Extending Printing Dialogs

Printing > Carbon

2006-10-03

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

Contents

Figures, Tables, and Listings

Introduction to Extending Printing Dialogs

As printer vendors and application developers extend the printing capabilities of their hardware and software products, they need a way to extend the Mac OS X printing system to make new printing features available to their customers. To address this need, Mac OS X has introduced the **printing plug-in**—a component architecture based on Core Foundation Plug-in Services.

There are four types of printing plug-ins in Mac OS X:

- **I/O modules** are used by the printing system to communicate with a printer using a standard transport-layer interface, such as AppleTalk or TCP/IP.
- **Printer browsers** provide a way for people to discover available local and network printers.
- **Printer modules** are used by the printing system to convert the graphics content in a print job for output to a specific printer or family of printers.
- **Printing dialog extensions** provide a way for people to view and change the settings for a set of related printing features. The user interface of a printing dialog extension is a pane in one of the printing dialogs.

This book is a guide to developing printing dialog extensions—including basic concepts, theory of operation, and a documented Carbon-based sample project.

Why Read This Book

You should read this book if:

- You're developing an application or a printer module, and you want to learn how to present your printing features using the Mac OS X printing dialogs.
- Your existing software already extends a printing dialog, and you want to improve the interface or take advantage of advanced features in the Mac OS X printing system.
- You want to see what's involved in writing a printing plug-in.

Organization of This Document

The first part covers the basic concepts. Both Carbon and Cocoa developers will benefit from reading these chapters:

- "Printing Features and Printing Dialog Panes" (page 15) discusses standard printing features, printing dialog panes, methods of extending a printing dialog, and the advantages of using a printing dialog extension.

- "Interface Guidelines for Custom Panes" (page 21) presents guidelines for designing an interface for a custom pane in a printing dialog.
- "Printing Dialog Extension Concepts" (page 27) discusses the printing dialog extension—its functional components, runtime behavior, and capabilities.

The second part is a tutorial that shows how to construct and use a printing dialog extension:

- "Creating a Plug-in Project" (page 35) shows how Project Builder can help you create a plug-in project to build a printing dialog extension.
- "Core Tasks" (page 43) shows how to implement much of the basic functionality in a printing dialog extension.
- "Custom Tasks" (page 61) shows how to implement the additional functionality that's specific to a custom pane.
- "Integration Tasks" (page 77) shows you how applications and printer modules install and communicate with their printing dialog extensions.

The appendixes cover a few remaining topics:

- "Utility Functions" (page 85) shows how to implement several utility functions that all printing dialog extensions can use.
- "Printing Plug-in Header Functions" (page 93) shows how to implement the three callback functions in the `PMPlugInHeader` interface.
- "Document Revision History" (page 97) contains a chronological list of the revisions to this book.

Other Options for Application Developers

Carbon applications can also extend the printing dialogs using the `AppendDITL` function, an older approach. Cocoa applications can extend the printing dialogs by adding an accessory view to an `NSPageLayout` or `NSPrintPanel` object.

In both cases, the printing system displays your custom controls inside a new dialog pane that's named for your application. Your application can host only a single custom pane in each dialog.

The section "Extending a Printing Dialog" (page 18) discusses the various options in more detail, and makes some recommendations.

Before You Start Reading

To get the most out of reading this book, you should first read *Mac OS X Printing System Overview*.

Printing Plug-in Interfaces Reference—the companion volume to this book—includes the reference documentation for the printing dialog extension API.

Both of these publications are available online in the ADC Reference Library.

You should also be familiar with Core Foundation plug-in concepts.

Printing Features and Printing Dialog Panes

This chapter describes standard printing features, their relation to printing dialog panes, and the various ways of adding new panes to printing dialogs.

Throughout this chapter, the term "printing dialog" always refers specifically to the Page Setup and Print dialogs, which are introduced and discussed in detail in *Mac OS X Printing System Overview*.

Note: The illustrations of printing dialogs in this chapter are taken from Mac OS X version 10.1.

Standard Sets of Printing Features

Apple has identified a number of printing features common to many printers and applications. For convenience, similiar features are organized into named sets, as listed in Table 1-1.

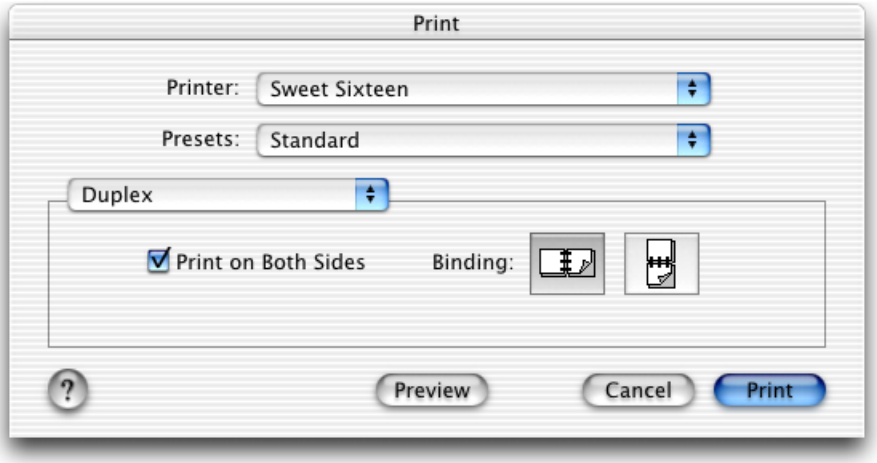
Table 1-1 Standard sets of printing features

Name	Features
Page Attributes	Formatting printer, paper size, orientation, scale
Copies & Pages	Copies per page, page range, collation
Layout	Page layout and border options
Duplex	Print on both sides, binding options
Output Options	Output to file in various formats
Paper Feed	Paper source tray selection
Error Handling	PostScript errors, tray switching
Printer Features	Features declared with PPD file keywords
Color Options	Color-matching system
Paper Type & Quality	Media types, paper quality levels

Each feature set has a corresponding set of parameters of various types. These parameters are used during the execution of a print job to control and refine the output.

The printing dialogs allow an application user to view and change control settings that represent the current values of these parameters. For example, Duplex is implemented in the Print dialog as shown in Figure 1-1.

Figure 1-1 Duplex pane in the Print dialog



Duplex has two parameters:

1. Print on both sides (Boolean)
2. Long-edge or short-edge binding (enumeration)

Note: The second parameter is defined only if two-sided printing is selected. Printing parameters often depend on other parameters in important ways.

A checkbox control represents the Boolean parameter, and two button controls represent the binding choices. Both buttons are disabled unless the checkbox is selected.

Printing Dialog Panes

A pane is a rectangular region inside a dialog window, in which a set of related controls are displayed together. Printing dialogs present a list of available panes using a pop-up menu, and allow the user to choose one for display.

A pane has the following characteristics:

- It has a **title**—a string that names the set of printing features it implements. The title appears in the pane pop-up menu, and also in the Summary pane.
- It has a set of controls, closely aligned to the parameters of a set of features provided by the destination printer or the application.
- It allows the user to examine and change settings for these controls.

From a user's perspective, all of the panes in a printing dialog look integrated and seem to have equal weight. The area allocated to each pane depends on its contents, and is limited only by the maximum width and height permitted by the printing system.

Standard and Custom Panes

The Apple-supplied printing dialog panes in Mac OS X are called **standard panes**. These panes implement standard sets of printing features, and give users a consistent interface as they switch between different applications and printers.

Table 1-2 lists the standard panes in Mac OS X. These panes implement most of the feature sets listed in Table 1-1 (page 15).

Table 1-2 The standard panes in Mac OS X

Pane	Dialog	PostScript
Page Attributes	Page Setup	
Copies & Pages	Print	
Layout	Print	
Duplex	Print	
Output Options	Print	
Paper Feed	Print	Y
Error Handling	Print	Y
Printer Features	Print	Y

Note: Three standard panes—Paper Feed, Error Handling, and Printer Features —implement features associated with PostScript printers. When the destination printer is a PostScript printer, these panes are available in the Print dialog.

Panes implemented by third-party application developers and printer vendors are called **custom panes**. Custom panes can implement standard feature sets found in Table 1-1 (page 15), or custom printing features for a specific application or printer.

When a custom pane implements a standard feature set, the printing system uses the standard name to identify the pane. However, the printing system never modifies the appearance of the user interface inside a custom pane.

Examples

A pane's dimensions are constrained by the maximum size of the dialog. The Page Setup and Print dialogs both display their panes in the central part of the window, but the two dialogs have different widths and layouts. To get a sense of where a pane appears in each dialog, it's useful to view some examples.

Figure 1-2 shows two examples of panes in the Page Setup dialog. The shaded area indicates the actual pane rectangle.

Figure 1-2 Panes in the Page Setup dialog

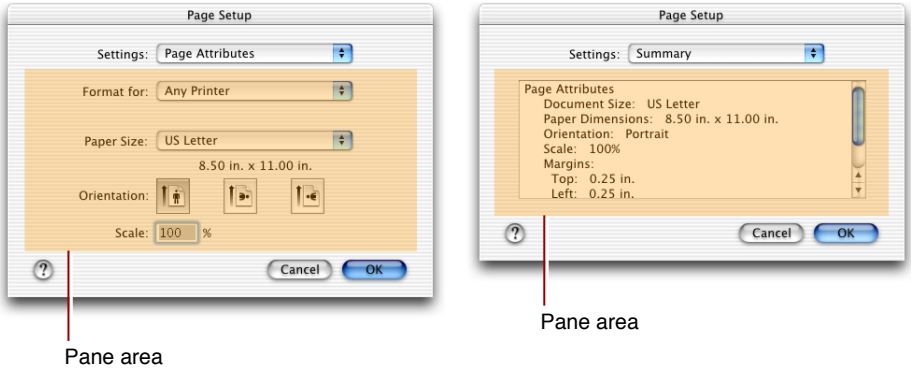
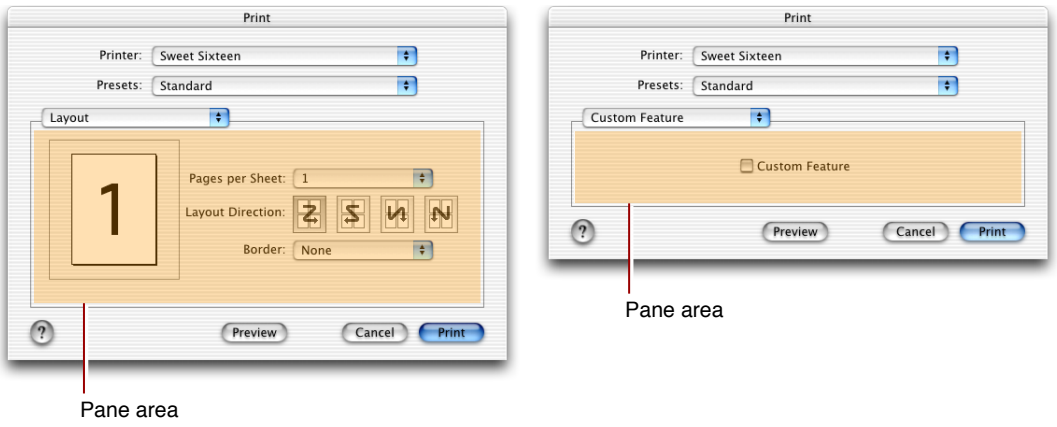


Figure 1-3 shows two examples of panes in the Print dialog. The dialog surrounds the pane with a group box that is slightly larger than the pane—the shaded area indicates the actual pane rectangle.

Figure 1-3 Panes in the Print dialog



The pane on the right probably looks unfamiliar—clearly it's not a standard pane. This is a custom pane implemented with a printing dialog extension.

Extending a Printing Dialog

Adding a custom pane to a printing dialog is said to extend the dialog, because the pane—together with the software that implements it—adds functionality that goes beyond what Mac OS X provides.

What's involved in extending a printing dialog?

At minimum, you need to

- define new parameters for any custom printing features

- design a human interface
- develop software that manages the interface and its parameters

Appending Dialog Items in Carbon

While the practice is discouraged, Carbon applications that run in Mac OS X can still use the `AppendDITL` function to extend a printing dialog. This approach is fully documented in Tech Note 1080, "Adding Items to the Printing Manager's Dialogs".

The application provides callback functions to initialize the dialog, append new items to it, draw the items, handle user actions on the appended items, and so on. The printing system displays the added controls inside a new dialog pane that's named for the application. The application can host only a single custom pane in each dialog.

Using the `AppendDITL` function is the only solution for application developers who want to maintain a common code base for Mac OS 9 and Mac OS X. In Mac OS X it has some drawbacks—for example, an application using this mechanism cannot display the dialog as a sheet.

Note: Printer module developers cannot use the `AppendDITL` function to extend a printing dialog.

Accessory Views in Cocoa

A Cocoa application can extend the Print dialog by adding a custom NSView to an NSPrintPanel object, using the `setAccessoryView` method. The custom view is displayed when the user selects the application's pane in the Print dialog. The printing system resizes the pane to accommodate the NSView you add.

In similiar fashion, a Cocoa application can extend the Page Setup dialog by adding a custom NSView to an NSPageLayout object.

For more information on how to add accessory views to printing dialogs in a Cocoa application, see *Printing Programming Topics for Cocoa*.

Printing Dialog Extensions

The **printing dialog extension** is a Mac OS X printing plug-in API that allows applications and printer modules to take advantage of advanced features in the printing system as they extend the printing dialogs. For a complete specification of this API, see *Printing Plug-in Interfaces Reference*.

Applications and printer modules can use printing dialog extensions to

- provide one or more custom dialog panes, each with a custom title
- support document-modal (or sheet) printing dialogs
- supply descriptions of custom pane settings in the Summary pane
- override a standard pane with a custom pane

Printer modules can also use printing dialog extensions to

- implement standard feature sets that have no standard pane, such as Color Options
- provide custom panes for PostScript features defined in PPD files

It's worth noting that Apple uses printing dialog extensions to implement almost all the standard panes. (The one exception is Summary, a special pane that contains information about other panes.)

Choosing the Best Solution

If you want to extend a printing dialog and you're not sure how to proceed, here are some recommendations.

Cocoa applications should use the approach described in "Accessory Views in Cocoa" (page 19).

If your Carbon application runs in Mac OS X exclusively, the choice is easy—you should always use a printing dialog extension.

For Carbon applications that run in both Mac OS 9 and Mac OS X:

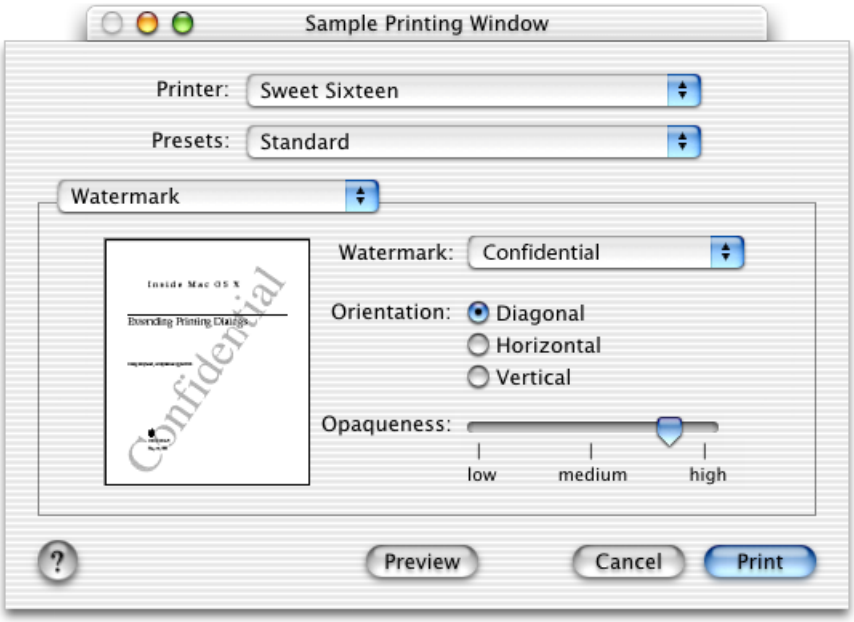
- You can use the approach described in "Appending Dialog Items in Carbon" (page 19) , although you won't be able to take advantage of some advanced printing features in Mac OS X.
- You can write a printing dialog extension, maintain two chunks of code, and determine at runtime which to use.

