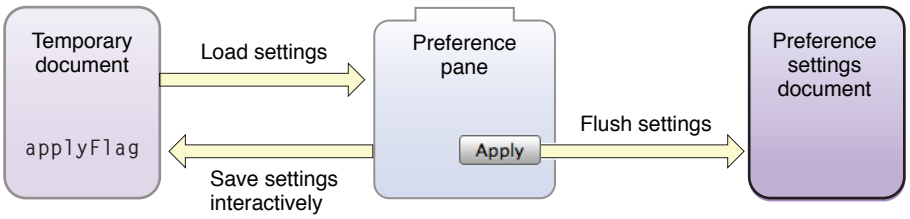
To disable sudden termination temporarily, call the `disableSuddenTermination` method in `NSProcessInfo`. When your pane is in a state that allows it to be safely terminated by `SIGKILL`, call `enableSuddenTermination`. You can nest calls to disable sudden termination, or disable and enable sudden termination on a background thread: sudden termination is not enabled until all calls to disable it have been balanced by a call to enable it again.

Ideally, your preference pane should update the preferences file each time the user makes a change in the pane, so no work needs to be done at shutdown.

For complex groups of preferences that need to be changed as a set, changes should be saved to a temporary document as they are modified, and an Apply button should be provided to flush the settings to the actual preferences.

The temporary document should include a flag to indicate that the settings have been applied. When the pane loads, it should load its settings from this temporary document, and set the Apply button active if the settings have not yet been applied. If you follow this recommendation, your pane will not need to disable sudden termination when the user makes changes, as no work will need to be done at shutdown. See Figure 1 for an illustration of this technique.

Figure 1 Figure



64-bit preference panes have sudden termination enabled by default. 32-bit preference panes can opt-in to sudden termination by setting the boolean value of `NSSupportsSuddenTermination` to `true` in the preference pane's `.plist` file.

Preventing Name Conflicts

The Objective-C runtime provides only a single flat, global name space per process for all exported symbols. This includes all global variables, nonstatic functions, class names, and categories declared for individual classes; protocols have a separate global name space of their own.

Because preference pane plug-ins from different vendors must coexist in the same process, you must follow conventions to avoid symbol name collisions. Every exported symbol in a preference pane plug-in must be prefixed with an identifier unique to the plug-in. This requirement is not circumvented by unloading each plug-in before loading the next one. Once an Objective-C symbol (class names, protocols and categories) gets loaded, it cannot be unloaded.

Uniqueness Algorithm

Your preference pane plug-in should derive its unique prefix from its bundle identifier using the following algorithm:

1. Start with the bundle identifier (`com.apple.preference.sound`)
2. Capitalize the first letter of each period-separated component (`Com.Apple.Preference.Sound`)
3. Remove the periods (`ComApplePreferenceSound`)

Note that this convention depends on the uniqueness of each bundle identifier. To guarantee uniqueness of the bundle identifier, each organization should prefix its identifiers with its reverse-ordered ICANN domain name (for example, "`com.apple`").

Each organization should institute its own processes and conventions to avoid bundle identifier collisions among bundles developed within the organization.

To avoid having to use the full, prefixed symbol names in source code, you can create shorthand preprocessor macros. These macros can be defined in a single header file that is imported into every source file. For example:

```
#define SoundPref ComApplePreferenceSoundPref
#define AlertController ComApplePreferenceSoundAlertController
#define MicrophoneController ComApplePreferenceSoundMicrophoneController
```

Obviously, these shortcuts are only valid in Objective-C source files that include the header file. References to class names outside of such source files (for example, in the bundle property list and in the main nib file) must specify the full, real name.

Categories

Preference pane plug-ins should avoid using Objective-C categories to override methods of classes in public frameworks. If multiple panels attempt to override the same method of the same class, only one override takes effect, leading to unpredictable behavior.

Wrapping Long Labels

Labels are not automatically wrapped by the system, so very long labels must be manually tagged so that they wrap as desired.

