Supporting Printing in Your Carbon Application

Carbon > Printing

2004-08-31

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Supporting Printing in Your Carbon Application

This document shows you how to set up a Carbon application to print in Mac OS X using the Carbon Printing Manager. The **Carbon Printing Manager** defines a programming interface that Carbon applications use for printing their documents. For Carbon applications, this programming interface replaces that of the original Printing Manager. The original Printing Manager—referred to as the old Printing Manager in this document—was introduced with the very first release of Macintosh system software. The Carbon Printing Manager allows applications to print both in Mac OS 8 and 9 with existing printer drivers and in Mac OS X with new printer drivers.

You should read this document if you are a developer who wants to support printing from your Carbon application. This document:

- describes the Carbon Printing Manager concepts you need to know to start coding your application
- provides examples of how to set up printing in a new application
- discusses how to revise an existing application

To get the most from this document, you should first read *Mac OS X Printing System Overview*, which describes the various portions of the printing system, including the user interface and the printing architecture.

This document describes how to write only the application portion of generating a print job. It does not describe how to write printer modules, converters, I/O modules, or printing dialog extensions. See the Printing Carbon Documentation for information on these other topics.

The document is divided into the following chapters:

- Chapter 2 (page 9) contains details on how the Carbon Printing Manager works and describes the concepts you need to use the Carbon Printing Manager in your application.
- Chapter 3 (page 25) shows you how to use the Carbon Printing Manager to support printing in a Carbon application. The chapter contains sample code you can customize for your application.
- Chapter 4 (page 49) explains what you need to do to revise an existing non-Carbon Mac OS 9 application so that it can print in either Mac OS 9 or Mac OS X.

See also *Carbon Printing Manager Reference*.

Printing Concepts for Carbon Developers

The Carbon Printing Manager is a collection of system software functions that your application can use to print to any type of supported printer. When printing, your application calls the same Carbon Printing Manager functions regardless of the type of printer selected by the user. An application that uses the Carbon Printing Manager can print in Mac OS 8 and 9 with existing printer drivers and in Mac OS X with new printer drivers.

This chapter provides an overview of the key concepts you need to support printing with the Carbon Printing Manager. It includes the following sections:

- "Overview of Printing Terminology" (page 9) defines the more frequently used printing terms.
- "High-Level Printing Tasks" (page 11) lists the high-level tasks that a Carbon application must to do support printing.
- "Printing Objects" (page 13) provides information about the data types you use to keep track of user selections and other data related to a print job.
- "Printing Functions" (page 14) gives an overview of the key functions needed by your application to support printing.
- "The Print Loop" (page 18) describes the code that sends a print job to a printer queue.
- "Sequence, Scope, and Usage" (page 18) provides guidelines for using printing functions.
- "Page and Paper Rectangles" (page 20) discusses the page and paper boundaries.
- "If You've Used the Old Printing Manager" (page 22) summarizes the differences between the old Printing Manager and the Carbon Printing Manager.

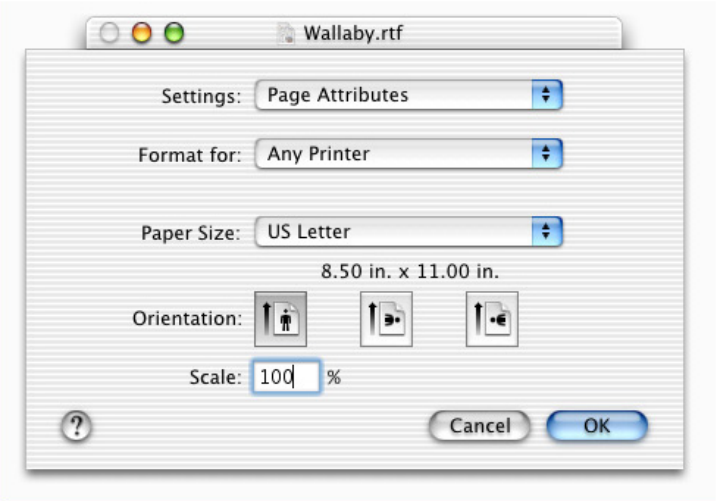
Overview of Printing Terminology

There are several terms that you'll see repeatedly in the Carbon Printing Manager documentation: page format, print settings, formatting printer, default printer, and print job.