Interface Guidelines for Custom Panes

Each pane in a printing dialog is a self-contained human interface that enables a user to control the settings for a named set of printing features. This chapter presents guidelines for the design of this interface. The guidelines presented here will help you design an interface that's effective, visually pleasing, and consistent with other dialog components.

Figure 2-1 illustrates a possible design for a custom pane that presents settings for a watermark printing feature. As you read this chapter, consider how this interface design might be improved.

Figure 2-1 An example of a custom pane



Getting Started

Here are some things to do first, before you begin designing a custom pane.

- Get acquainted with Aqua.

  If this is your first attempt at designing a Mac OS X human interface, you should get acquainted with Aqua, the set of visual features and design standards that creates the distinctive look and feel of Mac OS X. The principal source of information about Aqua is the publication *Apple Human Interface Guidelines*.

- Learn about dialogs in Mac OS X.

You should understand the basic features of application-modal and document-modal (or sheet) dialogs in Mac OS X. A good place to start is the "Dialogs" chapter in *Apple Human Interface Guidelines*. The section on positioning controls in dialogs and windows is especially useful.

- Choose the appropriate printing dialog.

  The Page Setup and Print dialogs cover two distinct areas of printing functionality. Study a few of the standard panes, and make sure you understand why a pane appears in one dialog or the other.

- Choose appropriate printing features to implement.

  A printing feature that depends on a specific printer capability, such as color options, should always be handled by a printer module—not an application.

- Take a fresh look at your existing interface.

  A custom interface designed for a different operating system or windowing environment won't necessarily work well in Mac OS X.

- Get professional help.

  Consider using a specialist to design your interface. A professional visual designer can help you present your printing technology in the best possible light.

Clarity

It's important to present your interface in a simple, direct manner.

- Name your pane carefully.

  Make sure the title of your custom pane clearly and succinctly describes the set of printing features it supports. Don't use jargon or technical language. For example, call it "Paper Options" instead of "Media Options."

- Know your audience.

  If the target audience for your interface includes novice users, keep your interface as simple as possible. Minimize the number of controls. Avoid technical terms such as halftone, saturation, and gamma.

  If the target audience includes professional users, give them the advanced interface they require, but separate the advanced features from the frequently used features. For example, most people won't need access to controls for specifying precise percentages of cyan, yellow, and magenta.

- Center the interface.

  The interface should be center-biased. That is, controls and other interface elements should tend to be horizontally centered, with the exception of controls that must be visually aligned to indicate functional grouping. Examples and more details about interface layout are available in *Apple Human Interface Guidelines*.

- Provide feedback and communication.

  Any change in the value, position, or state of a control should be accompanied by some sort of feedback—visual or otherwise. Feedback helps people determine desired settings, and it is especially important for controls that affect print quality and color management.

Consistency

The interface should look and act in a consistent manner with respect to the dialog in which it appears.

- Use familiar terms.

  If elements of your human interface have the same meaning as those used in standard panes, try to use the same names, phrases, and terminology that Apple uses to describe the element.

- Preserve existing functionality.

  When overriding a standard pane, you should preserve the appearance, placement, and behavior of the existing controls. People prefer to see the same controls in the same location.

- Avoid duplication.

  Avoid multiple paths to the same feature. In other words, don't refer to or control the same printing feature in two places.

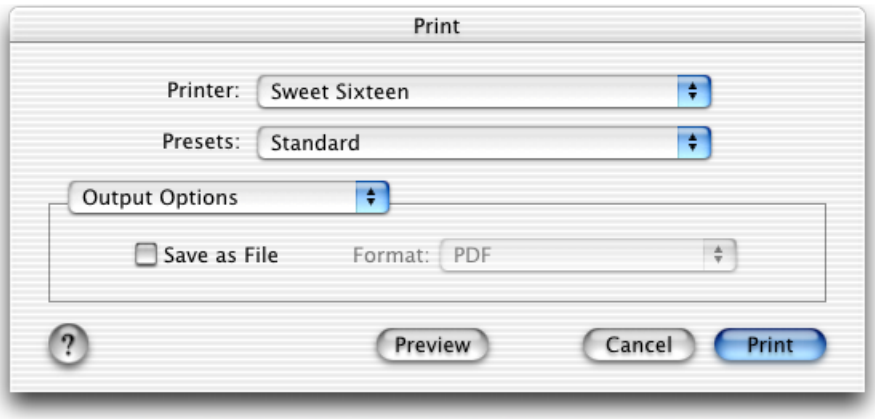
Dependencies

Dependency relationships sometimes exist between two or more printing features. A change in the setting of one feature might cause another feature to become invalid, or it might change the possible choices. Dependencies can exist in an interface whenever it contains two or more controls.

An interface should always provide feedback that indicates where dependencies exist, and prevents the user from making invalid choices.

For example, the Output Options pane in the Print dialog (shown in Figure 2-2) allows you to choose a file format only when Save as File is selected.

Figure 2-2 Output Options pane in the Print dialog



Localization

To **localize** means to adapt a human interface to meet the language, cultural, and other requirements of a specific geopolitical place or region. When localizing text, you localize for a standard language (such as English) or for one or more regional dialects (such as US English and British English).

To localize your human interface, you need to do the following:

1. Decide which languages and dialects you want to support (this is both a marketing and a technical decision).
2. Translate the text strings used in your interface into the various languages and dialects.
3. Create a visual layout of your human interface, showing the length and location of each string with respect to the associated control or other graphic element.
4. Choose the summary text you will provide when a user displays the Summary pane. Your summary text communicates the titles and current values of the settings in your human interface.
5. Translate the short strings used in your summary text into the various languages and dialects you support.
6. If you provide user assistance, you should localize your help text as well.
7. Configure your software so that the correct layout, summary text, and help text appear in the dialog at runtime.

For more information about localizing human interfaces, see *Getting Started with Internationalization*.

User Assistance

In Mac OS X, the printing dialogs have built-in user assistance implemented with a round help button in the lower-left corner. When a user clicks this button, the printing system uses Help Viewer to provide general information about the dialog.

You should consider providing additional user assistance for your custom pane. This is particularly important if you are introducing a new printing feature or a new type of control.

Here are a few general suggestions:

- Add graphics to the interface to illustrate what the current settings do.

  For example, the Watermark pane in Figure 2-1 (page 21) displays a graphic that previews the effect of the current settings.

- Use help tags to provide immediate assistance.

  If the cursor remains positioned over a control for a few seconds, you can display a help tag. Help tags are short messages displayed *in situ* to explain the meaning of your control in a few words.

  If you use Interface Builder to construct your human interface, you can use the Info window to associate a string of help text with each control. Interface Builder also allows you to choose on which side of the control the help tag appears.

- Provide in-depth assistance with Help Viewer.

  You can use Help Viewer to provide additional user assistance when your pane is visible. To provide this service, you need to do the following:

  1. Design HTML content for your custom pane.
  2. Write a Carbon event handler that detects a help-button click, launches Help Viewer, and displays your help content.
  3. Install the event handler when your Open function is called, and remove the handler in your Close function.

  To learn how to package HTML content for Apple Help, see *Apple Help Programming Guide*.

  "Handling the Help Event in a Printing Dialog" (page 88) explains how to implement a help event handler in your printing dialog extension.

Implementation

You can design the interface for your custom pane using Interface Builder, Apple's powerful interface development tool. Interface Builder is a straightforward and intuitive application that lets you

- drag controls from a palette into a layout window
- position and group the controls in compliance with the Aqua layout guidelines
- adjust the attributes of the controls (signature, size, visibility, etc.)
- save a static description of the interface in a `.nib` file that you incorporate into your project

To learn more about using Interface Builder to design an interface, see "Designing the Interface for a Custom Pane" (page 61).

Note: Carbon also supports the various resource formats defined in the Resource Manager.

Further Reading

If you're new to developing software for Mac OS X, or you want to learn more about the Mac OS X user experience, see http://developer.apple.com/ue/.

Printing Dialog Extension Concepts

This chapter describes printing dialog extensions in greater detail and explains how they are used in the printing system.

You should read this chapter if you are not familiar with printing dialog extensions or if you want to reinforce your understanding of the basic concepts.

Functional Components

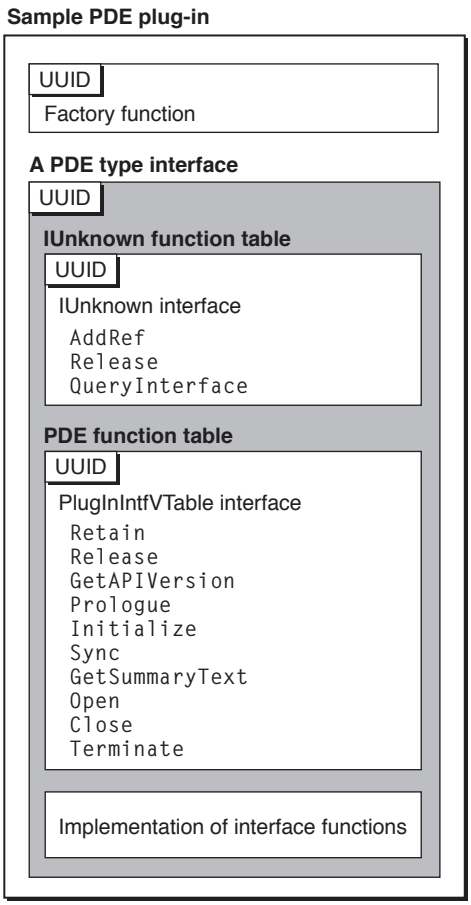
A printing dialog extension is a code module implemented as a plug-in. The printing system loads, manages, and unloads these plug-ins as needed, whenever an application user displays a printing dialog. This section provides a brief overview of the functional components that enable a printing dialog extension to operate successfully in the Mac OS X environment.

Carbon APIs usually describe a set of functions that you use to request system services. Your code calls the functions as needed, but you aren't aware of—or concerned with—their implementation.

Plug-in APIs are different. A plug-in does not call the functions described in the APIs, it implements them. The functions that a printing dialog extension implements are made available to its client—the printing system—using function tables that contain pointers to the executable code.

Figure 3-1 illustrates the functional components in a printing dialog extension.

Figure 3-1 Functional components of a printing dialog extension



The functional components in Figure 3-1 include the following:

- a factory function

  Core Foundation Plug-in Services uses the factory function to construct an instance of a base interface called `IUnknownVTbl`, which is used in turn to create instances of all other interfaces in a plug-in.

- two function tables

  The printing system uses these tables to call the functions you provide to implement the two required interfaces.

- the three functions in the `IUnknownVTbl` interface

  All Core Foundation plug-ins must implement these three functions. Plug-in Services uses them for reference-counting and to construct the other interfaces the plug-in implements.

- the first three functions in the `PlugInIntfVTable` interface

  All printing plug-ins must implement these functions. The printing system uses them for reference-counting and version control.

- the last seven functions in the `PlugInIntfVTable` interface

  All printing dialog extensions must implement these functions. The printing system calls them to do the work of initializing the pane and the parameter settings, managing the controls, and saving the settings.

Table 3-1 summarizes the programming interfaces that all printing dialog extensions implement.

Table 3-1 The required programming interfaces

Type	Description
`CFPlugInFactoryFunction`	Entry point for a CF plug-in
`IUnknownVTbl`	Base interface for a CF plug-in
`PlugInIntfVTable`	Base interface for a printing dialog extension

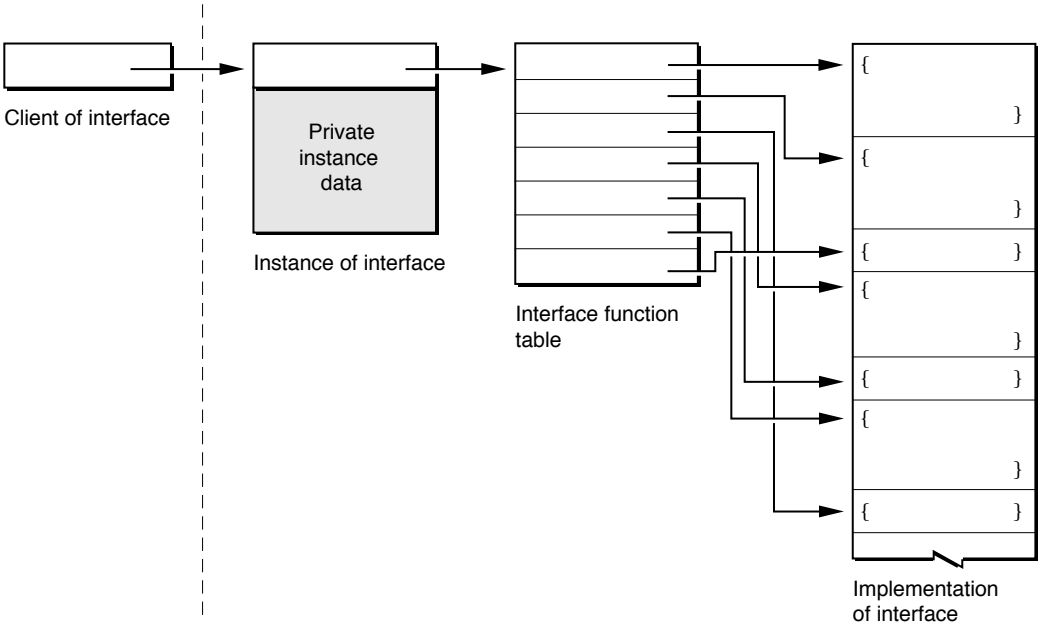
Instantiation of a Programming Interface

The specification for a programming interface describes the layout, calling syntax, and semantics of a set of functions. How does a plug-in make a programming interface available to clients at runtime?

Plug-ins provide access to an interface using a data structure called an **instance**. An instance contains a pointer to the interface function table (sometimes called a vtable), and possibly some data that's private to the instance.

The diagram in Figure 3-2 shows how the caller—in this case, the printing system—gains access to your executable code.

Figure 3-2 Using an instance to provide an interface to a plug-in client



A printing dialog extension implements two distinct interfaces, as illustrated in Figure 3-1 (page 28).Therefore it must supply at least two instances to the printing system, one for each interface.

When the printing system asks for an interface at runtime, you construct an instance and pass its address back to the caller. Note that the caller wants access only to your executable code—the caller knows nothing about your private instance data.

In "Plug-in Tasks" (page 43) you will learn when and how to construct a new instance for each of the two required interfaces.

Information Property Lists

As a plug-in, a printing dialog extension must have an information property list containing a prescribed set of key-value pairs, or associations. These entries contain configuration data used by various system services at runtime.

Several entries in the property list are used to discover and make use of the capabilities of a printing dialog extension:

- `CFBundleIdentifier` is a string that uniquely identifies the bundle. A printing dialog extension uses this identifier to locate resources within its bundle at runtime.
- `CFPlugInTypes` is a dictionary that identifies the types of custom panes being implemented, and identifies the factory function for each type.

  The printing system uses this information to discover three types of panes—Page Setup and Print dialog panes hosted by applications, and Print dialog panes hosted by printer modules.

  Note: Most printing dialog extensions implement only one type of pane.

- `CFPlugInFactories` associates factory UUIDs with factory function names.
- `CFPlugInDynamicRegistration` is configured to direct Plug-in Services to use `CFPlugInTypes` and `CFPlugInFactories` for static registration.

In "Editing the Bundle Properties" (page 36) you will learn how to define these property list entries.

Runtime Behavior

This section discusses some key aspects of the collaboration between Core Foundation Plug-in Services, the printing system, and the printing dialog extension at runtime.

Registration

When an application asks the printing system to display a new printing dialog, the printing system locates, loads, and activates the appropriate set of printing dialog extensions for the destination printer. To perform these tasks, the printing system collaborates with Core Foundation Plug-in Services, which maintains a registry of of all available printing dialog extensions.

The availability requirements for printing dialog extensions differ. An application-hosted printing dialog extension needs to be available only when the application is running. A printer module–based printing dialog extension needs to be available as long any associated printer queues exist.

Consequently, applications and printer modules register their printing dialog extensions with Core Foundation Plug-in Services in different ways:

- An application calls the function `CFPlugInCreate`, passing it a URL that specifies the location of the plug-in bundle. When the application terminates, its printing dialog extension is no longer available.
- A printer module passes the location of the plug-in bundle to the printing system, which takes care of registration. The printing system also maintains the association between the printing dialog extension and one or more printer queues.

To learn more about how applications and printer modules register printing dialog extensions, see "Integration Tasks" (page 77).

Activation

When an application asks the printing system to display a dialog, a set of printing dialog extensions are selected and loaded. The selection process must take into account a number of variables—the destination printer, the printer module, the dialog, rules of precedence, and so on.