Insert the necessary tags to split a long label into two lines into your preference pane's `.plist` file, as described in *Information Property List Key Reference*. If you need additional guidance, see *Property List Programming Guide*.

Using Preference Services

Core Foundation Preference Services provides functions for reading and writing preferences to and from any available preference domain (see "Preference Services" (page 19)). Preference data are stored in property lists as a series of key-value pairs. The key is a string identifying the preference. The value is the preference setting, which can be any of the following data types: CFData, CFString, CFArray, CFDictionary, CFDate, CFBoolean, and CFNumber. When the value is a CFArray or CFDictionary, each of its elements must be one of the allowed data types. Except for CFBoolean (which has no equivalent object), each of the Core Foundation types are interchangeable with their Cocoa equivalents (NSData, NSString, and so forth).

Reading and writing preferences look like this:

```
#include <CoreFoundation/CFPreferences.h>

CFStringRef appID, userName, hostName; // Assigned elsewhere
CFStringRef key = CFSTR("PrefKey");
CFPropertyListRef value; // Any allowed data type

value = CFPreferencesCopyValue(key, appID, userName, hostName);
CFPreferencesSetValue(key, value, appID, userName, hostName);
```

If a key does not exist in the given domain, the `CFPreferencesCopyValue` function returns `NULL`. Conversely, passing `NULL` to `CFPreferencesSetValue` for the key's value removes the key from that domain. For performance reasons, changes made to a domain are cached. To force changes to be flushed to disk, call `CFPreferencesSynchronize` for the particular domain. This can be an expensive operation since it requires accessing the disk, so do not synchronize too often.

Several convenience functions automatically search through the domains of a particular application for a requested key and associated value. The search proceeds from the most specific domain—current user, the particular application, current host—at the top of Table 1 (page 19) to the most general domain—any user, any application, any host—at the bottom of the table, until a matching key is found. When writing a value, it is stored in the application's default domain: current user, the particular application, any host. Use of these routines looks like this:

```
value = CFPreferencesCopyAppValue(key, appID);
CFPreferencesSetAppValue(key, value, appID);
```

If a suite of applications share certain preferences, they can be stored together in their own set of domains defined by a suite ID, similar to the application ID. The suite domains can be added to the search path of the `CFPreferencesCopyAppValue` function with the `CFPreferencesAddSuitePreferencesToApp` function. Multiple suites of domains can be added. After searching each application-specific domain, Preference Services searches the corresponding suite-specific domains before searching the general domain. Preferences are still stored in the application's default domain when using these functions, though.

For user-specific and application-specific preferences, these functions should suffice. For system-level preferences, you need to use the more general functions.

Note: Property lists, which are used to store the preference data, are intended for relatively small amounts of data (less than a few hundred kilobytes). If you need to store large amounts of data, especially CFData or NSData objects, consider storing the larger preferences in regular files instead of property lists.

Communicating With the Target Application

In some situations you need to communicate directly with the target application rather than rely solely on the preference file. This section describes how to use two methods—distributed objects and distributed notifications—to achieve this.

Using Distributed Objects

Distributed objects allow one application to communicate with an object in another application. You can use distributed objects only in Cocoa applications.

The application that owns the object, the target application in this case, makes the object available to other applications by vending the object with code such as

```
id serverObject; // Assume this exists.
NSConnection *theConnection;

theConnection = [NSConnection defaultConnection];
[theConnection setRootObject:serverObject];
[theConnection registerName:@"MyServer"];
```

where `serverObject` is an object that defines a set of methods for external use. These methods can provide low-level "get" and "set" accessors for the application settings or higher-level queries and requests. To gain access to the object, another application, such as your preference pane, executes code such as

```
NSConnection *theConnection;
id remoteObject;

theConnection = [NSConnection connectionWithRegisteredName:@"MyServer"
            host:nil];
remoteObject = [[theConnection rootProxy] retain];
```

where `remoteObject` is now a proxy object representing the vended object. Interaction with the object occurs normally:

```
x = [remoteObject defaultWidth];
[remoteObject setBackgroundColor:[NSColor redColor]];
```

Using Distributed Notifications

Distributed notifications allow an application to broadcast a message to any number of other applications without needing to know who those other applications are, or even if the other applications exist. Every application type—Cocoa, Carbon, BSD—can use distributed notifications.