Page format describes how pages of a document should be printed, and includes such information as paper size and orientation. Although an application can programmatically set up the page format, most applications allow users to set options that control the page format in the Page Setup dialog.

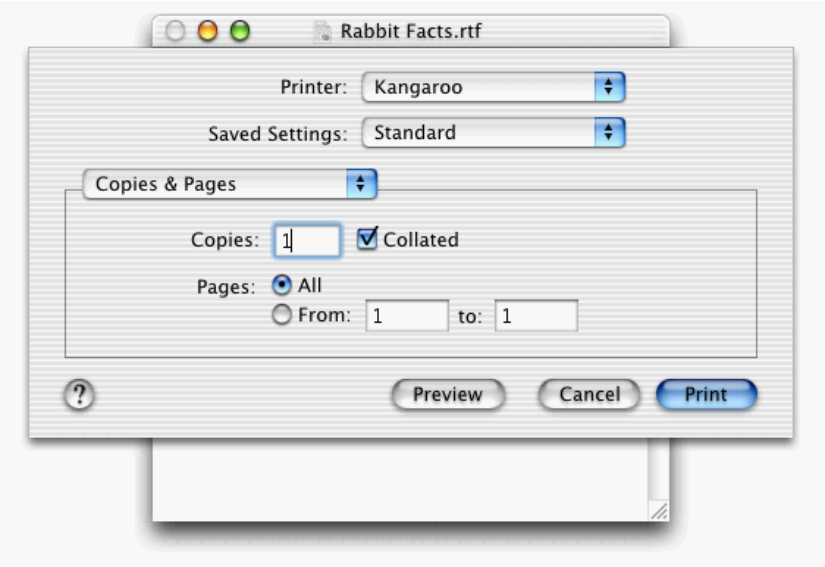
The default page format settings are determined by the formatting printer. The **formatting printer** is the one that is displayed in the "Format for" pop-up menu in the Page Setup dialog. The default formatting printer is the generic Any Printer, as shown in the Page Setup dialog in Figure 2-1.

Figure 1-1 The Page Setup Dialog



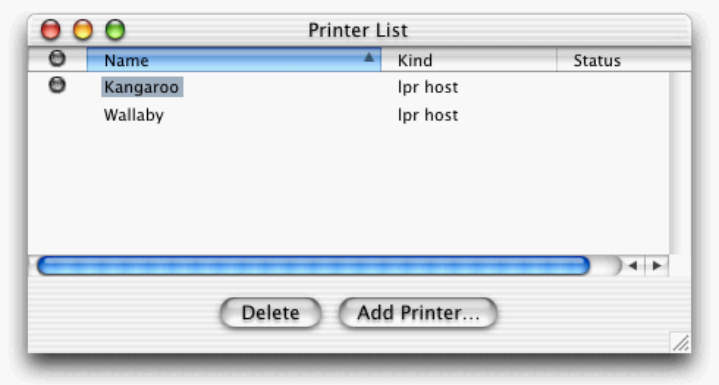
Print settings control the execution of a print job on a specific printer, and include such information as the number of copies, which pages to print, and the number of pages per sheet. As with the page format, an application can programmatically set print settings but usually allows users to open the Print dialog and make print settings instead. Figure 2-2 shows the Copies & Pages pane of the Print dialog.

Figure 1-2 The Copies & Pages pane of the Print dialog



The default print settings, as well as any constraints on those values, are determined by the printer module for the default printer. Before the Print dialog opens, the **default printer** in Mac OS X is the printer that is currently selected in Print Center. In Mac OS 9, it is the printer that is currently selected in the Chooser. Figure 2-3 shows Print Center with the default printer set to a printer named Kangaroo.

Figure 1-3 Print Center with the default printer set to Kangaroo



A **print job** consists of two items:

- the drawing commands that describe a document
- the settings that control printing the document and keep track of it once the job has been added to a printer's queue

High-Level Printing Tasks

The high-level tasks needed to print in a Carbon application are

- setting up the page format
- setting up the print settings
- printing the job

As you'll see later in this chapter, Carbon Printing Manager data structures and functions are grouped to support these tasks.