Once a printing dialog extension is activated, the printing system calls its functions to initialize, (possibly) display, and release its custom pane.

- The `Prologue` function provides basic information about the pane—such as its size—and allocates memory for private data a printing dialog extension needs to maintain.
- The `Initialize` function initializes settings and provides additional information about the capabilities of the printing dialog extension.
- The `Sync` function maintains the correspondence between the pane settings and the associated parameter values recorded in a job ticket.
- The `GetSummaryText` function provides localized descriptions of each printing feature, along with its current value, for display in a Summary pane.
- The `Open` function performs any tasks deemed necessary just before the pane becomes visible, such as constructing the interface and installing Carbon event handlers.
- The `Close` function performs any tasks deemed necessary just before the pane is hidden, such as removing Carbon event handlers.
- The `Terminate` function performs tasks related to tearing down the human interface (such as releasing resources and memory) and preparing the plug-in to be unloaded.

Embedded Controls

Before its pane is made visible, a printing dialog extension creates a set of Carbon controls and embeds them inside a specialized container control called a user pane. The printing system can manipulate the user pane and its contents as a unit, without knowing or caring what's inside. The Control Manager draws the embedded Carbon controls and performs standard event-handling.

A printing dialog extension can use a Carbon event handler to register an event-handling function for any of its controls. The event-handling function is called whenever a user interacts with the corresponding control. The embedding hierarchy ensures that the proper event-handling function is called.

Contexts

In Mac OS X, applications can choose to support document-modal (or sheet) printing dialogs. Sheet dialogs make it possible to display the same printing dialog in several document windows at the same time. Printing dialog extensions must support this possibility, which means they must be reentrant.

When the printing system creates a new dialog, each printing dialog extension allocates a block of memory for private state information specific to the dialog. A **context** is a pointer to this memory. The printing system maintains the correspondence between a context and the dialog pane to which it applies, passing the correct context to the printing dialog extension as needed.

To learn more about contexts, see "Defining a Custom Context" (page 65).

Tickets

The Mac OS X printing system uses an opaque object called a **ticket** to maintain information associated with a print job. Printing dialog extensions use tickets to share their settings data with the host application or printer module, with the printing system, and potentially with other printing dialog extensions.

In general, printing plug-ins—such as printing dialog extensions and printer modules—have direct access to tickets via Ticket Services. Applications cannot directly access tickets. Instead, applications must query a `PMPageFormat` or `PMPrintSettings` object to retrieve ticket data.

Almost all printing dialog extensions introduce new printing features. For each new feature, you need to define

- a parameter or setting
- a data format to represent the parameter in a ticket
- a ticket key to use when accessing the parameter in a ticket

A printing dialog extension must use Core Foundation base types to represent its settings in a ticket. The following base types are permitted—CFString, CFNumber, CFDate, CFBoolean, CFArray, CFDictionary, and CFData. CFData can be used to represent data that does not correspond to one of the other Core Foundation base types.

Note: In Mac OS X version 10.2 and later, printing dialog extensions that use tickets to pass data to a printer module are limited to the following base types to represent the data—CFString, CFBoolean, CFDictionary, CFNumber (integer only), and CFArray.

To learn more about how printing dialog extensions use tickets, see "Accessing Ticket Data" (page 82).

Implementation of Standard Printing Features

Both printer modules and applications can host a printing dialog extension that implements—and possibly overrides another implementation of—one of the standard sets of printing features listed in Table 1-1 (page 15).

When implementing a standard feature set, the printing system ignores your custom title and uses the localized name of the feature set to identify your pane in the dialog.

The parameters for a standard feature set are considered to be public data if the printing system defines the keys and data types used to store the parameters in a ticket. If you implement a feature with a public parameter, you must add an entry to the appropriate ticket using the system-defined key and data type.

In addition to the supplying the same public data, your printing dialog extension can store its private data in the same ticket, using its own keys and data types.

If your printing dialog extension overrides a standard pane, you should extend—but not replace—the interface in the standard pane. Your customers benefit when you retain the appearance, placement, and behavior of the existing controls.

Important: If you want to override a standard pane, you should contact Apple Developer Technical Support for advice and assistance before you proceed.

Order of Precedence

If several registered printing dialog extensions implement the same set of standard printing features, the printing system uses precedence rules to determine which one is actually used in the dialog.

There are three levels of precedence:

1. Application
2. Printer module
3. Printing system

For example, an application-hosted printing dialog extension always takes precedence over printer modules or the printing system, regardless of the registration order.

The printing dialog extension that gets precedence must provide an implementation—in other words, it must display its interface within the assigned pane and provide the public data for all standard printing features.

If the user switches printers from inside the dialog, the printing system unloads all printing dialog extensions and rebuilds the dialog, re-applying the precedence rules.

PostScript Printer Features

PostScript printers have a number of features typically not available in raster printers. These features are defined in PostScript printer description (PPD) files supplied by printer vendors. PPD files contain keywords and other information to specify features and default settings for a PostScript printer. PPD files are created for specific printers or for printer families.

The purpose of the Printer Features pane in the Print dialog is to provide a default interface for the PostScript features defined in the PPD file associated with the destination printer.

Printer vendors are strongly encouraged to write printing dialog extensions that selectively override the Printer Features pane. For example, a custom pane could handle features that users often overlook, or features that would benefit from a well-designed human interface.

The setting for each PostScript feature you handle must be added to the print settings ticket, using the mechanism described in "Handling PostScript Features" (page 74).

Further Reading

To learn how printing dialog extensions declare the PPD features they implement, see "Registering PPD Main Keywords" (page 40).

For general information about PPD files, see *Using PostScript Printer Description Files*.

Creating a Plug-in Project

This chapter shows you how to configure a Project Builder project to build a printing dialog extension. Project Builder is Apple's integrated development environment (IDE) for Mac OS X.

To use Project Builder, you should install the latest release of Mac OS X developer tools. All of the software development applications and utilities discussed in this book are available on the Mac OS X Developer Tools CD. You can also download them from the Apple Developer Connection website, http://developer.apple.com/tools/.

If you prefer to use another IDE, you need to make sure that your printing dialog extension has the following characteristics:

- Core Foundation plug-in architecture

  The executable and its associated resources must be bundled as a Core Foundation plug-in, which means it needs to have the correct bundle structure and property list entries.

- C calling conventions

  The compiler must generate executable code that uses the standard C calling conventions. This applies to any function that might be called—by name or through a pointer—from outside the executable.

- Mach-O binary format

  The executable code must be packaged using the Mach object file (Mach-O) format.

Creating a Project With Project Builder

You can create a Project Builder plug-in project in one of two ways—using the sample project that accompanies this book or using a project template.

Using the Sample Project

Apple Technical Publications has created a sample project for developers who want to write a printing dialog extension. All of the source code examples in this book are adapted from this project.

The project defines two targets:

1. **PDEPrint** builds `PDEPrint.plugin`, a printing dialog extension that adds a custom pane to the Print dialog.

2. **AppUsingSheets** builds an application that you can use to test PDEPrint.

The core data types and functions that can be used in all printing dialog extensions are factored into a set of source files named `PDECore.*` and `PDEUtilities.*`. You should be able to use this source code in a real-world project with little or no modification.

The remaining data types, functions, and resources are in a set of files named `PDECustom.*` and `PDEPrint.*`. These files can serve as the basis for the custom components of a real-world project, but you will need to adapt and extend them to fit your specific requirements.

To view or download the current version of the sample project, see *PDEProject* in the ADC Reference Library.

Note: Even if you already have some experience developing printing dialog extensions, you may still benefit from new ideas and techniques introduced in this project.

Using a Project Builder Template

Project Builder ships with a number of project templates, including a plug-in template that's suitable for starting a new project from scratch.

To use this template:

1. Choose New Project from the Project Builder File menu.
2. Select the template CFPlugIn Bundle.
3. Specify the project name and desired location for the new project folder.

Editing the Bundle Properties

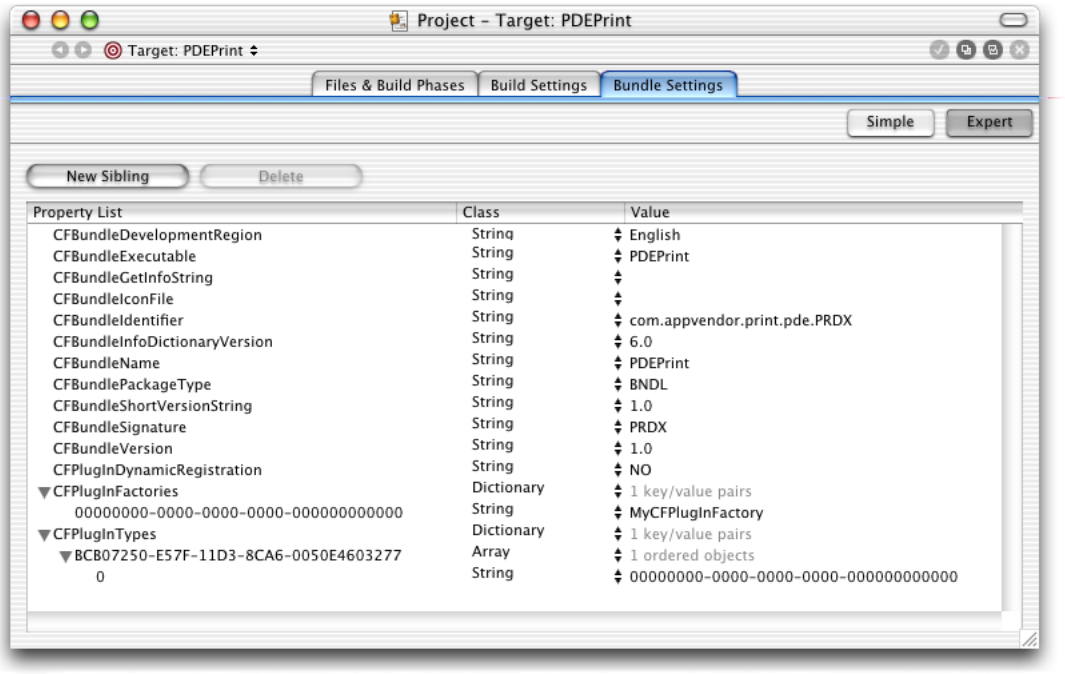
For your printing dialog extension to operate correctly, several important associations must be defined as key-value pairs in its information property list. In this section you will learn how to define these associations using Project Builder.

An information property list is a dictionary of key-value pairs. The keys must be strings, and the values must be valid property list types. For example, the required key-value pairs in the property list for a printing plug-in have values of type CFString, CFArray, and CFDictionary.

Project Builder generates the information property list for a target, using the information you supply in the target's bundle settings pane.

Figure 4-1 shows the property list entries for a printing dialog extension.

Figure 4-1 Bundle properties for a printing dialog extension



Listing 4-1 shows an XML representation of the same information, extracted from the `Info.plist` file generated by Project Builder.

Listing 4-1 XML representation of the bundle settings for a printing dialog extension

```
<plist version="0.9">
    <dict>
        <key>CFBundleDevelopmentRegion</key>
        <string>English</string>
        <key>CFBundleExecutable</key>
        <string>PDEPrint</string>
        <key>CFBundleGetInfoString</key>
        <string>1.0</string>
        <key>CFBundleIdentifier</key>
        <string>com.appvendor.print.pde.PRDX</string>
        <key>CFBundleInfoDictionaryVersion</key>
        <string>6.0</string>
        <key>CFBundleName</key>
        <string>PDEPrint</string>
        <key>CFBundlePackageType</key>
        <string>BNDL</string>
        <key>CFBundleShortVersionString</key>
        <string>1.0</string>
        <key>CFBundleSignature</key>
        <string>PRDX</string>
        <key>CFBundleVersion</key>
        <string>1.0</string>
        <key>CFPlugInDynamicRegistration</key>
        <string>NO</string>
        <key>CFPlugInFactories</key>
```

```
        <dict>
            <key>00000000-0000-0000-0000-000000000000</key>
            <string>MyCFPluginFactory</string>
        </dict>
        <key>CFPlugInTypes</key>
        <dict>
            <key>BCB07250-E57F-11D3-8CA6-0050E4603277</key>
            <array>
                <string>00000000-0000-0000-0000-000000000000</string>
            </array>
        </dict>
    </dict>
</plist>
```

To edit the bundle settings using Project Builder, open your project file and navigate to the bundle settings editor:

1. Click the Targets tab (or press Command-4).
2. Select the name of the desired target in the Targets list.
3. Click the Bundle Settings tab.
4. Click the Expert button.

Now you're ready to edit existing properties (or add new ones). To edit an existing property, double-click the key or value you want to change.

All of the properties you need to change are discussed in the following sections.

Warning: Project Builder maintains an internal representation of a target's bundle properties inside the project, and the `Info.plist` file is generated each time you build the target. That means you will lose any changes you make to the bundle properties outside of Project Builder.

Defining the Bundle Identifier

`CFBundleIdentifier` is the unique identifier string for the bundle. The bundle identifier can be used to locate the bundle at runtime using the function `CFBundleGetBundleWithIdentifier`.

Listing 4-2 shows an example of a bundle identifier.

Listing 4-2 A bundle identifier

```
        <key>CFBundleIdentifier</key>
        <string>com.appvendor.print.pde.PRDX</string>
```

Using the bundle settings editor, replace `com.appvendor.print.pde.PRDX` with a unique bundle identifier in the form of a Java-style package name—for example, `com.mycompany.pde.foo`.

Defining the Plug-in Factories

`CFPlugInFactories` is a list of one or more factory functions that can construct an instance of a specific interface type. The factory list is implemented as a dictionary. Each key-value pair consists of a unique factory ID and the name of a factory function.

Most printing dialog extensions have only one factory function, and therefore have only a single entry in their factory list. This case is illustrated in Listing 4-3.

Listing 4-3 A factory list with a single factory

```
<key>CFPlugInFactories</key>
<dict>
    <key>00000000-0000-0000-0000-000000000000</key>
    <string>MyCFPluginFactory</string>
</dict>
```

You need to supply both the identifier and the function name. The identifier must be a textual representation of a universally unique identifier (UUID) that you generate.

Using the bundle settings editor, replace `00000000-0000-0000-0000-000000000000` with your actual identifier, and replace `MyCFPluginFactory` with your actual factory function.

Tip: In Mac OS X 10.1 and later, with developer tools installed, you can use the command-line utility `/usr/bin/uuidgen` to generate a new UUID.

To learn more about UUIDs, see *Inside Carbon: Core Foundation Utility Services Concepts*.

Defining the Interface Types

`CFPlugInTypes` is a list of the interface types that the plug-in implements (not including `IUnknownVTbl`). The list is implemented as a dictionary. Each key-value pair consists of an interface ID paired with an array of one or more factory IDs.

Printing dialog extensions come in three flavors, as shown in Table 4-1.

Table 4-1 Printing dialog extension plug-in interface types

Interface type	UUID
Application pane for the Page Setup dialog	`B9A0DA98-E57F-11D3-9E83-0050E4603277`
Application pane for the Print dialog	`BCB07250-E57F-11D3-8CA6-0050E4603277`
Printer module pane for the Print dialog	`BDB091F4-E57F-11D3-B5CC-0050E4603277`

Most printing dialog extensions implement only a single type of interface with a single factory. This case is illustrated in Listing 4-4.

Listing 4-4 An interface types list with a single type and factory

```
<key>CFPlugInTypes</key>
<dict>
    <key>BCB07250-E57F-11D3-8CA6-0050E4603277</key>
    <array>
        <string>00000000-0000-0000-0000-000000000000</string>
    </array>
</dict>
```

You need to supply both identifiers. The first identifier should be the UUID in Table 4-1 that corresponds to the specific interface your printing dialog extension implements. The second identifier should be the UUID you created for the associated factory in "Defining the Plug-in Factories" (page 39).

Registering PPD Main Keywords

Note: This section applies to PostScript printer vendors only. If you are an application developer, you do not need to read this section.