An application, the target application in this case, expresses an interest in receiving a broadcasted message by registering itself with the system's distributed notification center, identifying exactly what message, or notification type, it wants to receive. The notification type is defined by an arbitrary string agreed upon by the sender and receiver of the notification. As an example, Cocoa's NSWindow class defines the notification type `@"NSWindowDidCloseNotification"`, which an NSWindow instance broadcasts when its window closes. Any other object can register to receive this notification. (This notification, however, is internal to a single application and is not distributed to the rest of the system.)

In addition to the message, the application can identify the particular object sending the message. When the sender and receiver are in the same application—in other words, using nondistributed notifications—the observed object can be anything. When using distributed notifications, though, the object must be a string (CFString or NSString). A useful choice for the observed string is the bundle identifier of the target application.

In registering for the notification, the application provides a callback to be executed when it receives the notification. The application then proceeds with its duties. To receive the notification, the application must enter a Core Foundation run loop. This occurs for both the Cocoa run loop and Carbon event manager.

To register to receive a notification, a Cocoa application executes code such as the following:

```
NSString *observedObject = @"com.apple.example.PrefpaneTarget";
NSDistributedNotificationCenter *center =
            [NSDistributedNotificationCenter defaultCenter];
[center addObserver: self
            selector: @selector(callbackWithNotification:)
            name: @"My Notification"
            object: observedObject];
```

The observer argument is the object on which the callback method is invoked. The callback method, identified by the `selector` argument and implemented by the observer object, has a signature of

```
- (void)callbackWithNotification:(NSNotification *)myNotification;
```

The NSNotification object passed to this method contains the specific object and notification message received.

The analogous code for a Carbon or BSD application using Core Foundation is

```
void *observer;
CFStringRef observedObject =
            CFSTR("com.apple.example.PrefpaneTarget");
CFNotificationCenterRef center =
            CFNotificationCenterGetDistributedCenter();
CFNotificationCenterAddObserver(center, observer, myCallbackFntn,
            CFSTR("My Notification"), observedObject,
            CFNotificationSuspensionBehaviorDeliverImmediately);
```

with a callback function prototype of

```
(void)myCallbackFntn(CFNotificationCenterRef center, void *observer,
            CFStringRef notificationName, const void *observedObject,
            CFDictionaryRef userInfo);
```

Because the callback is a function instead of a method invocation, the `observer` argument is any additional data (in the form of a pointer) that you want to pass to the callback function.

Next, the broadcasting application—your preference pane—sends the notification. It calls the system's notification center, tells the center what notification to send, and optionally passes a dictionary containing additional information. The dictionary can be used to pass the modified preferences directly to the application. Or, the preference pane can choose not to use the dictionary and instead write the changes out to disk. The notification is then used to tell the application to update its preferences from the disk.

Cocoa code to send the notification looks like this:

```
NSString *observedObject = @"com.apple.example.PrefpaneTarget";
NSDistributedNotificationCenter *center =
            [NSDistributedNotificationCenter defaultCenter];
[center postNotificationName: @"My Notification"
            object: observedObject
            userInfo: nil /* no dictionary */
            deliverImmediately: YES];
```

The Core Foundation code looks like this:

```
CFStringRef observedObject =
            CFSTR("com.apple.example.PrefpaneTarget");
CFNotificationCenterRef center =
            CFNotificationCenterGetDistributedCenter();
CFNotificationCenterPostNotification(center, CFSTR("My Notification"),
            observedObject, NULL /* no dictionary */, TRUE);
```

The notification center looks up all the applications that registered to receive the given notification type from the particular `observedObject`. It then notifies each application's run loop of the notification and gives it a copy of the dictionary. The selected callback function or method is executed during the application's next pass through its run loop.