The way you implement the high-level printing tasks depends on whether the printing tasks are driven by the Page Setup and Print dialogs. In most document-based applications, such as a text editing or drawing application, high-level printing tasks are driven by user interaction with these dialogs. The user can choose Page Setup and Print from the File menu and change settings. A document prints when the user clicks Print in the Print dialog.

An application can also support printing without requiring a user to open either the Page Setup or Print dialogs. This is common for applications that provide an option for the user to print one copy using default settings. In this case, an application can programmatically set up the page format and print settings, eliminating the need for the user to make settings in either of the printing dialogs. You'll see how to do this in "Printing One Copy" (page 46).

Another situation in which your application could bypass the printing dialogs is to provide support for the user to save a document as a portable document format (PDF) file. The default spool file format in Mac OS X is PDF. As a result, it is straightforward for those applications that support printing in Mac OS X (version 10.1 and later) to also support saving a document as a PDF file. You'll see how to implement this in "Saving a Document as a PDF File" (page 44).

Let's take a look at what an application needs to do to implement high-level printing tasks in each situation: printing driven by printing dialogs and printing that does not require a user to interact with the printing dialogs.

When Printing Dialogs Are Required

If your application lets the user choose Page Setup and Print from the File menu, you need to perform the following to implement each of the high-level printing tasks.

Setting up the page format. The user has the option to choose Page Setup from the File menu and make settings in the Page Setup dialog, but the user is not required to do so. Regardless of whether the user opens the Page Setup dialog, your application must make sure that a document has appropriate page format settings by setting default values.

If the user opens the Page Setup dialog, your application sets the page format values to defaults and then displays the Page Setup dialog. When the users closes the Page Setup dialog by clicking the OK button, your application should save the page format settings so it can retrieve the settings when the user prints the document. Your application should be able to retrieve the page format even if the document is not printed until the next time the user launches the application.

If the user opens the Print dialog, your application needs to check for valid page format settings. If there aren't any, your application sets the page format default values before the Print dialog opens.

Setting up the print settings. Your application needs to set default print settings for the current printer before it opens the Print dialog. When the user clicks Print in the Print dialog your application must invoke the next high-level printing task, which is printing the job.

Printing the job. Your application must determine the number of pages to print, then draw the pages in the range specified by the user.

Chapter 3, "Printing Tasks," (page 25) provides detailed information and sample code that shows how to implement each high-level printing task when your application uses the Page Setup and Print dialogs.

When Printing Dialogs Are Not Required

If your application does not require a user to open either the Page Setup or Print dialogs, it performs the high-level printing tasks sequentially, without interruption. The tasks should be implemented as described here.

Setting up the page format. Your application must make sure that a document has appropriate page format settings.

Setting up print settings. Your application must make sure that a document has appropriate print settings, and that the destination (typically the default printer, but it could be a PDF file or other destination) is set.

Printing the job. Your application must determine the number of pages to print and then draw each page so they are sent to the printer queue or PDF file. Creating a PDF requires that your application set the print destination as a PDF file. This feature is available only in Mac OS X, version 10.1 and later.

In some cases when an application doesn't require printing dialogs, it also doesn't need the printing system to display a printing status dialog. If you want to suppress the printing status dialog, your application needs to use the "No Status Dialog" versions of four of the Carbon Printing functions. You'll see what those functions are and how to call them in "Saving a Document as a PDF File" (page 44).

Printing Objects

The main Carbon Printing Manager objects used to implement the high-level printing tasks are the page format (`PMPageFormat`), print settings (`PMPrintSettings`), and printing session (`PMPrintSession`) objects. The Carbon Printing Manager hides the underlying implementation of these objects, so you can't access the contents directly. Instead, you must use Carbon Printing Manager functions to access the internal data stored in these objects. See "Printing Functions" (page 14) for information on the functions most commonly used with these objects. See the *Carbon Printing Manager Reference* for information on all the printing objects available.