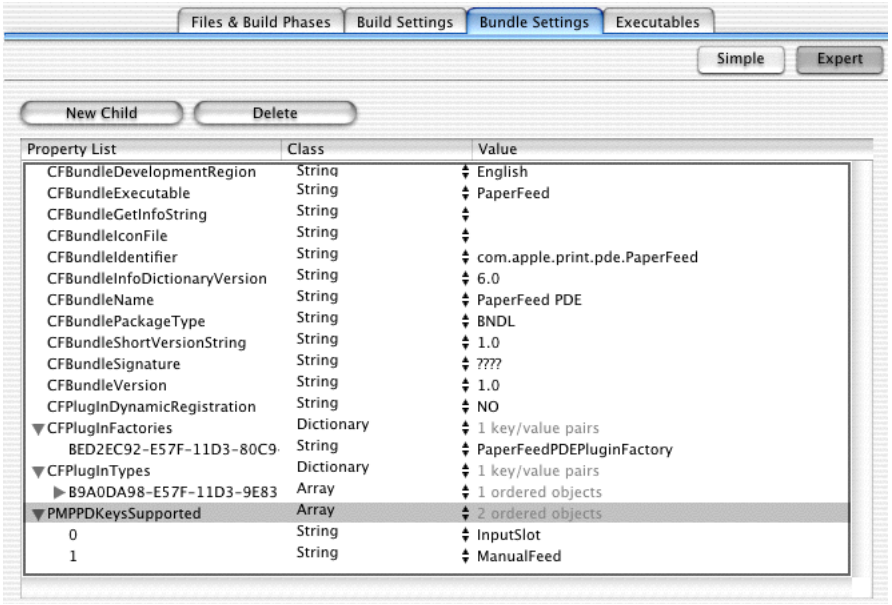
PostScript printer description (PPD) files are created by printer vendors to specify the set of features available in their PostScript printers. When a print queue is created for a PostScript printer, the printing system parses the contents of the appropriate PPD file looking for the data structures—identified by PPD main keywords—that specify the features and default settings the printer provides. The printing system uses this information to build a Printer Features pane for the printer.

A printer vendor can write a printing dialog extension that handles the settings for one or more PostScript printer features. You register PPD main keywords by adding them to the information property list of your printing dialog extension. If you register the PPD main keyword for a PostScript feature, the printing system omits the feature from the Printer Features pane.

You should make sure that keywords are unique with respect to the printer. Conflicts between printing dialog extensions that support the same keyword are not detected by the printing system. You should also make sure your extension loads only for the specific printer for which it is intended.

Figure 4-2 shows the bundle settings for a printing dialog extension that supports two paper feed options—input slot and manual feed.

Figure 4-2 Bundle settings for a printing dialog extension that supports paper-feed features



To register the PPD main keywords, you add a new key-value pair to the information property list. The key is `PMPPDKeysSupported`, and the value is an array that contains the keywords (in any order).

Here's how it's done:

1. Select the last entry (`CFPlugInTypes`), and click New Sibling.
2. Type `PMPPDKeysSupported`, making sure the case is correct.
3. Choose Array from the Class pop-up menu.
4. Click the disclosure triangle next to `PMPPDKeysSupported`. Then click the New Child button (when you click the disclosure triangle, the New Sibling button changes to New Child.)
5. Double-click the text box in the Value column, type the PPD main keyword, then make sure its class is set to String. For example, if the PPD main keyword is `*InputSlot`, type `InputSlot`.
6. For each additional PostScript feature you handle, add another child to the `PMPPDKeysSupported` property, then repeat the previous step.

At runtime your printing dialog extension code must add the setting for each PostScript feature you handle to the print settings ticket, using the mechanism described in "Handling PostScript Features" (page 74).

Further Reading

If you are not familiar with Project Builder, the O'Reilly publication *Learning Carbon* provides a good introduction to its basic features.

For detailed information about using PostScript printer description files in Mac OS X, see *Using PostScript Printer Description Files*.

Core Tasks

This chapter shows you how to implement most of the required callback functions in a printing dialog extension—that is, the functions in the `IUnknownVTbl` and `PlugInIntfVTable` interfaces. The source code presented here contains a core set of data types and functions that you can use in a real-world project with little or no modification.

All the code examples in this chapter are adapted from a complete sample project for writing a printing dialog extension. To view or download the current version of the sample project, see *PDEProject* in the ADC Reference Library.

Note: If you're using the sample project, this chapter is not required reading. You can proceed to "Custom Tasks" (page 61) and return to this chapter as needed.

Plug-in Tasks

A printing dialog extension is implemented as a loadable bundle that must perform a core set of tasks—including registration, interface discovery, and instance management—that allow it to operate as a printing plug-in in Mac OS X.

Defining Interface Structures

To provide a way for the printing system to use your callback functions at runtime, you need to define the data structures described in the following sections:

- "Defining Function Tables" (page 43) describes how to define two static tables with pointers to your callback functions.
- "Defining an Instance" (page 44) describes how to define a structure that represents an instance of one of your printing dialog extension interfaces.

Defining Function Tables

For each of the two required interfaces that your printing dialog extension must implement—`IUnknownVTbl` and `PlugInIntfVTable`—you need to define a table of function pointers. Taken together, these two tables give your client access to all of the required functions in your printing dialog extension.

Listing 5-1 shows how to define two static data structures for this purpose.

Listing 5-1 Function tables for the two required interfaces

```
static const IUnknownVTbl sMyIUnknownVTable =
```

```
{
    NULL, // required padding for COM
    MyQueryInterface,
    MyIUnknownRetain,
    MyIUnknownRelease
};

static const PlugInIntfVTable sMyPDEVTable =
{
    {
        MyPMRetain,
        MyPMRelease,
        MyPMGetAPIVersion
    },
    MyPrologue,
    MyInitialize,
    MySync,
    MyGetSummary,
    MyOpen,
    MyClose,
    MyTerminate
};
```

The `sMyIUnknownTable` structure consists of a table of pointers to the three callback functions that all Core Foundation plug-ins must implement.

Note: The `IUnknownVTbl` data type is based on the Component Object Model (COM) specification for IUnknown, the base interface that every COM plug-in implements. In this chapter, IUnknown and `IUnknownVTbl` refer to the same programming interface.

The `sMyPDEVTable` structure consists of a table of pointers to three functions that all printing plug-ins must implement and to seven functions that all printing dialog extensions must implement.

Your printing dialog extension supplies the base address of each table at runtime, as explained in "Defining an Instance" (page 44).

Defining an Instance

Listing 5-2 illustrates how you might define instance structures for these two interfaces. Because both interfaces support reference counting, each structure includes an integer counter.

Listing 5-2 Definition of instances in a printing dialog extension

```
typedef struct
{
    const IUnknownVTbl *vtable;
    SInt32 refCount;
    CFUUIDRef factoryID;

} MyIUnknownInstance;

typedef struct
{
    const PlugInIntfVTable *vtable;
    SInt32 refCount;
```

```
} MyPDEInstance;
```

Each `vtable` field points to a function table for one of the two required interfaces. You should assign a pointer to the appropriate function table described in "Defining Function Tables" (page 43).

Each `refCount` field is an integer used to keep track of the number of references to an instance. For more information on reference counting, see the discussion of memory management in *Core Foundation Overview*.

In the `IUnknownVTbl` instance, the `factoryID` field identifies the factory function that created the instance.

When a caller asks for an interface at runtime, you should construct an instance and pass its address back to the caller. The caller will pass this pointer back to your printing dialog extension later, giving various callback functions access to the same instance.

The instance pointer you supply must also point to the base address of the function table for the requested interface. In other words, the address of the function table must always be the first field in the instance structure. In Listing 5-2 this is the `vtable` field.

Later in this chapter, you will learn when and how to construct a new instance for each of the two required interfaces.

Implementing a Factory Function

Your factory function is the main entry point for your printing dialog extension. When the printing system calls `CFPlugInInstanceCreate` to request an instance of your `IUnknownVTbl` interface, Plug-in Services loads your executable and invokes its factory function.

You can give your factory function any name you wish. To support runtime name binding, you need to

- use this name in the `CFPlugInFactories` entry in your property list
- direct the compiler to export the name

Listing 5-3 implements a factory function that constructs an instance of your `IUnknownVTbl` interface and returns its address to the caller.

Listing 5-3 A factory function that supplies an instance of the `IUnknownVTbl` interface

```
extern void* MyCFPlugInFactory (
    CFAllocatorRef allocator,
    CFUUIDRef typeUUID
)

{
    CFBundleRef         myBundle    = NULL;
    CFDictionaryRef     myTypes     = NULL;
    CFStringRef         requestType = NULL;
    CFArrayRef          factories   = NULL;
    CFStringRef         factory     = NULL;
    CFUUIDRef           factoryID   = NULL;
    MyIUnknownInstance  *instance   = NULL;                                    // 1

    myBundle = MyGetBundle();                                                  // 2
```

```
    if (myBundle != NULL)
    {
        myTypes = CFBundleGetValueForInfoDictionaryKey (                        // 3
            myBundle, CFSTR("CFPlugInTypes"));

        if (myTypes != NULL)
        {
            requestType = CFUUIDCreateString (allocator, typeUUID);
            if (requestType != NULL)
            {
                factories = CFDictionaryGetValue (myTypes, requestType);        // 4
                CFRelease (requestType);
                if (factories != NULL)
                {
                    factory = CFArrayGetValueAtIndex (factories, 0);            // 5
                    if (factory != NULL)
                    {
                        factoryID = CFUUIDCreateFromString (
                            allocator, factory);
                        if (factoryID != NULL)
                        {
                            instance = malloc (sizeof(MyIUnknownInstance));     // 6
                            if (instance != NULL)
                            {
                                instance->vtable = &sMyIUnknownVTable;
                                instance->refCount = 1;                         // 7
                                instance->factoryID = factoryID;                // 8
                                CFPlugInAddInstanceForFactory (factoryID);      // 9
                            }
                            else {
                                CFRelease (factoryID);
                            }
                        }
                    }
                }
            }
        }
    }

    return instance;
}
```

Here's what the factory function in Listing 5-3 does:

1. Sets the default return value to `NULL`. The factory function returns `NULL` if there is an error, to indicate that no instance was created.
2. Calls a utility function that returns a reference to your plug-in bundle. An implementation of `MyGetBundle` is provided in the sample project.
3. Gets a reference to the `CFPlugInTypes` dictionary in the property list, which contains the type identifier for this plug-in.
4. Gets a reference to the `CFPlugInFactories` entry for the plug-in type requested by the caller.

5. Gets a reference to the factory identifier for this plug-in type. If your plug-in has more than one factory function, you may need to modify this code.

6. Allocates memory for a new instance of `IUnknownVTbl`.

7. Sets the initial reference count to 1, because the caller owns this instance.

8. Saves the identifier for the factory that created this instance. This identifier is needed later when you remove the registration for this instance.

9. Registers this instance with Plug-in Services. Plug-In Services will not unload your printing dialog extension while an instance is registered.

Implementing the IUnknown Interface

As a Core Foundation plug-in, your printing dialog extension must implement the three functions in `IUnknownVTbl`. These functions give the printing system access to the other interfaces your printing dialog extension implements. To learn more about how a plug-in provides access to an interface, see "Instantiation of a Programming Interface" (page 29).

Query Interface

Listing 5-4 implements a query interface function that creates an instance of the `PlugInIntfVTable` interface.

Listing 5-4 A query interface function in a printing dialog extension

```
static HRESULT MyQueryInterface (
    void *this,                                                          // 1
    REFIID iID,                                                          // 2
    LPVOID *ppv                                                          // 3
)

{
    CFUUIDRef requestID = NULL;
    CFUUIDRef actualID  = NULL;
    HRESULT   result = E_UNEXPECTED;

    requestID = CFUUIDCreateFromUUIDBytes (kCFAllocatorDefault, iID);    // 4
    if (requestID != NULL)
    {
        actualID = CFUUIDCreateFromString (                              // 5
            kCFAllocatorDefault,
            kDialogExtensionIntfIDStr
        );

        if (actualID != NULL)
        {
            if (CFEqual (requestID, actualID))                           // 6
            {
                MyPDEInstance *instance = malloc (sizeof(MyPDEInstance));
                if (instance != NULL)
                {
```

```
                instance->vtable = &sMyPDEVTable;
                instance->refCount = 1;
                *ppv = instance;                                        // 7
                result = S_OK;
            }
        }
        else
        {
            if (CFEqual (requestID, IUnknownUUID))                      // 8
            {
                MyIUnknownRetain (this);
                *ppv = this;
                result = S_OK;
            }
            else
            {
                *ppv = NULL;
                result = E_NOINTERFACE;
            }
        }

        CFRelease (actualID);
    }

    CFRelease (requestID);
}

return result;
}
```

Here's what the code in Listing 5-4 does:

1. Receives a pointer to the same `IUnknownVTbl` instance that was constructed by your factory function. (To the caller, this pointer is the address of a pointer to the function table for your `IUnknownVTbl` interface.)
2. Receives a string that contains the UUID of the interface the client wants.
3. Receives the address of storage in the calling function for a pointer to a pointer to the function table.
4. Gets a reference to the `CFUUID` for the requested interface. In general, you should always convert a UUID string into a `CFUUID` object before using it.
5. Gets a reference to the conventional `CFUUID` for a printing dialog extension interface. The printing system (the caller) always requests an instance of this interface using the string constant kDialogExtensionIntfIDStr.
6. Checks to see if the caller wants an instance of the `PlugInIntfVTable` interface. This is the usual case.
7. Passes back a pointer to the new instance of the `PlugInIntfVTable` interface.
8. Checks to see if the caller wants a copy of the `IUnknownVTbl` instance. If so, this code assigns the same instance that your factory function supplied. The constant `IUnknownUUID` is defined in the Plug-in Services API.

Add Reference

Listing 5-5 implements a function that retains (increments the reference count of) an instance of the IUnknown interface.

Listing 5-5 A function that retains an instance of the IUnknown interface

```
static ULONG MyIUnknownRetain (void* this)
{
    MyIUnknownInstance* instance = (MyIUnknownInstance*) this;              // 1
    ULONG refCount = 1;                                                    // 2

    if (instance != NULL) {
        refCount = ++instance->refCount;                                   // 3
    }

    return (refCount);
}
```

Here's what the code in Listing 5-5 does:

1. Defines a pointer to the `IUnknownVTbl` instance that was constructed by the factory function. This pointer provides access to the `refCount` field for this instance.
2. Defines a variable for the updated reference count, and sets its default value to 1.
3. Increments the reference count for this instance.

Release Reference

Listing 5-6 implements a function that releases (decrements the reference count of) an instance of the IUnknown interface.

If the reference count reaches zero, the function releases Core Foundation objects on which the instance depended, and then destroys the instance.

Listing 5-6 A function that releases an instance of the IUnknown interface

```
static ULONG MyIUnknownRelease (void* this)
{
    MyIUnknownInstance* instance = (MyIUnknownInstance*) this;              // 1
    ULONG refCount = 0;                                                    // 2

    if (instance != NULL)
    {
        refCount = --instance->refCount;                                   // 3
        if (refCount == 0)
        {
            CFPlugInRemoveInstanceForFactory (instance->factoryID);        // 4
            CFRelease (instance->factoryID);                               // 5
            free (instance);                                               // 6
            MyFreeBundle();                                                // 7
        }
    }
```

```
    return refCount;
}
```

Here's what the code in Listing 5-6 does:

1. Defines a pointer to an instance of the `IUnknownVTbl` interface. This is the same instance the factory function supplied.
2. Defines a variable for the updated reference count, and sets its default value to zero.
3. Decrements the reference count for this instance.
4. Removes the instance from the Plug-in Services registry.
5. Releases the factory ID, because it is no longer needed.
6. Deallocates storage for the instance.
7. Releases our bundle reference, in case the plug-in is being unloaded. For more information about `MyFreeBundle`, see the comments in the sample project that accompanies this book.

Printing Dialog Extension Tasks

The sample code presented in this section shows you how to implement the seven required callback functions discussed in "Activation" (page 31), in a manner that supports reentrancy.

Defining a Context

To support document-modal (or sheet) dialogs, a printing dialog extension must be capable of managing the state of its pane in several dialog windows concurrently.

For each dialog window, your printing dialog extension allocates memory for pane state information and supplies its address—called a **context**—to the printing system.

Listing 5-7 shows how the context is defined in the sample project.

Listing 5-7 A sample context structure

```
typedef struct
{
    ControlRef       userPane;
    EventHandlerRef  helpHandler;
    EventHandlerUPP  helpHandlerUPP;
    void*            customContext;
    Boolean          initialized;

} MyContextBlock;

typedef MyContextBlock* MyContext;
```

The `userPane` field is a reference to the container control into which you embed the controls in your custom interface.

The `helpHandler` field is a reference to a Carbon event handler that's active when your custom pane is visible. The handler detects when the help button is clicked, and displays your custom help content using Help Viewer.