When using Preference Services, be certain to flush changes to the disk with the appropriate synchronize functions before sending notifications of changes. Otherwise, due to the caching performed by Preference Services, the disk may not accurately reflect the changes when the target receives the notification. Likewise, the target application must resynchronize its preferences after receiving the notification.

Implementing a Preference Pane Help Menu

This section takes you through the steps to add context-sensitive Help menu entries for a preference pane. You can implement Help Viewer anchors in two different ways, both which can provide your user with specific assistance accessible from their Help menu. If you have already created a skeletal preference pane as described in "Creating a Preference Pane Bundle" (page 47), you can use it here. Or, you can create a fresh preference pane and refer to the following instructions where appropriate.

Adding Global Help Menu Items

One method of offering help to your preference pane user is by adding help menu items that become available whenever your preference pane is loaded. Upon selection of your preference pane, the System Preferences application will automatically add the Help menu items as described by a static array in the pane's `Info.plist` file. Once the preference pane is deselected, the System Preferences application will remove them from the menu.

First, you need to create some help material, unless you are linking to existing material on the system. If you are not familiar with creating content for the Mac OS X Help Viewer, refer to Providing User Assistance with Apple Help.

Second, you need to add entries to the array values of the `NSPrefPaneHelpAnchors` key in your `Info.plist` file. You will statically define each anchor with an associated title. The title represents the string value of the menu item which will display the help book specified by the anchor. For example, if you wanted to add a "Sample Help" item in the Help menu for your preference pane, which will open a book marked by the `SMPL001` anchor, add this to your `Info.plist` file:

Listing 1 Info.plst entry for the SMPL001 Help menu item

```
<key>NSPrefPaneHelpAnchors</key>
<array>
    <dict>
        <key>title</key>
        <string>Sample Help</string>
        <key>anchor</key>
        <string>SMPL001</string>
    </dict>
</array>
```

Once the preference pane is loaded by the System Preferences application, a new menu item in the Help menu called "Sample Help" will appear. When selected, it will load the Help book specified by the `SMPL001` anchor.

You can also localize the Help menu item for your preference pane. In the example above, you would replace the "Sample Help" string with some internal string, such as `SAMPLE_PREFPANE_MENU_TITLE`. Then in the Localizable.strings file for all your languages, you would add the appropriate entry to override this internal string:

Listing 2 English Localizable.strings entry for the SMPL001 Help menu item

```
"SAMPLE_PREFPANE_MENU_TITLE" = "Sample Help";
```

Listing 3 French Localizable.strings entry for the SMPL001 Help menu item

```
"SAMPLE_PREFPANE_MENU_TITLE" = "Aide Sample";
```

Adding Dynamic Help Menu Items

Often, you want the items in your preference pane's Help menu to show or hide based off context—for example, if you have multiple subpanes in your preference pane's main view, you may want to add a Help menu item for one subpane, and hide the menu items for the others.

You can accomplish this with the method `updateHelpMenuWithArray:`. This method, implemented in the PreferencePanes framework, is called with one argument, an array of dictionaries corresponding to the same format as the array and dictionaries in "Adding Global Help Menu Items" (page 45). Instead of adding the items statically in the `Info.plist` file, you create dynamic NSArray and NSDictionary objects and pass those into this method, which will update the Help menu accordingly.

The code in Listing 4 shows you how to construct a help menu that changes its menu item title based off the identifier of the selected tab.