Page Format Object

The page format object (`PMPageFormat`) stores information about how pages of a document should be formatted, such as paper size and orientation. You use the function `PMCreatePageFormat` to allocate a page format object and the function `PMSessionDefaultPageFormat` to assign default values. Optionally, you can also use the functions `PMSetPageFormatExtendedData` and `PMGetPageFormatExtendedData` to store into and retrieve application-specific data from the page format object.

When the user saves a document, your application should flatten the page format object associated with that document and then save the flattened data with the document. When the user opens the document later, your application can unflatten the page format data and restore it as a page format object. This allows users to resume using their style preferences for printing their documents.

Print Settings Object

The print settings object (`PMPrintSettings`) stores information from the Print dialog, such as page range and number of copies. You allocate a print settings object by calling the function `PMCreatePrintSettings` and assign default values by calling the function `PMSessionDefaultPrintSettings`.

Apple recommends that you do not reuse this information if the user prints the document again. The information supplied by the user in the Print dialog should pertain to the document only during a single printing of the document, so there is no need to save the print settings object. The next time the user chooses to print the document, your application should create a new print settings object whose values are set to the defaults.

Printing Session Object

A printing session object (`PMPrintSession`) stores information the printing system uses for a print job. You create a printing session object using the function `PMCreateSession`. A printing session object contains information that's needed by the page format and print settings objects, such as default page format and print settings values. For this reason, some Carbon Printing Manager functions can be called only after you have created a printing session object. For example, setting defaults for or validating page format and print settings objects can only be done after you have created a printing session object.

In Mac OS X, Carbon applications can create more than one printing session object. This means your application can execute more than one print loop at a time. In Mac OS 8 and 9, you are limited to using one printing session object at any given time.

Printing Functions

This section provides an overview of the functions you use to work with the three main printing objects—page format, print settings, and printing session. The *Inside Mac OS X: Carbon Printing Manager Reference* provides a complete reference for the functions available to support printing in your application.

Functions Used With a Page Format Object

Table 2-1 (page 14) shows some of the accessors and other functions available to work with a page format object, which is used to store information displayed in the Page Setup dialog. Applications typically store page format information with a document, and also maintain the information between calls to display the Page Setup dialog, because users expect changes made in the Page Setup dialog to persist with a specific document. The table describes the functions used to create a page format object, set it to default values, validate it against information in the current printing session object, extract information from it, and so on.

Table 1-1 Functions for working with a page format object

Function	Description
`PMCreatePageFormat`	Allocates a new page format object. Increments the reference count to 1.
`PMRelease`	Decrements the reference count for a printing object (such as an instance of page format data type). When an object's reference count reaches 0, the object is deallocated.
`PMSessionDefaultPageFormat`	Assigns default parameter values to an existing page format object. The default values are obtained from the specified printing session object, and specify the page format for the generic ("Any printer") printer.
`PMSessionValidate-PageFormat`	Validates a page format object against the specified printing session object.
`PMFlattenPageFormat`	Flattens a page format object for storage in a user document or other location.

Function	Description
`PMUnflattenPageFormat`	Creates a page format object from a flattened representation produced previously by `PMFlattenPageFormat`.
`PMGetPageFormat-ExtendedData`	Obtains extended page format data for the application.
`PMSetPageFormat-ExtendedData`	Sets extended page format data for the application.
`PMGetAdjustedPageRect`	Obtains the page size, taking into account orientation, application drawing resolution, and scaling settings. Outside of this page rectangle all drawing operations are clipped.
`PMGetUnadjustedPageRect`	Obtains the page size, but does not take into account adjustments for orientation, application drawing resolution, and scaling settings.
`PMGetAdjustedPaperRect`	Obtains the paper size, taking into account orientation, application drawing resolution, and scaling settings. This is the full sheet of paper, including the margins beyond the page rectangle.
`PMGetUnadjustedPaperRect`	Obtains the paper size, but does not take into account orientation, application drawing resolution, and scaling settings.
`PMGetScale`	Obtains the scaling factor currently applied to the page and paper rectangles.
`PMSetScale`	Sets the scaling factor currently applied to the page and paper rectangles.
`PMGetOrientation`	Obtains the current setting for page orientation.
`PMSetOrientation`	Sets the current setting for page orientation.
`PMSetResolution`	Sets the application drawing resolution.