The `helpHandlerUPP` field is a universal procedure pointer that's allocated and used whenever the help event handler is installed.

The `customContext` field is for a pointer to a block of additional memory allocated in your custom code, as described in "Defining a Custom Context" (page 65).

The `initialized` field indicates whether the interface in your custom pane has been constructed and initialized.

Note: The printing system always refers to your context using the generic type `PMPDEContext`. To gain access to your context data, you should cast the generic type to your actual context type.

Implementing the Required Callbacks

All printing dialog extensions must implement the ten functions in the `PlugInIntfVTable` interface. The first three functions—`PMRetain`, `PMRelease`, and `PMGetAPIVersion`—are described in the appendix "Printing Plug-in Header Functions" (page 93). The remaining seven functions are described here in this section.

Prologue

The prologue function allocates a block of memory for a context, and passes this context—along with some static information about your custom pane—back to the printing system.

You can give your prologue function any name you wish. You need to enter this name in the function table for the `PlugInIntfVTable` interface, as described in "Defining Function Tables" (page 43).

Listing 5-8 implements a prologue function.

Listing 5-8 A prologue function

```
static OSStatus MyPrologue                                          // 1
(
    PMPDEContext    *outContext,
    OSType          *creator,
    CFStringRef     *paneKind,
    CFStringRef     *title,
    UInt32          *maxH,
    UInt32          *maxV
)

{
    MyContext context = NULL;
    OSStatus result = kPMInvalidPDEContext;
```

```
    context = malloc (sizeof (MyContextBlock));                          // 2
    if (context != NULL)
    {
        context->customContext = MyCreateCustomContext();                // 3
        context->initialized = false;
        context->userPane = NULL;

        *outContext = (PMPDEContext) context;                            // 4
        *creator    = kMyBundleCreatorCode;                              // 5
        *paneKind   = kMyPaneKindID;                                     // 6
        *title      = MyGetTitle();                                      // 7
        *maxH       = kMyMaxH;                                           // 8
        *maxV       = kMyMaxV;                                           // 9

        result = noErr;
    }

    return result;                                                       // 10
}
```

Here's what the code in Listing 5-8 does:

1. The six calling parameters are all pointers to storage provided by the caller (the printing system) to receive information from this prologue function.
2. Allocates memory for a new context.
3. Calls a function that creates and returns a custom context. This context is described in "Defining a Custom Context" (page 65).
4. Passes back your context to the printing system.
5. Passes back the unique 4-byte creator code that identifies your printing dialog extension. This code is described in "Defining Bundle, Pane, and Nib Identifiers" (page 63).
6. Passes back the string that identifies your custom pane. This identifier is described in "Defining Bundle, Pane, and Nib Identifiers" (page 63).
7. Passes back the title of your custom pane, which is supplied by the custom function described in "Providing the Title of your Custom Pane" (page 67). Your printing dialog extension retains ownership of the title string, and should release it when it's no longer needed.

   If you are implementing one of the standard sets of printing features listed in Table 1-1 (page 15), the printing system may not use your custom title.
8. Passes back the desired width of the custom pane in pixel units. While this value is not used by the printing system in Mac OS X version 10.2 and earlier, it might be used in a future version of Mac OS X.
9. Passes back the desired height of the custom pane in pixel units.
10. Returns a result code to the caller. If your prologue function returns a non-zero result code, the printing system immediately unloads your printing dialog extension (without calling your terminate function.)

Initialize

If your prologue function returns `noErr`, the printing system proceeds to call your initialize function.

You can give the initialize function any name you wish. You need to enter this name in the function table for the `PlugInIntfVTable` interface, as described in "Defining Function Tables" (page 43).

As the name suggests, the purpose of the initialize function is to provide initial values for the settings associated with your user interface. If the settings are already defined in a ticket, the initialize function should retrieve their values from the ticket at this time.

In Mac OS X version 10.2 and later, you can use `SetControlBounds` to adjust the vertical size of the user pane at this time. If your pane is displayed later, the dialog will reflect the adjusted pane height. This feature is useful if the desired size of your pane is not known until your initialize function is called.

Listing 5-9 implements a "lazy" initialize function that defers the task of creating and embedding Carbon controls until later, when the open function is called (see "Open" (page 56)). As a result, the cost of creating the user interface is not incurred unless a user actually displays the pane.

Listing 5-9 An initialize function

```
static OSStatus MyInitialize
(
    PMPDEContext inContext,
    PMPDEFlags *flags,
    PMPDERef ref,                                                    // 1
    ControlRef userPane,                                             // 2
    PMPrintSession session                                           // 3
)

{
    MyContext context = (MyContext) inContext;                       // 4
    OSStatus result = noErr;

    *flags = kPMPDENoFlags;
    context->userPane = userPane;                                    // 5

    result = MySync (
        inContext, session, kSyncPaneFromTicket);                    // 6

    return result;                                                   // 7
}
```

Here's what the code in Listing 5-9 does:

1. The `ref` parameter is not used.
2. Receives a reference to a user pane created for you by the printing system. Later, you will embed the Carbon controls for your user interface into this user pane.
3. Receives a reference to a session object that contains information about the current printing session. Your printing dialog extension uses this parameter to gain access to a job ticket.
4. Casts the parameter to your context type. This is the same context you defined and passed back to the printing system in your prologue function.

5. Saves the pane reference for later use.

6. Calls the function described in "Sync" (page 54) to initialize your settings. If the job ticket does not contain your settings, then default values are used.

7. Returns a result code to the caller. If your initialize function returns a non-zero result code, the printing system immediately calls your terminate function.

Sync

The sync function in a printing dialog extension maintains the correspondence between the settings in a custom pane and their recorded values in either the `PMPageFormat` or the `PMPrintSettings` ticket.

You can give your sync function any name you wish. You need to enter this name in the function table for the `PlugInIntfVTable` interface, as illustrated in "Defining Function Tables" (page 43).

There are two possible ways to synchronize—update ticket from pane, or update pane from ticket. Because synchronization requires knowledge of your custom settings, the work is done in the custom functions `MySyncPaneFromTicket` and `MySyncTicketFromPane`. These two functions are described in "Synchronizing User Settings With a Ticket" (page 69).

Listing 5-10 implements a sync function that calls the appropriate custom sync function, based on the sync direction specified by the caller. Two parameters are passed to the custom function—the custom context described in "Defining a Custom Context" (page 65), and the printing session.

Listing 5-10 A sync function

```
static OSStatus MySync
(
    PMPDEContext inContext,
    PMPrintSession session,
    Boolean syncDirection
)

{
    MyContext context = (MyContext) inContext;
    OSStatus result = noErr;

    if (syncDirection == kSyncPaneFromTicket) {
        result = MySyncPaneFromTicket (context->customContext, session);
    }
    else {
        result = MySyncTicketFromPane (context->customContext, session);
    }

    return result;
}
```

Get Summary Text

The summary function in a printing dialog extension provides the title and current value of each setting in its custom pane. The printing system displays this information in the Summary pane.

You can give this function any name you wish. You need to enter this name in the function table for the `PlugInIntfVTable` interface, as illustrated in "Defining Function Tables" (page 43).

Listing 5-11 implements a summary function that creates two summary arrays, calls the custom function `MyGetSummaryText` to fill them in, and then passes the arrays back to the printing system.

Listing 5-11 A summary function

```
static OSStatus MyGetSummary
(
    PMPDEContext inContext,
    CFArrayRef *titles,
    CFArrayRef *values
)

{
    MyContext context = (MyContext) inContext;
    CFMutableArrayRef titleArray = NULL;
    CFMutableArrayRef valueArray = NULL;

    OSStatus result = kPMInvalidPDEContext;

    titleArray = CFArrayCreateMutable (                                    // 1
        kCFAllocatorDefault, 0, &kCFTypeArrayCallBacks);

    if (titleArray != NULL)
    {
        valueArray = CFArrayCreateMutable (                                // 2
            kCFAllocatorDefault, 0, &kCFTypeArrayCallBacks);

        if (valueArray != NULL)
        {
            result = MyGetSummaryText (                                    // 3
                context->customContext,
                titleArray,
                valueArray
            );
        }
    }

    if (result != noErr)
    {
        if (titleArray != NULL)
        {
            CFRelease (titleArray);
            titleArray = NULL;
        }
        if (valueArray != NULL)
        {
            CFRelease (valueArray);
            valueArray = NULL;
        }
    }

    *titles = titleArray;                                                  // 4
    *values = valueArray;
```

```
    return result;
}
```

Here's what the code in Listing 5-11 does:

1. Creates the mutable array `titleArray`. The second argument is zero to indicate that the array should not have a fixed size.

   Core Foundation arrays may contain references to mixed types, but in this case the printing system assumes this is an array of strings.

2. Creates the mutable array `valueArray` with the same characteristics.

3. Calls a custom function that fills in `titleArray` and `valueArray` with strings that describe the current values of the settings in your pane.

4. Passes the arrays back to the printing system, which is responsible for releasing the arrays and their contents.

Open

When the dialog user displays your custom pane, the printing system calls your open function immediately before the pane becomes visible.

You can give this function any name you wish. You need to enter this name in the function table for the `PlugInIntfVTable` interface, as illustrated in "Defining Function Tables" (page 43).

Listing 5-12 implements an open function that constructs a custom nib-based interface inside a dialog pane, sets the default values of the controls in the interface, prepares the interface for display, and installs an event handler for the help button.

Listing 5-12 An open function

```
static OSStatus MyOpen (PMPDEContext inContext)
{
    MyContext context = (MyContext) inContext;
    OSStatus result = noErr;

    if (!context->initialized)
    {
        IBNibRef nib = NULL;

        result = CreateNibReferenceWithCFBundle (                    // 1
            MyGetBundle(),                                           // 2
            kMyNibFile,
            &nib
        );

        if (result == noErr)
        {
            WindowRef nibWindow = NULL;

            result = CreateWindowFromNib (                           // 3
                nib,
                kMyNibWindow,
```

```
                &nibWindow
            );

            if (result == noErr)
            {
                result = MyEmbedCustomControls (                        // 4
                    context->customContext,
                    nibWindow,
                    context->userPane
                );

                if (result == noErr)
                {
                    context->initialized = TRUE;
                }

                DisposeWindow (nibWindow);
            }

            DisposeNibReference (nib);
        }
    }

    if (context->initialized)
    {
        result = MyInstallHelpEventHandler (                            // 5
            GetControlOwner (context->userPane),                        // 6
            &(context->helpHandler),                                    // 7
            &(context->helpHandlerUPP)                                  // 8
        );
    }

    return result;
}
```

Here's what the code in Listing 5-12 does:

1. Creates a nib object, using the nib file located inside the plug-in bundle for this printing dialog extension.
2. Calls a custom function that returns a reference to the plug-in bundle. For an implementation of this function, see the sample project.
3. Instantiates your nib-based interface in an offscreen Carbon window.
4. Calls a custom function that embeds your custom controls inside the dialog pane. For more information, see "Embedding Your Controls in a Dialog Pane" (page 68).
5. Calls a custom function that installs a Carbon event handler when your custom pane is visible. For more information, see "Installing a Help Event Handler" (page 89).
6. Gets a reference to the dialog window.
7. Saves a reference to the event handler. You should remove the handler when the pane closes.
8. Saves the event handler UPP. You should deallocate the UPP when the event handler is removed.

Close

The printing system pairs each call to the open function with a corresponding call to the close function—except when your pane is visible and the user cancels the dialog. The close function performs any necessary tasks when the printing system hides your pane.

You can give your close function any name you wish. You need to enter this name in the function table for the `PlugInIntfVTable` interface, as illustrated in "Defining Function Tables" (page 43).

Listing 5-13 implements a close function that removes the help event handler installed in "Open" (page 56).

Listing 5-13 A close function

```
static OSStatus MyClose (PMPDEContext inContext)
{
    MyContext context = (MyContext) inContext;
    OSStatus result = noErr;

    result = MyRemoveHelpEventHandler (
        &(context->helpHandler),
        &(context->helpHandlerUPP)
    );

    return result;
}
```

Terminate

The printing system calls the terminate function when the dialog is dismissed for any reason, or when the user changes the destination printer. The terminate function performs necessary clean-up tasks, such as releasing resources and deallocating memory.

You can give this function any name you wish. You need to enter this name in the function table for the `PlugInIntfVTable` interface, as described in "Defining Function Tables" (page 43).

Listing 5-14 implements a terminate function that frees the two contexts allocated in the prologue function, described in "Prologue" (page 51).

Listing 5-14 A terminate function

```
static OSStatus MyTerminate (
    PMPDEContext inContext,
    OSStatus inStatus
)

{
    MyContext context = (MyContext) inContext;
    OSStatus result = noErr;

    if (context != NULL)
    {
        result = MyRemoveHelpEventHandler (                    // 1
            &(context->helpHandler),
            &(context->helpHandlerUPP)
        );
```

```
        if (context->customContext != NULL) {
            MyReleaseCustomContext (context->customContext);                    // 2
        }

        free (context);                                                         // 3
    }

    return result;
}
```

Here's what the code in Listing 5-14 does:

1. Removes the help event handler if it's still installed. For more information, see "Installing a Help Event Handler" (page 89).
2. Releases your custom context. For more information, see "Defining a Custom Context" (page 65).
3. Frees memory allocated for the context. For more information about contexts, see"Defining a Context" (page 50) and "Prologue" (page 51).

Custom Tasks

This chapter shows you how to implement a set of custom data types and functions that add unique behavior to your printing dialog extension. You can use the source code in this chapter in a real-world project, but you will need to adapt and extend the code to fit your specific requirements.

All the code examples in this chapter are adapted from a complete sample project for writing a printing dialog extension. To view or download the current version of the sample project, see *PDEProject* in the ADC Reference Library.

Designing the Interface for a Custom Pane

A graphical editor like Apple's Interface Builder is an important tool for designing and constructing a user interface. As a visual designer, you can use an editor to create a prototype of the interface. As a developer, you can use the same editor to fill in the details and construct an archive that can be instantiated at runtime. Tools of this type are especially important when you need to manage many localized versions of a single interface.

This section shows how to use Interface Builder to design a custom pane for your printing dialog extension. Interface Builder saves your design as an XML archive using the `.nib` filename extension. The archive is a static representation of the interface that can be efficiently loaded into memory when needed.

Designing the Interface

Designing an interface with Interface Builder is straightforward and intuitive. You drag Carbon controls from a palette into a Carbon window, position and group the controls as needed, and use an Info window to adjust their attributes. You save the Carbon window as an archive, and install the archive in an appropriate location inside the plug-in bundle.

The following procedure shows how to design the interface for a printing dialog extension using Interface Builder.

1. Decide on a 4-byte signature (or creator code) that identifies your interface.
2. Open Interface Builder and create a new Carbon window.
3. Open the Info window (Tools > Show Info or Commmand-Shift-I).
4. Set the width and height of the Carbon window in pixels. These values might change as you refine your design.
5. For each control in your interface:
    a. Drag a control from the Controls palette into position inside your Carbon window.

b. Set the control signature to your 4-byte interface signature.

c. Set the control ID to a unique positive integer.

d. Enter the title string.

e. Enter text for a help tag, and specify where the help tag should appear.

6. Save your layout in a project subdirectory for localized resources—for example, `Resources/English.lproj/`.

Localizing the Interface

This brief procedure shows how to localize your interface for additional languages or dialects.

For each new locale:

1. Duplicate your original localized resources folder, and rename the new folder appropriately. The name of the nib file inside the new folder should remain the same.

> **Note:** For information on folder-naming conventions for localized resources, read the section "Localized Resources" in *Inside Mac OS X: System Overview* .

2. Open the new nib file using Interface Builder, and for each control in the interface:

a. Enter the localized title string.

b. Resize the control as needed to accomodate the new title.

c. Enter the localized help tag.

Instantiating the Interface

At runtime, you construct the interface as follows:

1. Call `CreateWindowFromNib` to instantiate a localized version of the interface in a Carbon window.

2. Find the individual controls in the window, and embed them inside the dialog pane, adjusting their coordinates as needed.

These tasks are described in detail in "Open" (page 56) and "Embedding Your Controls in a Dialog Pane" (page 68).

See Also

Interface Builder is available on the latest Mac OS X Developer Tools CD, or from the Apple Development Tools site at http://developer.apple.com/tools/.

For general information about localizing human interfaces, see *Getting Started with Internationalization*.

Defining Identifiers and Constants

You need to define several unique identifiers used to locate external resources such as nib files and tickets, and integer constants that are passed to the printing system to specify the dimensions of your pane.