Listing 4 Dynamic help menu for a tab view

```
- (void)tabView:(NSTabView *)tabView didSelectTabViewItem:(NSTabViewItem
*)tabViewItem
{
    NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
        [tabViewItem identifier], @"title",
        @"SMPL001", @"anchor",
        NULL];

    NSArray *array = [NSArray arrayWithObject:dictionary];

    [self updateHelpMenuWithArray:array];
}
```

Note that the title string used in this method call is not localizable with the Localizable.strings file. If you want to localize the title of the help menu item, you must do so programatically or using identifiers from the language-specific nib file, demonstrated in the example above.

Creating a Preference Pane Bundle

This section takes you through the steps to create and install a preference pane bundle for use by System Preferences. You need to perform these actions for every preference pane you create. It is assumed that you are already familiar with Xcode and Interface Builder. For help using these development tools, see the Currency Converter tutorial for Cocoa.

Important: Xcode has a Preference Pane template project. Selecting this template instead of the generic Cocoa Bundle in step 3 of "Create the Project" (page 47) renders most of the remaining directions in this section unnecessary. The project already contains a nib file and skeleton source files; you can immediately build and install the preference pane according to "Build and Install the Preference Pane" (page 49). You should still read these directions and customize settings, such as the bundle identifier, where appropriate. Before moving onto the next section and constructing a user interface, though, you need to inform the nib file of your custom `NSPreferencePane` subclass by dragging the class's header file to the nib file window in Interface Builder and setting the File's Owner to your subclass.

Create the Project

This section describes how to create the preference pane project and add the Preference Panes framework.

1. Start Xcode.
2. Choose New Project from the File menu.
3. Select the Cocoa Bundle project type and create the project.
4. Choose Add Frameworks from the Project menu. If the selection is not already there, go to the `/System/Library/Frameworks` directory. Select `PreferencePanes.framework`.

Create the Nib File

This section describes how to create a simple preference pane nib file and add it to the preference pane project.

1. While Xcode is still running with your project open, start Interface Builder.
2. Create an empty Cocoa nib.
3. Create a window and resize it to a suitable size. For System Preferences, the window should not be more than 595 pixels wide. As the window itself is not used by the preference pane, only its contents, you do not need to specify a window title nor localize the title.

4. Build the user interface in the window.
5. Return to Xcode and locate the `NSPreferencePane.h` header file in `PreferencePanes.framework`. Drag the header file to the nib's main window in Interface Builder.
6. In the Classes pane, select the NSPreferencePane class and create a subclass of it. Rename it to whatever you want. This is a global property within the preference application, so include a unique prefix in the name as described in "Preventing Name Conflicts" (page 35).
7. With the subclass selected, go to the Attributes pane of the Info window. Create any outlets or actions you need for the user interface.
8. In the Instances pane, select the File's Owner object. In the Custom Class pane of the Info window, select your preference pane class.
9. Draw a connection (Control-drag) between the File's Owner object and the window object. Connect the window to the `_window` outlet.
10. Connect the remaining outlets and actions needed for the user interface.
11. Save the nib file into the `English.lproj` directory of your project. When asked whether to add it to the project, click the Add button.

Create the Preference Pane Files

This section describes how to create the initial source files and to insert the preference pane's icon into the project.

1. In Interface Builder, with the nib file open, click the Classes tab and select your preference pane subclass.
2. Choose Create Files from the Classes menu. Save the files in your project folder and make sure the "Insert into targets" checkbox is checked.
3. In Xcode, edit the header file of your preference pane subclass. After the line importing `Cocoa.h`, add the line

   ```
   #import <PreferencePanes/NSPreferencePane.h>
   ```

4. Add your preference pane icon to the project's Resources folder.

Update the Build Settings

This section describes how to modify the default project settings to produce a custom preference pane bundle. This mostly involves assigning values to the necessary keys in the bundle's information property list.