Defining Bundle, Pane, and Nib Identifiers

Listing 6-1 shows how to define several identifiers associated with your bundle property list and your nib file.

Important: If you use the definitions in Listing 6-1 in a real-world project, you should always define these symbols using your own custom identifiers and names.

Listing 6-1 Custom identifiers

```
#define  kMyBundleSignature     "PRDX"                                  // 1
#define  kMyBundleCreatorCode   'PRDX'                                  // 2
#define  kMyBundleIdentifier \
            CFSTR("com.appvendor.pde." kMyBundleSignature)              // 3

#define  kMyPaneKindID  kMyBundleIdentifier                             // 4
#define  kMyNibFile     CFSTR("PDEPrint")                               // 5
#define  kMyNibWindow   CFSTR("PDEPrint")                               // 6
```

Here's what the definitions in Listing 6-1 represent:

1. A string representation of the unique 4-byte signature of an application-hosted printing dialog extension. This signature is used as a suffix when defining keys for extended data in a print settings or page format ticket, as described in "Defining Custom Ticket Keys" (page 64).

2. An OSType representation of the same 4-byte signature. An application uses this code to retrieve its extended data, as described in "Data Retrieval" (page 79). Printer module–hosted printing dialog extensions do not need to define this constant.

3. A Core Foundation string used by your printing dialog extension to specify its bundle when calling the function `CFBundleGetBundleWithIdentifier`. This string must be the same unique identifier used in the `CFBundleIdentifier` property.

4. A Core Foundation string that represents your custom pane. This string should be a unique identifier in the form of a Java-style package name. In "Prologue" (page 51), this string is passed back to the printing system.

Note: If your custom pane implements one of the standard feature sets listed in Table 1-1 (page 15), you must define `kMyPaneKindID` using the appropriate Apple-defined kind ID in `PMPrintingDialogExtensions.h`.

5. A Core Foundation string that represents the name of your nib file (without the `.nib` extension.) In "Open" (page 56), this string is passed to the function `CreateNibReferenceWithCFBundle`.

6. A Core Foundation string that represents the name of your nib-based Carbon window. In "Open" (page 56), this string is passed to the function `CreateWindowFromNib`.

Defining Custom Ticket Keys

A ticket key is a unique string identifier that's used to access a data item in a print job ticket.

Applications

An application using a printing dialog extension must observe some restrictions when defining ticket keys for either the print settings or page format ticket.

A key must

- start with the appropriate prelude string defined below in Table 6-1
- end with a custom 4-byte code that represents the data item

Note: The code bytes must be ASCII characters in the range 0x20-0x7F.

Table 6-1 Prelude strings for ticket keys defined by applications

Printing dialog	Associated ticket	Prelude string for ticket key
Page Setup	Page format	`com.apple.print.PageFormatTicket.`
Print	Print settings	`com.apple.print.PrintSettingsTicket.`

Listing 6-2 shows how you can define custom keys for the print settings or page format tickets, assuming that you are storing a single data item in each ticket.

Listing 6-2 Examples of two custom ticket keys

```
#define kMyAppPageFormatKey \
        CFSTR("com.apple.print.PageFormatTicket." kMyBundleSignature)

#define kMyAppPrintSettingsKey \
        CFSTR("com.apple.print.PrintSettingsTicket." kMyBundleSignature)
```

Printer Modules

If your printer module uses a printing dialog extension to implement one of the standard sets of printing features listed in Table 1-1 (page 15), you should use the Apple-defined ticket keys for those features.

If you are implementing a custom printing feature, you need to define a custom ticket key for the feature. By convention, a custom key should be an identifier in the form of a Java-style package name. For example, if you are writing a printer module for a printer that has a custom toner-saving option, you might define the ticket key as follows:

```
#define kMySaveTonerKey CFSTR("com.mycompany.pm.savetoner")
```

Note: The restrictions on ticket keys described in "Applications" (page 64) do not apply to ticket keys defined for printer modules.

Providing the Dimensions of Your Custom Pane

The printing system needs to know the vertical and horizontal extent (in pixel units) of your custom pane. A printing dialog extension passes these values back to the printing system when its prologue function is called.

The values you provide can be arbitrarily large. The printing system takes the values into account—along with a number of other constraints—when it determines the actual size of your pane.

Note: The printing system adjusts the height of the dialog to accomodate your custom pane, but not the width. This behavior might change in a future version of Mac OS X, so you should always supply the actual width of your pane.

Listing 6-3 illustrates how to define the two constant values passed to the printing system in "Prologue" (page 51).

Listing 6-3 Vertical and horizontal extent of your custom pane in pixels

```
enum {
    kMyMaxV = 80,
    kMyMaxH = 478
};
```

In Mac OS X version 10.2 and later, you can adjust the dimensions of your pane when the initialize function is called. If you do so, and the user displays your pane, the dialog will reflect the adjusted height. For example, you might decide to increase the height of your pane and display additional controls whenever the destination printer is a PostScript printer. For information about using this feature, see "Initialize" (page 53).

Defining a Custom Context

The use of contexts in a printing dialog extension is discussed in "Defining a Context" (page 50) and in "Prologue" (page 51). The sample project factors its context data into two parts:

- A generic context contains state information used in the generic code that's supplied for you in finished form.
- A custom context contains state information used in the custom code that you need to adapt for your custom pane.

Listing 6-4 shows how you could define your custom context for a pane that contains one or more controls—for example, a checkbox. The data type `MyCustomContextBlock` represents the state of this checkbox in a single instance of your pane.

Listing 6-4 Data types for a custom context

```
typedef struct {
    ControlRef checkbox;
} MyControls;

typedef struct {
    Boolean selected;
} MySettings;

typedef struct {
    MyControls controls;
    MySettings settings;
} MyCustomContextBlock;

typedef MyCustomContextBlock *MyCustomContext;
```

Implementing Your Custom Functions

The following sections describe the custom functions in the sample project. If you're using the sample project as the basis for a real-world project, then you need to implement all the functions presented here. These functions are called from the code described in "Implementing the Required Callbacks" (page 51).

Managing a Custom Context

You don't have to use a custom context. If you do, you need to allocate memory for a new custom context each time a new dialog is created.

Listing 6-5 shows how your printing dialog extension can create and release an instance of its custom context. The generic code described in "Prologue" (page 51) and "Terminate" (page 58) can't perform these tasks, because the implementation details of your custom context are private.

Listing 6-5 Managing an instance of your custom context

```
extern MyCustomContext MyCreateCustomContext()
{
    MyCustomContext context = calloc (1, sizeof (MyCustomContextBlock));      // 1
    return context;
}

extern void MyReleaseCustomContext (MyCustomContext context)
```

```
{
    free (context);                                                    // 2
}
```

Here's what the code in Listing 6-5 does:

1. Allocates zeroed storage for a new custom context. The initialization of your context data is handled in other custom functions.
2. Frees the storage for the custom context.

Providing the Title of your Custom Pane

Listing 6-6 implements a custom function that provides the title of your custom pane. The printing system displays this title in two places in the dialog—the pane pop-up menu and the Summary pane.

This implementation gets a localized copy of the title string using the Core Foundation Bundle Services macro `CFCopyLocalizedStringFromTableInBundle`. The string is retained for re-use, and can be released by passing in `FALSE`.

Listing 6-6 Providing your custom pane title

```
extern CFStringRef MyGetCustomTitle (Boolean stillNeeded)
{
    static CFStringRef sTitle = NULL;

    if (stillNeeded)                                                   // 1
    {
        if (sTitle == NULL)                                            // 2
        {
            sTitle = CFCopyLocalizedStringFromTableInBundle (
                CFSTR("Custom Feature"),                               // 3
                CFSTR("Localizable"),                                  // 4
                MyGetBundle(),                                         // 5
                CFSTR("the custom pane title"));                       // 6
        }
    }
    else
    {
        if (sTitle != NULL)
        {
            CFRelease (sTitle);                                        // 7
            sTitle = NULL;
        }
    }

    return sTitle;
}
```

Here's what the code in Listing 6-6 does:

1. Checks `stillNeeded` to see if the caller wants to get or release the string.
2. Checks to see if a copy of the title string already exists.

3. Supplies the lookup key for the title string.
4. Supplies the name of the localized strings file (without the `.strings` extension) to be searched.
5. Supplies a reference to your plug-in bundle.
6. Supplies a comment to assist translators.
7. Releases the string because the caller passed in `FALSE`.

Embedding Your Controls in a Dialog Pane

Listing 6-7 implements a custom function that embeds your nib-based controls—in this case, a checkbox—inside the dialog pane provided by the printing system.

The utility function `MyEmbedControl` described in "Embedding a Nib-Based Control" (page 85) is used to position and embed each control.

Listing 6-7 Embedding nib-based controls in a dialog pane

```
extern OSStatus MyEmbedCustomControls (
    MyCustomContext context,
    WindowRef nibWindow,
    ControlRef userPane
)

{
    static const ControlID controlID = { kMyBundleCreatorCode, 4001 };          // 1
    OSStatus result = noErr;

    if (context != NULL)
    {
        result = MyEmbedControl (                                               // 2
            nibWindow,
            userPane,
            controlID,
            &(context->controls.checkbox)
        );

        if (context->controls.checkbox != NULL) {
            SetControlValue (                                                   // 3
                context->controls.checkbox,
                context->settings.checkbox
            );
        }
    }

    return result;
}
```

Here's what the code in Listing 6-7 does:

1. Defines the signature and ID for the control. These values should match the signature and ID assigned to this control in the nib file.

2. Embeds the control inside your dialog pane. `MyEmbedControl` passes back a reference to the control, which is saved in the custom context. For an implementation of `MyEmbedControl`, see "Embedding a Nib-Based Control" (page 85).

3. Initializes the value of the control.

Synchronizing User Settings With a Ticket

The purpose of sychronization is explained in "Sync" (page 54). In the sample project, the actual work of synchronization is done in two custom functions described in "Updating Your Pane" (page 69) and "Updating a Ticket" (page 71).

Your custom sync functions need to agree on the data types used to represent your pane settings, both in memory and in the ticket. The sample project uses the data structure `MySettings` in memory, and `CFData` in the ticket.

Updating Your Pane

Listing 6-8 implements a custom function that reads your data from the print settings ticket, and uses it to update the controls in your pane.

Note: If your custom printing dialog extension extends the Page Setup dialog, then this function should update the page format ticket.

Listing 6-8 A custom function to update your pane

```
extern OSStatus MySyncPaneFromTicket (
    MyCustomContext context,
    PMPrintSession session
)

{
    CFDataRef data = NULL;
    CFIndex length = 0;
    OSStatus result = noErr;
    PMTicketRef ticket = NULL;

    result = MyGetTicket (session, kPDE_PMPrintSettingsRef, &ticket);          // 1

    if (result == noErr)
    {
        result = PMTicketGetCFData (                                           // 2
            ticket,
            kPMTopLevel,
            kPMTopLevel,
            kMyAppPrintSettingsKey,
            &data
        );

        if (result == noErr)
        {
            length = CFDataGetLength (data);
```

```
            if (length == sizeof(MySettings))                                   // 3
            {
                CFDataGetBytes (                                                // 4
                    data,
                    CFRangeMake(0,length),
                    (UInt8*) &(context->settings)
                );
            }
            else
            {
                result = kPMKeyNotFound;
            }
        }

        if (result == kPMKeyNotFound)
        {
            context->settings.checkbox = FALSE;                                 // 5
            result = noErr;
        }
    }

    if ((result == noErr) && (context->controls.checkbox != NULL))              // 6
    {
        SetControlValue (
            context->controls.checkbox,
            context->settings.checkbox
        );
    }

    return result;
}
```

Here's what the code in Listing 6-8 does:

1. Gets a reference to the print settings ticket using the utility function described in "Getting a Ticket Reference" (page 87).

   Warning: The printing system retains ownership of the ticket reference you obtain, and it could become invalid later. For this reason, you should use the reference for the task at hand and then discard it. Do not save this reference in your context.

2. Checks the ticket for your custom data item. If the item exists, a reference to the data is passed back. Otherwise the result code `kPMKeyNotFound` is returned. For more information about retrieving data from tickets, see "Accessing Ticket Data" (page 82).

3. Checks to make sure the ticket data has the expected length.

4. Copies the bytes into your custom context.

5. Initializes the setting to its default value. This code executes when the ticket has no data item for this setting, or the data item found is not the correct size.

6. Uses your context to update the control. The Control Manager takes care of redrawing the control.

Updating a Ticket

Listing 6-9 implements a custom function that reads the current values of the controls in your pane, and updates your data in the print settings ticket.

Note: If your custom printing dialog extension extends the Page Setup dialog, then this function should update the page format ticket.

Listing 6-9 A custom function that updates the print settings ticket

```
extern OSStatus MySyncTicketFromPane
(
    MyCustomContext context,
    PMPrintSession session
)

{
    CFDataRef data = NULL;
    OSStatus result = noErr;
    PMTicketRef ticket = NULL;

    result = MyGetTicket (session, kPDE_PMPrintSettingsRef, &ticket);          // 1
    if (result == noErr)
    {
        if (context->controls.checkbox != NULL) {
            context->settings.checkbox =
                GetControlValue (context->controls.checkbox);                  // 2
        }

        data = CFDataCreate (                                                  // 3
            kCFAllocatorDefault,
            (UInt8*) &context->settings,
            sizeof(MySettings)
        );

        if (data != NULL)
        {
            result = PMTicketSetCFData (                                       // 4
                ticket,
                kMyBundleIdentifier,
                kMyAppPrintSettingsKey,
                data,
                kPMUnlocked
            );

            CFRelease (data);
        }
    }

    return result;
}
```

Here's what the code in Listing 6-9 does:

1. Gets a reference to the print settings ticket, using the utility function described in "Getting a Ticket Reference" (page 87).

2. Reads the current value of the control.

   You might choose to validate the control settings at this point. If you find an error, you should provide feedback and return the result code `kPMDontSwitchPDEError` to prevent the user from switching to another pane.

3. Creates a `CFData` representation of your settings data.

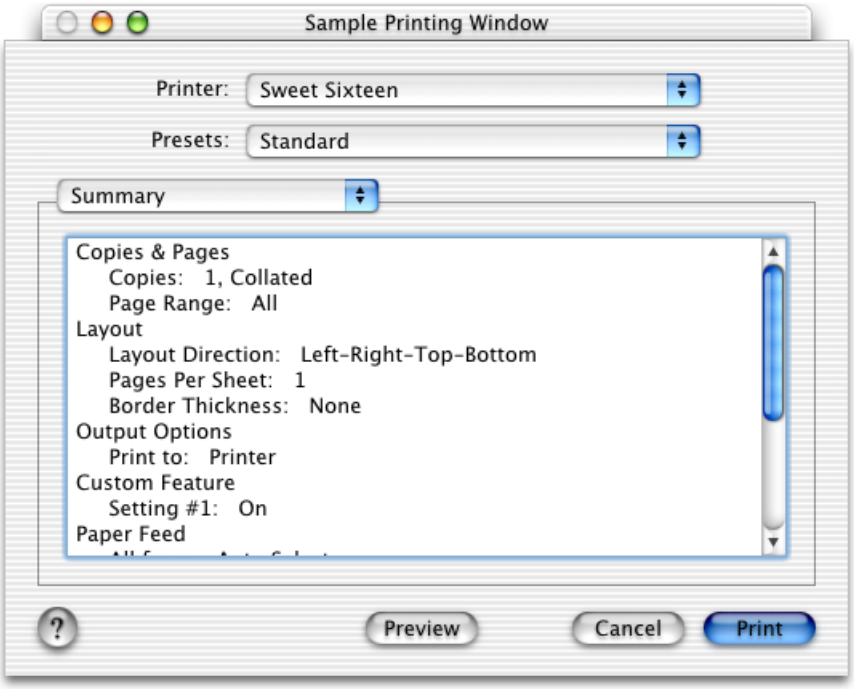
4. Adds a new ticket entry—or updates an existing ticket entry—with the current settings data. For more detailed information about storing data in tickets, see "Accessing Ticket Data" (page 82).

Supplying Summary Text

When a user selects the Summary pane in a printing dialog, each active printing dialog extension is polled for its summary text.

In Figure 6-1, the Summary pane tells the user that the checkbox in the Custom Feature pane is now selected.

Figure 6-1 The Summary pane in the Print dialog



Listing 6-10 implements a custom function that supplies the summary text for the Custom Feature setting in Figure 6-1. This function copies localized descriptions of the title and current value of the setting into two summary arrays.