1. Choose Edit Active Target from the Project menu and go to the Bundle Settings pane.

2. Change the "Identifier" field to an appropriate unique value for the `CFBundleIdentifier` key. The value should be prefixed by the reverse domain name of your organization (see "Preventing Name Conflicts" (page 35)).
3. Change the "Principal class" field in the Cocoa-specific section to the name of your preference pane subclass. This is the `NSPrincipalClass` key.
4. Change the "Main nib file" field to the name of your nib file. Do not include the `.nib` extension. This is the `NSMainNibFile` key.
5. Enter Expert mode by clicking the Expert button at the top of the Bundle Settings window. Create a new key by clicking the New Sibling button. Rename the new key `NSPrefPaneIconFile` and set its value to the name of your icon file.
6. Go to the Build Settings pane, scroll to the bottom of the window, and change the `WRAPPER_EXTENSION` entry value to `prefPane`.
7. Select the `InfoPlist.strings` file in the project's Resources folder. Update the `CFBundleName` value if it should be different from the project name. Alternatively, you can add an entry for `NSPrefPaneIconLabel`, if you need to split the name between two lines.

Build and Install the Preference Pane

This section describes how to make the preference pane available to System Preferences.

1. Build the project.
2. In Finder, locate the `build` directory for the project. The default location is inside the project folder. The preference pane is in this folder.
3. Move the preference pane into one of the `PreferencePanes` family of folders listed in "Where Preference Panes Live" (page 31). For testing, use the `PreferencePanes` folder in `~/Library`. You may need to create the `PreferencePanes` folder.

When you run System Preferences you should now see your preference pane at the bottom of the window in the "Other" category.

Implementing a Simple Preference Pane

This section takes you through the steps to create a simple preference pane that interacts with the user preference system. The preference pane stores and retrieves a pair of values using Core Foundation Preference Services. If you have already created a skeletal preference pane as described in "Creating a Preference Pane Bundle" (page 47), you can use it here. Or, you can create a fresh preference pane and refer to the following instructions where appropriate.

Build the User Interface

The preference pane created in this section consists of a text field and a checkbox illustrating the handling of string and Boolean preferences.

1. Open the nib file in Interface Builder.
2. Drag a text field into the window. Label the field "A string value".
3. Drag a checkbox into the window. Change its label to "A Boolean value".
4. In the Classes pane of the main window, select your preference pane subclass.
5. In the Attributes pane of the Info window, add two outlets named `theTextField` and `theCheckbox`.
6. Add an action named `checkboxClicked:`.
7. In the Instances pane of the main window, make connections between the File's Owner object (representing your preference pane subclass) and the text field and checkbox, connecting them to the `theTextField` and `theCheckbox` outlets.
8. Make a connection between the checkbox and the File's Owner object, connecting the `checkboxClicked:` target-action method.
9. Save the nib file.
10. With your subclass highlighted in the Classes pane, choose Create Files from the Classes menu. Save the files into your project, overwriting if necessary.

Update the Header File

The default preference pane header created by Interface Builder requires a few additions.

1. In Xcode, select the preference pane's header file.

2. After the line importing `Cocoa.h` add the lines

```
#import <PreferencePanes/NSPreferencePane.h>
#import <CoreFoundation/CoreFoundation.h>
```

3. Update the outlet declarations to be

```
IBOutlet NSButton    *theCheckbox;
IBOutlet NSTextField *theTextField;
```

4. Add a new instance variable to hold the application ID of the target application:

```
CFStringRef appID;
```

Implement the initWithBundle: Method

The preference pane is initialized using the `initWithBundle:` method. Only the `appID` instance variable needs to be initialized here, but when overriding an `init` method, you also need to call the superclasses implementation. Add the following code to the preference pane's implementation file.

```
- (id)initWithBundle:(NSBundle *)bundle
{
    if ( ( self = [super initWithBundle:bundle] ) != nil ) {
        appID = CFSTR("com.apple.example.prefPaneSample");
    }

    return self;
}
```

Implement the mainViewDidLoad Method

Immediately after the nib file has been loaded, the object receives a `mainViewDidLoad` message from the default implementation of `loadMainView`. Here you should initialize the user interface elements to reflect the current preference settings. Add the following code to the implementation file.

```
- (void)mainViewDidLoad
{
    CFPropertyListRef value;

    /* Initialize the checkbox */
    value = CFPreferencesCopyAppValue( CFSTR("Bool Value Key"),  appID );
    if ( value && CFGetTypeID(value) == CFBooleanGetTypeID()  ) {
        [theCheckbox setState:CFBooleanGetValue(value)];
    } else {
        [theCheckbox setState:NO];
    }
    if ( value ) CFRelease(value);

    /* Initialize the text field */
    value = CFPreferencesCopyAppValue( CFSTR("String Value Key"),  appID );
    if ( value && CFGetTypeID(value) == CFStringGetTypeID()  ) {
```

```
        [theTextField setStringValue:(NSString *)value];
    } else {
        [theTextField setStringValue:@""];
    }
    if ( value ) CFRelease(value);
}
```

For each of the two preferences being used, `mainViewDidLoad` requests the preference's value from Core Foundation Preference Services. If a value is found for the preference (`value` is not `NULL`) and the value is of the correct data type, the preferences value is used to set the value of the appropriate user interface element. If the value does not exist, it initializes the elements with default values.

Implement the checkboxClicked: Method

When the user clicks the checkbox, it sends an action message to the preference pane object. The `checkboxClicked:` method obtains the new state of the checkbox and stores it under the name "Bool Value Key". Add the following code to the implementation file; an empty method definition should have been created by Interface Builder.

```
- (IBAction)checkboxClicked:(id)sender
{
    if ( [sender state] )
        CFPreferencesSetAppValue( CFSTR("Bool Value Key"),
                    kCFBooleanTrue, appID );
    else
        CFPreferencesSetAppValue( CFSTR("Bool Value Key"),
                    kCFBooleanFalse, appID );
}
```

Implement the didUnselect Method

When the preference pane gets deselected, either because the application is exiting or another preference pane is selected, it is sent a `didUnselect` message. In this method you want to extract the user's preferences and save the changes to disk. Since the checkbox gets recorded whenever the user clicks it, only the text field needs to be updated here. After flushing the preferences to the disk, `didUnselect` broadcasts a notification. The notification assumes the target application is implemented to receive this notification and update its preferences while it is running. Add the following code to the implementation file.

```
- (void)didUnselect
{
    CFNotificationCenterRef center;

    CFPreferencesSetAppValue( CFSTR(“String Value Key”),
                [theTextField stringValue], appID );
    CFPreferencesAppSynchronize( appID );

    center = CFNotificationCenterGetDistributedCenter();
    CFNotificationCenterPostNotification(center,
            CFSTR("Preferences Changed"), appID, NULL, TRUE);
}
```

Using Preference Panes in Other Applications

The NSPreferencePane class is not restricted to use only by System Preferences. Your own application can use it as well. You can reuse preference panes intended for System Preferences just like the Mac OS X Setup Assistant does with the Date & Time preference pane. Or, you can write preference panes for use exclusively by your own application.

Embedding a Single Pane

Embedding a preference pane into your own application is largely a matter of adding the preference pane's main view into a window and sending the proper information messages to the preference pane object. The preference pane object is responsible for accessing and saving user preferences. The procedure is as follows.

1. Initialize the preference pane. If you have the path to the preference pane bundle, load and initialize the preference pane object with the following lines of code.