Note: If your custom pane has more than one setting, you could write a summary function for each setting.

Listing 6-10 Custom function to supply summary text for a setting

```
extern OSStatus MyGetSummaryText (
    MyContext context,
    CFMutableArrayRef titleArray,
    CFMutableArrayRef valueArray
)

{
    CFStringRef title = NULL;
    CFStringRef value = NULL;

    OSStatus result = kPMInvalidPDEContext;                                    // 1

    title = CFCopyLocalizedStringFromTableInBundle (                           // 2
        CFSTR("Setting #1"),
        CFSTR("Localizable"),
        MyGetBundle(),
        CFSTR("the title of our first setting"));

    if (title != NULL)
    {
        SInt16 controlValue = GetControlValue (context->controls.item1);       // 3
        if (controlValue == 0)
        {
            value = CFCopyLocalizedStringFromTableInBundle (
                CFSTR("Off"),
                CFSTR("Localizable"),
                MyGetBundle(),
                CFSTR("the value of setting #1 when not selected"));
        }
        else
        {
            value = CFCopyLocalizedStringFromTableInBundle (
                CFSTR("On"),
                CFSTR("Localizable"),
                MyGetBundle(),
                CFSTR("the value of setting #1 when selected"));
        }

        if (value != NULL)                                                     // 4
        {
            CFArrayAppendValue (titleArray, title);
            CFArrayAppendValue (valueArray, value);
            CFRelease (value);                                                 // 5
            result = noErr;                                                    // 6
        }

        CFRelease (title);
    }

    return result;
}
```

Here's what the code in Listing 6-10 does:

1. Sets the default result code to a non-zero value.
2. Gets the appropriate localized string from the `Localizable.strings` file in the plug-in bundle.
3. Gets the current integer value of the checkbox, assumed to be zero or one.
4. Appends each string to the end of the corresponding array. This transaction is done only if both strings are defined.
5. Releases the value string. Releasing the string is safe to do because `CFArrayAppendValue` retained it.
6. Sets the result code to `noErr`. This is done only after both arrays have been successfully updated.

Handling PostScript Features

Note: This section applies to PostScript printer vendors only. If you are an application developer, you do not need to read this section.

A PostScript printer module can use a printing dialog extension to present features described in a PostScript printer description (PPD) file. At runtime, the printing dialog extension must tell the printing system what PostScript code needs to be emitted for these features.

The printing dialog extension does this by adding entries to a special dictionary called PPDDict. The Ticket Services function `PMTicketGetPPDDict` obtains PPDDict from the print settings ticket.

For each PPD feature it presents, the printing dialog extension adds a key-value pair to PPDDict. The key is the PPD Main keyword for that feature, and the value is the PPD Option keyword that corresponds to the feature's setting. Adding this key-value pair to PPDDict causes the printing system to insert the PostScript code corresponding to that Main-Option keyword pair when it generates the PostScript code for the print job.

For example, consider a PostScript feature that allows the user to route printed pages to an upper or lower output bin. Listing 6-11 shows the PPD entry for this feature.

Listing 6-11 PPD entry for the output bin feature

```
*OpenUI *OutputBin/Output Bin: PickOne
*OrderDependency: 50 AnySetup *OutputBin
*DefaultOutputBin: Upper
*OutputBin Upper/Upper: "1 dict dup /OutputFaceUp false put setpagedevice"
*OutputBin Lower/Lower: "1 dict dup /OutputFaceUp true put setpagedevice"
*CloseUI: *OutputBin
```

If you use a printing dialog extension to present this feature and allow users to modify its setting, you need to do the following:

1. Add an output bin control to your custom pane.

2. Use the mechanism described in "Registering PPD Main Keywords" (page 40) to ensure that output bin appears only in your pane, and not in the Printer Features pane.

3. In your custom `MySyncPaneFromTicket` function, get the current bin setting from the print settings ticket. In your custom `MySyncTicketFromPane` function, update the ticket with the current bin setting.

The following sections show how to implement the synchronization tasks described in step 3.

Getting a PostScript Setting from a Print Settings Ticket

Listing 6-12 implements a function that gets PPDDict in a print settings ticket and uses PPDDict to get the current bin selection.

Listing 6-12 Getting a PostScript setting

```
typedef enum {LOWER, UPPER} MyBinType;

OSStatus MyGetOutputBin (CFTicketRef printSettings, MyBinType *bin)
{
    CFMutableDictionaryRef ppdDict = NULL;
    OSStatus result = noErr;

    result = PMTicketGetPPDDict (
        printSettings, kPMTopLevel, kPMTopLevel, &ppdDict);                  // 1

    if (result == noErr)
    {
        CFStringRef str = NULL;

        if (CFDictionaryGetValueIfPresent (
            ppdDict, CFSTR("OutputBin"), &str)
                && CFGetTypeID (str) == CFStringGetTypeID())                 // 2
        {
            if (CFStringCompare (
                str, CFSTR("Upper"), 0) == kCFCompareEqualTo)                // 3
                { *bin = UPPER; }
            else
                { *bin = LOWER; }
        }
    }
    else
    {
        *bin = UPPER;                                                        // 4
    }

    return result;
}
```

Here's what the code in Listing 6-12 does:

1. Gets a reference to PPDDict in the print settings ticket.

2. Finds the desired key-value pair and verifes that its value is a CFString.

3. Finds out which bin is selected, and passes back the appropriate value.

4. Initializes the setting to a hard-coded default value because there is no valid entry already in PPDDict. A more appropriate implementation would obtain the default setting for this feature from the current PPD file.

Updating a PostScript Setting in a Print Settings Ticket

Listing 6-13 implements a function that gets PPDDict from a print settings ticket, and uses the current bin selection to update the output bin setting in PPDDict.

Listing 6-13 Updating a PostScript setting

```
OSStatus MySetOutputBin (CFTicketRef printSettings, MyBinType bin)
{
    CFMutableDictionaryRef ppdDict = NULL;
    OSStatus result = noErr;

    result = PMTicketGetPPDDict (
        printSettings, kPMTopLevel, kPMTopLevel, &ppdDict);                    // 1

    if (result == noErr)
    {
        switch (bin)                                                           // 2
        {
        case LOWER:
        CFDictionarySetValue (
            ppdDict, CFSTR("OutputBin"), CFSTR("Lower"));

        case UPPER:
        default:
        CFDictionarySetValue (
            ppdDict, CFSTR("OutputBin"), CFSTR("Upper"));
        }
    }

    return result;
}
```

Here's what the code in Listing 6-13 does:

1. Gets a reference to PPDDict from the print settings ticket.

2. Updates the bin setting in PPDDict.

Integration Tasks

Your printing dialog extension is taking shape, and now you're wondering how to integrate it with your other software. This chapter shows you how applications and printer modules deploy and communicate with their printing dialog extensions.

The first two sections "Integrating With an Application" (page 77) and "Integrating With a Printer Module" (page 80) describe tasks that include installation, localization, and registration.

The last section "Accessing Ticket Data" (page 82) desribes data communication tasks.

Integrating With an Application

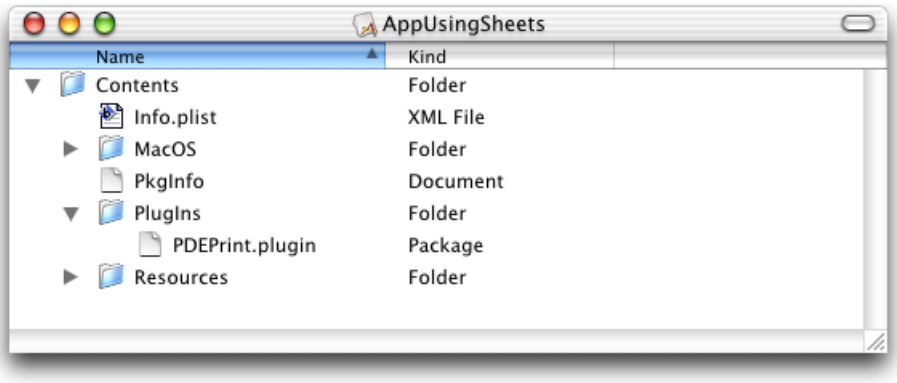
A printing dialog extension hosted by an application should be packaged inside the application bundle, and should be localized for the same languages and regions as the application. The host and plug-in share settings data using several different printing APIs.

Installation

You should install an application-hosted printing dialog extension somewhere inside the application bundle.

If you want to locate the plug-in at runtime using the function `CFBundleCopyBuiltInPlugInsURL`, you should install it inside a subfolder named PlugIns, as illustrated in Figure 7-1.

Figure 7-1 A plug-in installed inside an application bundle



In the sample project, the target AppUsingSheets has a build phase called Copy Files that installs the plug-in in this manner.

Localization

If you localize a printing dialog extension for a specific language or region, make sure the application has an `.lproj` directory for the same locale.

For example, the application bundle in Figure 7-1 (page 77) contains directories named `English.lproj` for both the application and the printing dialog extension.

Registration

An application needs to register its printing dialog extension with Core Foundation Plug-in Services. Registration should be done once, prior to displaying the printing dialog for the first time.

Listing 7-1 implements a function that an application can use to register a printing dialog extension.

Listing 7-1 Locating and registering a printing dialog extension

```
OSStatus MyRegisterPDE (CFStringRef plugin)                                    // 1
{
    OSStatus result = kPMInvalidPDEContext;

    CFURLRef pluginsURL = CFBundleCopyBuiltInPlugInsURL (                      // 2
        CFBundleGetMainBundle());

    if (pluginsURL != NULL)
    {
        CFURLRef myPluginURL = CFURLCreateCopyAppendingPathComponent (         // 3
            NULL,
            pluginsURL,
            plugin,
            false);

        if (myPluginURL != NULL)
        {
            if (CFPlugInCreate (NULL, myPluginURL) != NULL) {                  // 4
                result = noErr;
            }

            CFRelease (myPluginURL);
        }

        CFRelease (pluginsURL);
    }

    return result;
}
```

Here's what the code in Listing 7-1 does:

1. Receives a string that names a printing dialog extension bundle, such as `"PDEPrint.plugin"`.
2. Gets a Core Foundation URL that specifies the location of the folder that contains the bundle.

3. Appends the bundle name to this URL, and returns a new URL—a fully qualified path to the printing dialog extension.

4. Registers the plug-in with Core Foundation Plug-in Services. The function `CFPlugInCreate` takes two arguments—an optional allocation function, and a fully qualified path.

Note: In this example, the CFPlugIn object returned by `CFPlugInCreate` is never explicitly released.

Data Retrieval

All printing dialog extensions store their private settings data in a job ticket, as described in "Synchronizing User Settings With a Ticket" (page 69). Because applications do not have direct access to tickets, a host application must query a `PMPageFormat` or `PMPrintSettings` object to retrieve the data.

The section "Defining Custom Ticket Keys" (page 64) explains how an application-hosted printing dialog extension needs to define the keys it uses to store its extended data in a ticket. To retrieve this data later, an application must supply the same 4-byte code found at the end of the ticket key. For example, if the ticket key ends with `PRDX` then the application must use the 4-byte code `PRDX`. The bytes must be ASCII characters in the range 0x20-0x7F.

Listing 7-2 illustrates how an application retrieves data from a `PMPrintSettings` object.

Listing 7-2 Retrieving extended data from a print settings ticket

```
OSStatus MyGetPDEData (
    PMPrintSettings printSettings,
    MySettings *mysettingsP
)

{
    UInt32 bufferSize = sizeof(*mysettingsP);                          // 1
    UInt32 extendedDataSize = 0;
    OSStatus result = noErr;

    result = PMGetPrintSettingsExtendedData (                          // 2
                printSettings,
                kMyBundleCreatorCode,                                  // 3
                &extendedDataSize,
                kPMDontWantData);

    if (result == noErr)
    {
        if (bufferSize == extendedDataSize)                            // 4
        {
            result = PMGetPrintSettingsExtendedData (                  // 5
                printSettings,
                kMyBundleCreatorCode,
                kPMDontWantSize,
                mysettingsP);
        }
        else
        {
```

```
            result = kPMInvalidPDEContext;                                  // 6
        }
    }

    return result;
}
```

Here's what the code in Listing 7-2 does:

1. Declares an integer whose value is the expected number of bytes.
2. Checks for the existence of your extended data. On return, `bufferSize` contains the actual number of bytes of extended data in the ticket.
3. Provides the 4-byte code used to find the data.
4. Checks to see if the actual byte count is equal to the expected byte count. If so, the ticket contains your data.
5. Copies the data directly into the caller's buffer.
6. Sets the result code to indicate an exception. Another course of action might be to return default settings.

Integrating With a Printer Module

For printing dialog extensions hosted by printer modules, the integration tasks are conceptually similar to those discussed in "Integrating With an Application" (page 77). However, there are some important differences in the way these tasks are performed.

Installation

The Mac OS X printing system expects all printer modules, and their printing dialog extensions, to reside in `/Library/Printers/` or in one of its subfolders. For housekeeping purposes, printer vendors generally place all their vendor-specific files in a subfolder.

If a printing dialog extension is used by a single printer module, it's a good idea to install it in a PlugIns folder inside the printer module bundle.

Custom printing dialog extensions that implement PostScript features should reside in `/Library/Printers/PPD Plugins/`. For more information about using printing dialog extensions to implement PostScript features, see "Registering PPD Main Keywords" (page 40) and "Handling PostScript Features" (page 74).

Registration

At runtime, the printing system asks a printer module for the location of each of its printing dialog extensions relative to the base path `/Library/Printers/`.

All printer modules must supply a callback function of type `PMCreatePrintingDialogExtensionsPathsProcPtr` that provides these locations. The printing system calls this function after it loads your printer module.

With this information, the printing system performs two important services:

- It registers each extension with Core Foundation Plug-in Services.
- It remembers the custom extensions that must be loaded for the default printer when the Print dialog is displayed.

Listing 7-3 shows how to write a function that creates an empty array, inserts a single path expression, and passes the array to the printing system.

Listing 7-3 Providing the locations of the printing dialog extensions hosted by a printer module

```
OSStatus MyCreatePrintingDialogExtensionsPaths
(
    PMContext pmContext,
    CFArrayRef *pdePaths
)

{
    OSStatus status = noErr;

    CFMutableArrayRef cfArray = CFArrayCreateMutable (
        CFAllocatorGetDefault(), 0, &kCFTypeArrayCallBacks);                    // 1

    if ( cfArray != NULL && pdePaths != NULL)
    {
        CFArrayAppendValue (cfArray,
            CFSTR("MyPrinterCo/PrintQuality.plugin"));                          // 2
    }
    else
        status = memFullErr;

    *pdePaths = cfArray;                                                        // 3
    return (status);
}
```

Here's what the code in Listing 7-3 does:

1. Creates a mutable array of variable size with zero entries. Core Foundation arrays may contain references to mixed types, but in this case the printing system expects an array of strings.
2. Appends to the array a relative path expression—note the absence of a leading "/". The implied full path specifies the location of a custom printing dialog extension that's associated with this printer module.
3. Passes the array back to the printing system, which is responsible for releasing the array and its contents when it is no longer needed.

Data Retrieval

Printer modules have direct access to tickets, and can easily find the printer settings placed there by printing dialog extensions.

For each ticket item, the printer module and its printing dialog extension must agree on the ticket type, the key, and the data type of the item. This agreement ensures that the printer module can locate and retrieve its data.

The section "Defining Custom Ticket Keys" (page 64) explains how ticket keys for data items that represent custom printing features are defined.

The section "Getting a Ticket Value" (page 83) explains in greater detail how printer modules retrieve data from a ticket.

Mac OS X applications can print without using dialogs, so your printing dialog extension might never get called. Consequently, printer modules should be prepared to handle missing ticket items and supply appropriate default values as needed.

Note: In Mac OS X version 10.2 and later, printing dialog extensions that use tickets to pass data to a printer module are limited to the following Core Foundation base types to represent the data—CFString, CFBoolean, CFDictionary, CFNumber (integer only), and CFArray.

Accessing Ticket Data

This section explains how printing plug-ins—including printing dialog extensions—can access their custom data in a ticket.

Setting a Ticket Value

Ticket Services provides a number of functions for setting items in a ticket based on the item's data type, such as `PMTicketSetSInt32` and `PMTicketSetCFString`. You should use the appropriate `PMTicketSet` function for the data type you are setting.