   ```
   NSBundle *prefBundle = [NSBundle bundleWithPath: pathToPrefPaneBundle];
   Class prefPaneClass = [prefBundle principalClass];
   NSPreferencePane *prefPaneObject = [[prefPaneClass alloc]
               initWithBundle:prefBundle];
   ```

   For preference panes stored within the application's bundle, use one of the NSBundle `pathForResource` methods to obtain the path to the preference pane. For example, if the preference panes are stored in a subdirectory named `PreferencePanes` in the application's `Resources` directory, the full path can be obtained using

   ```
   pathToPrefPaneBundle = [[NSBundle mainBundle]
               pathForResource: @"NameOfPane" ofType: @"prefPane"
               inDirectory: @"PreferencePanes"];
   ```

2. Select the preference pane. When you are ready to display the preference pane, send it a `loadMainView` message. Its return value is the preference pane's main view if successful; on failure it returns `nil`. Next, notify the preference pane that it is about to be displayed by sending it a `willSelect` message. Because this method may potentially alter the preference pane's main view, get the main view again with the `mainView` message. Now add the view into your window. Center the preference pane view horizontally, but resize the window vertically to accommodate the view. Finally, send the preference pane object a `didSelect` message.

   The code for selecting the preference pane looks like the following.

   ```
   NSView *prefView;
   if ( [prefPaneObject loadMainView] ) {
       [prefPaneObject willSelect];
       prefView = [prefPaneObject mainView];
       /* Add view to window */
       [prefPaneObject didSelect];
   } else {
   ```

```
    /* loadMainView failed -- handle error */
}
```

3. Deselect the preference pane. The application is required to deselect the preference pane before any of the following actions occur:

    - the window switches to a different view
    - the preference pane's window closes
    - the application quits

    The application is required to release the object only when quitting the application; for the other events, the preference pane object can be reselected at a later time.

    The first step is to ask the preference pane object whether it is willing to be deselected by sending it a `shouldUnselect` message. The object can refuse to be deselected if one of the user preferences has an unacceptable value. The method returns one of the values from Table 1 (page 28) to indicate its willingness to be deselected. If the preference pane object returns `NSUnselectLater`, it is indicating that it needs to obtain some more information from the user before it knows what action to take. When the preference pane object is ready, it posts one of the following two notifications to indicate whether it is now OK to continue or you should abort the deselection.

`NSPreferencePaneDoUnselectNotification`	Do the deselection
`NSPreferencePaneCancelUnselectNotification`	Cancel the deselection

    When `NSUnselectLater` is returned, register for these two notifications coming from the preference pane object and temporarily abort the deselection. Continue as appropriate after receiving one of these notifications.

    When the preference pane object indicates it can be deselected, send it a `willUnselect` message. Next, perform the appropriate action causing the deselection: remove the view, close the window, or prepare to exit. Finally, send the object a `didUnselect` message.

    If you do not expect to use the preference pane object again in your application, release it to reclaim the memory resources it consumed.

Managing a Collection of Panes

A large application probably has a large number of preferences. Although you can use an NSTabView to associate closely related preferences into a single preference pane object, you may need to create multiple preference panes. Your application needs to provide a user interface for selecting the panes.

If you have a small number of preference panes, place their icons into a fixed-size view at the top of the window and place the preference panes' views into the bottom of the window as each is selected. See the Mail application for an example.

If you have more preference panes than can fit into the width of your window, provide an additional "Show All" icon in the top–left corner. The icon should be the application icon to which the preferences are applied (not necessarily the preference application's own icon). Selecting this icon presents a two-dimensional matrix

of all the preference pane icons from which the user can select. Also provide a view to the right of the "Show All" icon into which users can drag their favorite preference panes. The user's favorite preference panes should be stored as part of the preference application's own user preferences. See the System Preferences application for an example.

Document Revision History

This table describes the changes to *Preference Pane Programming Guide*.

Date	Notes
2008-10-15	Updated for Mac OS X v10.6: 64-bit, garbage collection, sudden termination. Added a section on wrapping long labels. Changed title from "Preference Panes" to "Preference Pane Programming Guide."
2005-04-29	Fixed an incorrect instruction and replaced a duplicate graphic.
	Added article on using new custom Help menu items.
2003-08-12	Updated tabbed view screen shot in “The Preference Application” article.
2003-02-24	Revision history was added to existing topic. It will be used to record changes to the content of the topic.