If you call the appropriate `PMTicketSet` function and the item does not exist, a new one is added to the ticket. If the item already exists, its value is updated. If the item is locked and can't be updated, the status `kPMItemIsLocked` is returned. Items can be locked by clients. You can check the locked state of a ticket by calling the function `PMTicketGetLockedState`.

The first four parameters of every `PMTicketSet` function are the following:

1. The ticket (`PMTicketRef`) in which to put the data.
2. A string (`CFStringRef`) that specifies the client ID.

   You should provide a string that uniquely identifies your plug-in, such as its bundle identifier.
3. A string (`CFStringRef`) that specifies the key for this ticket item.

When available, you should provide one of the keys defined in the Carbon Printing Manager for ticket items. If you supply a custom key, you should avoid using the same key in different parts of the job ticket. Otherwise, you may not be able to retrieve the item later, due to the search semantics. The ticket is searched from the job ticket level down (page format, print settings, and so forth). `PMTicketGet` functions return the value for the first instance of the key.

4. The item data.

   The last parameter to every `PMTicketSet` function is a `Boolean` that specifies whether the item is locked (`TRUE`) or unlocked (`FALSE`).

Listing 7-4 shows how to use a `PMTicketSet` function to insert or update an item (a key-value pair) in a ticket. In this case, the item is unlocked (`kPMUnlocked`) and the client setting the item is identified by the constant `kMyPrinterModuleID`.

Listing 7-4 Setting an integer value in a ticket

```
#define kMyPrinterModuleID CFSTR("com.mycompany.myprintermodule")

result = PMTicketSetSInt32 (
    ticketRef,
    kMyPrinterModuleID,
    kPMQualityKey,
    qualityValue,
    kPMUnlocked
);
```

Getting a Ticket Value

As a convenience for developers, Ticket Services provides a number of `PMTicketGet` functions for specific types of ticket data—for example, `PMTicketGetSInt32` and `PMTicketGetCFString`. You can also use the generic function `PMTicketGetItem`.

The first four parameters of every `PMTicketGet` function specify

- the ticket reference (`PMTicketRef`)
- the document number (`UInt32`) associated with the item
- the page number (`UInt32`) associated with the item
- the lookup key (`CFStringRef`) for the item

Note: Ticket Services currently does not use the document and page parameters. Instead, you should pass the constant `kPMTopLevel` for each parameter.

The other parameters in a `PMTicketGet` function are specific to the data you are fetching. You need to specify a pointer to the storage for the data to be fetched by the function.

Listing 7-5 shows how a printing dialog extension hosted by a printer module can get a `CFString` value stored in a print settings ticket.

Listing 7-5 Getting a `CFString` value in a print settings ticket

```
CFStringRef myCFString;

err = PMTicketGetCFString (printSettingsTicket,
                           kPMTopLevel,
                           kPMTopLevel,
                           kMyTicketKey,
                           &myCFString);
```

Listing 7-6 shows how to retrieve the unadjusted paper rectangle from a paper info ticket that's inside a print settings ticket. First, you need to call the function `PMTicketGetTicket` to retrieve the paper info ticket from the print settings ticket. If there isn't an error when you retrieve the paper info ticket, use the function `PMTicketGetPMRect` to get the value.

Listing 7-6 Getting the unadjusted page rectangle in a paper info ticket

```
PMRect unadjustedPage;
PMTicketRef paperInfoTicket = NULL;

result = PMTicketGetTicket (printSettingsTicket,
                           kPMPaperInfoTicket,
                           kPMTopLevel,
                           &paperInfoTicket);

if (result == noErr && paperInfoTicket != NULL) {
        result = PMTicketGetPMRect (paperInfoTicket,
                           kPMTopLevel,
                           kPMTopLevel,
                           kPMUnadjustedPageRectKey,
                           &unadjustedPage);
}
```

Further Reading

For more information about the functions in Ticket Services, see *Ticket Services Reference*.

Utility Functions

In your custom code, you might need to embed a control in a dialog pane, get a ticket reference, or implement a help event handler.

This appendix explains how to perform these tasks. You should be able to use this sample code in a real-world project with little or no modification.

Embedding a Nib-Based Control

Listing A-1 implements a function that embeds a nib-based control inside the dialog pane provided by the printing system.

This function takes three parameters—a nib-based window, a Carbon container control (called a user pane), and the ID of the nib-based control being embedded.

During execution, this function

- gets a reference to the nib-based control
- positions the control with respect to the dialog window
- embeds the control inside the dialog user pane
- returns a reference to the control being embedded

Listing A-1 Embedding a nib-based control inside a dialog pane

```
extern OSStatus MyEmbedControl (
    WindowRef nibWindow,
    ControlRef userPane,
    const ControlID *controlID,
    ControlRef* outControl
)

{
    ControlRef control = NULL;
    OSStatus result = noErr;

    *outControl = NULL;

    result = GetControlByID (nibWindow, controlID, &control);                    // 1

    if (result == noErr)
    {
        SInt16 dh, dv;
        Rect nibFrame, controlFrame, paneFrame;
```

```
        (void) GetWindowBounds (nibWindow, kWindowContentRgn, &nibFrame);
        (void) GetControlBounds (userPane, &paneFrame);
        (void) GetControlBounds (control, &controlFrame);

        dh = ((paneFrame.right - paneFrame.left) -
                (nibFrame.right - nibFrame.left))/2;                    // 2

        if (dh < 0) dh = 0;

        dv = ((paneFrame.bottom - paneFrame.top) -
                (nibFrame.bottom - nibFrame.top))/2;                    // 3

        if (dv < 0) dv = 0;

        OffsetRect (                                                    // 4
            &controlFrame,
            paneFrame.left + dh,
            paneFrame.top + dv
        );

        (void) SetControlBounds (control, &controlFrame);

        result = SetControlVisibility (control, TRUE, FALSE);          // 5

        if (result == noErr)
        {
            result = EmbedControl (control, userPane);                  // 6

            if (result == noErr)
            {
                *outControl = control;                                  // 7
            }
        }
    }

    return result;
}
```

Here's what the code in Listing A-1 does:

1. Gets a reference to the desired control. This control already exists in the nib window that contains your custom interface.
2. Finds the delta needed to position the control such that the nib-based interface is horizontally center-aligned inside the dialog pane.Finds the delta needed to position the control such that the nib-based interface is vertically center-aligned inside the dialog pane.
3. Adjusts the coordinates of the top-left corner of the control, so that the control is positioned correctly with respect to the dialog pane.
4. Makes sure the control is visible if the dialog pane is displayed.
5. Embeds the control inside the dialog pane. As a side effect, the printing system now owns the control reference—your printing dialog extension should not release it.
6. Passes the embedded control back to the caller.

Note: A printing dialog is a Carbon window, and the position of a graphics object inside the window is specified using local QuickDraw coordinates.

Getting a Ticket Reference

To get a ticket reference, you need to implement a function that calls `PMSessionGetData` for the session object associated with the current print job.

Warning: The printing system retains ownership of the ticket reference you obtain, and it could become invalid later. For this reason, you should use the reference for the task at hand and then discard it. Do not save or retain the reference in your context.

Table A-1 lists the four standard ticket identifiers defined for use by printing dialog extensions.

Table A-1 Ticket identifiers used by printing dialog extensions

Identifier (CFStringRef)	Ticket	Comments
`kPDE_PMPrintSettingsRef`	Print settings	Used by application and printer module printing dialog extensions.
`kPDE_PMPageFormatRef`	Page format	Used by application and printer module printing dialog extensions.
`kPDE_PMJobTemplateRef`	Job template	Available only during some printer module printing dialog extension routines.
`kPDE_PMPrinterInfoRef`	Printer info	Available only during some printer module printing dialog extension routines.

Listing A-2 illustrates how to write a function that constructs and passes back a ticket reference.

Listing A-2 A utility function that gets a ticket reference

```
extern OSStatus MyGetTicket
(
    PMPrintSession  session,
    CFStringRef     ticketID,
    PMTicketRef*    ticketPtr
)

{
    OSStatus result = noErr;
    CFTypeRef type = NULL;
    PMTicketRef ticket = NULL;

    *ticketPtr = NULL;

    result = PMSessionGetDataFromSession (session, ticketID, &type);          // 1
```

```
    if (result == noErr)
    {
        if (CFNumberGetValue (                                                  // 2
            (CFNumberRef) type, kCFNumberSInt32Type, (void*) &ticket))
        {
            *ticketPtr = ticket;                                                // 3
        }
        else {
            result = kPMInvalidValue;
        }
    }

    return result;
}
```

Here's what the code in Listing A-2 does:

1. Searches the printing session object for a specialized job ticket. The value passed back is a generic object reference. The printing system provides the session object when it calls the sync function.
2. Gets a numeric value that represents an opaque ticket reference. This function returns `true` if the operation was successful.
3. Passes back the ticket reference to the caller.

Handling the Help Event in a Printing Dialog

This section explains how your printing dialog extension can override the default behavior of the help button in a printing dialog. Of course, you don't want to override the help button unless your pane is visible.

The tasks are straightforward:

- Add a field for a handler reference to the context described in "Defining a Context" (page 50).
- In the open function described in "Open" (page 56), install a Carbon event handler that handles the command event associated with the help button. Save the handler reference in the context.
- In the close function described in "Close" (page 58), remove the handler to restore the default behavior of the help button.

Handling a Window Command Event

Listing A-3 shows how to implement a Carbon event handler that detects and handles a help-button click in a printing dialog. The installing function—described in "Installing a Help Event Handler" (page 89)—configures this handler for command events only.

Listing A-3 An event handler for the help event in a printing dialog

```
static OSStatus MyHandleHelpEvent                                               // 1
(
    EventHandlerCallRef call,
```

```
    EventRef event,
    void *userData
)

{
    HICommand commandStruct;                                    // 2
    OSStatus result = eventNotHandledErr;                       // 3

    GetEventParameter (                                         // 4
        event, kEventParamDirectObject,
        typeHICommand, NULL, sizeof(HICommand),
        NULL, &commandStruct
    );

    if (commandStruct.commandID == 'help') {                    // 5
        result = MyDisplayHelp();                               // 6
    }

    return result;
}
```

Here's what the code in Listing A-3 does:

1. Declares the name and parameters for a Carbon event handler of type `EventHandlerProcPtr`. In this implementation, `event` is the only parameter actually used.
2. Declares a data structure for a command event. See the Carbon Event Manager reference for details.
3. Initializes the return code. The usual default value `noErr` is not appropriate here, because in this context `noErr` means "event handled".
4. Retrieves the event data. Since this handler is installed at the window level, the event could contain one of several different commands.
5. Checks the command signature to see if the help button was clicked.
6. Calls a custom function you implement to display the help book for your application or printer module.

Installing a Help Event Handler

Listing A-4 shows how to write a function that installs a help event handler, specifying the printing dialog as the event target.

Listing A-4 Installing a help event handler

```
#define kMyNumberOfEventTypes   1

extern OSStatus MyInstallHelpEventHandler
(
    WindowRef inWindow,
    EventHandlerRef *outHandler
)

{
```

```
    static const EventTypeSpec sEventTypes [kMyNumberOfEventTypes] =          // 1
    {
        { kEventClassCommand, kEventCommandProcess }
    };

    OSStatus result = noErr;
    EventHandlerRef handler = NULL;
    EventHandlerUPP handlerUPP = NewEventHandlerUPP (MyHandleHelpEvent);      // 2

    result = InstallWindowEventHandler (                                      // 3
        inWindow,
        handlerUPP,
        kMyNumberOfEventTypes,
        sEventTypes,
        NULL,
        &handler
    );

    *outHandler = handler;                                                    // 4
    return result;
}
```

Here's what the code in Listing A-4 does:

1. Defines an array of event type specifiers. This array represents the set of event types you want to handle. In this case, you need to handle only one type.
2. Creates the universal procedure pointer used by the Carbon Event Manager to call your event handler.
3. Installs the handler. On return, `context->handler` contains the `EventHandlerRef` used later to remove the event handler.
4. Stores this handler reference in the context.

Removing a Help Event Handler

Listing A-5 shows how to write a function that removes the help event handler described in "Installing a Help Event Handler" (page 89).

Listing A-5 Removing a help event handler

```
extern OSStatus MyRemoveHelpEventHandler (
    EventHandlerRef *helpHandlerP,
    EventHandlerUPP *helpHandlerUPP
)

{
    OSStatus result = noErr;

    if (*helpHandlerP != NULL)
    {
        result = RemoveEventHandler (*helpHandlerP);                          // 1
        *helpHandlerP = NULL;                                                 // 2
    }
```

```
    if (*helpHandlerUPP != NULL)
    {
        DisposeEventHandlerUPP (*helpHandlerUPP);                    // 3
        *helpHandlerUPP = NULL;                                      // 4
    }

    return result;
}
```

Here's what the code in Listing A-4 does:

1. Removes the help event handler.
2. Deletes the reference to the handler in your context.
3. Deallocates the UPP used by the Carbon Event Manager to call your help event handler.
4. Deletes the reference to this UPP in your context.

Further Reading

For more information about the Carbon event model and writing Carbon event handlers, and see *Carbon Event Manager Programming Guide*.

Printing Plug-in Header Functions

All printing plug-ins—including printing dialog extensions—must implement the three functions defined in the `PMPluginHeader` interface. These functions perform reference counting and provide version information for the larger `PlugInIntfVTable` interface.

This appendix explains how to perform these tasks. You should be able to use this sample code in a real-world project with little or no modification.

PMRetain

Listing B-1 implements a function that retains (increments the reference count of) an instance of the `PlugInIntfVTable` interface.

Listing B-1 A retain function for the `PlugInIntfVTable` interface

```
static OSStatus MyPMPluginRetain (PMPlugInHeaderInterface* this)
{
    MyPDEInstance* instance = (MyPDEInstance*) this;                    // 1
    OSStatus result = noErr;

    if (instance != NULL) {
        ++instance->refCount;                                           // 2
    }

    return result;
}
```

Here's what the code in Listing B-1 does:

1. Defines a pointer to an instance of the `PlugInIntfVTable` interface. This is the same instance the query interface function supplied.
2. Increments the reference count for this instance.

PMRelease

Listing B-2 implements a function that releases (decrements the reference count of) an instance of the `PlugInIntfVTable` interface, and frees the instance if the reference count reaches zero.

Listing B-2 A release function for the `PlugInIntfVTable` interface

```
static OSStatus MyPMRelease (
```

```
    PMPlugInHeaderInterface** this
)

{
    MyPDEInstance* instance = (MyPDEInstance*) *this;
    ULONG refCount = 0;
    OSStatus result = noErr;

    *this = NULL;

    if(instance != NULL)
    {
        refCount = --instance->refCount;

        if (refCount == 0)
        {
            free (instance);
            MyFreeBundle();
            MyFreeTitle();
        }
    }

    return result;
}
```

Here's what the code in Listing B-2 does:

1. Defines a pointer to an instance of the `PlugInIntfVTable` interface. This is the same instance the query interface function supplied.
2. Defines a variable for the updated reference count, and sets its default value to 0.
3. Clears the caller's instance pointer.
4. Decrements the reference count for this instance.
5. If the reference count is zero, deallocates storage for the instance.
6. Releases our bundle reference, in case the plug-in is being unloaded. For more information about `MyFreeBundle`, see the comments in the sample project that accompanies this book.

PMGetAPIVersion

Listing B-3 implements a function that supplies API version information to the printing system.

Listing B-3 An API version function for the `PlugInIntfVTable` interface

```
static OSStatus MyPMPluginGetAPIVersion (
    PMPlugInHeaderInterface *this,
    PMPlugInAPIVersion *versionPtr                                    // 1
)

{
```

```
    OSStatus result = noErr;

    versionPtr->buildVersionMajor = kPDEBuildVersionMajor;                    // 2
    versionPtr->buildVersionMinor = kPDEBuildVersionMinor;
    versionPtr->baseVersionMajor = kPDEBaseVersionMajor;
    versionPtr->baseVersionMinor = kPDEBaseVersionMinor;

    return result;
}
```

Here's what the code in Listing B-3 does:

1. Receives the address of a structure that the caller uses for version control information.
2. Assigns four constant values to fields in the structure. The constants are defined in `PMPrintingDialogExtensions.h` for use in this function.

Document Revision History

This table describes the changes to *Extending Printing Dialogs*.

Date	Notes
2006-10-03	Added a link to Cocoa printing documentation.
2005-04-29	Updated links to the sample project.
2002-09-01	Updated for Mac OS X version 10.2.
2002-06-01	First release.

Document Revision History