Functions Used With a Print Settings Object

Table 2-2 (page 15) shows some of the accessor functions available to work with a print settings object, which is used to store information displayed in the Print dialog. Applications typically don't store print settings information because users expect the Print dialog to show default values. The table describes functions used to create a print settings object, set it to default values, validate it against information in the current printing session object, extract information from it, and so on.

Table 1-2 Functions for working with a print settings object

Function	Description
`PMCreatePrintSettings`	Allocates a new print settings object. Increments the reference count to 1.

Function	Description
`PMRelease`	Decrements the reference count for a printing object (such as an instance of print settings data type). When an object's reference count reaches 0, the object is deallocated.
`PMSessionDefaultPrintSettings`	Assigns default parameter values to an existing print settings object. The default values are obtained from the specified printing session object.
`PMSessionValidatePrintSettings`	Validates a print settings object against the specified printing session object.
`PMFlattenPrintSettings`	Flattens a print settings object for storage in a user document or other location. (Apple recommends that you do not save print settings.)
`PMUnflattenPrintSettings`	Creates a print settings object from a flattened representation produced previously by `PMFlattenPrintSettings`.
`PMGetPrintSettingsExtendedData`	Obtains extended print settings data for the application.
`PMSetPrintSettingsExtendedData`	Sets extended print settings data for the application.
`PMGetCopies`	Obtains the number of copies that the user has requested to be printed.
`PMSetCopies`	Sets the number of copies that the user has requested to be printed.
`PMSetFirstPage`	Sets the default page number of the first page to be printed, as displayed in the Print dialog.
`PMGetFirstPage`	Obtains the page number entered by the user in the From field of the Print dialog
`PMSetLastPage`	Sets the default page number of the last page to be printed, as displayed in the Print dialog.
`PMGetLastPage`	Obtains the page number entered by the user in the To field of the Print dialog
`PMSetPageRange`	Sets the valid range of pages that can be printed. In Mac OS X, these values appear as the default values in the To and From fields of the Print dialog. In Mac OS 8 and 9, this function has no effect on the values that appear in the To and From fields.
`PMGetPageRange`	Obtains the valid range of pages than can be printed.

Functions Used With a Printing Session Object

Table 2-3 (page 17) shows some of the accessors and other functions available to work with a printing session object, which contains information the printing system uses for a specific print job. An application typically creates a printing session object as needed (such as before displaying the Page Setup or Print dialog) and releases it when it is no longer needed (such as when the Page Setup dialog is dismissed by the user or when the print loop has completed).

Table 1-3 Functions for working with a printing session object

Function	Description
`PMCreateSession`	Allocates a printing session object and initializes with values for the current print job. Increments the reference count to 1.
`PMRelease`	Decrements the reference count for a printing object (such as an instance of a printing session object). When an object's reference count reaches 0, the object is deallocated.
`PMSessionBeginDocument`	Begins a print job.
`PMSessionEndDocument`	Ends the print job that was started with `PMSessionBeginDocument`.
`PMSessionBeginPage`	Informs the printing system that the drawing that follows is part of a new page.
`PMSessionEndPage`	Completes drawing the current page.
`PMSessionGetCurrentPrinter`	Obtains a reference to the printer object (`PMPrinter`) for the current printer (not the formatting printer).
`PMSessionGetDataFromSession`	Obtains the data the application previously stored in a printing session object.
`PMSessionSetDataInSession`	Sets the data the application previously stored in a printing session object.
`PMSessionGetGraphicsContext`	Obtains the graphics context associated with the current page.
`PMSessionPageSetupDialog`	Displays the Page Setup dialog and records the user's selections in a page format object.
`PMSessionPrintDialog`	Displays the Print dialog and records the user's selections in a print settings object.
`PMSessionUseSheets`	Specifies that a printing dialog be displayed as a sheet (that is, attached to the window of the document being printed). Sheets are displayed only in Mac OS X. This function returns the result `kPMNotImplemented` in Mac OS 8 and 9.

The Print Loop

The **print loop** is your application's code that calls all the necessary functions to print a selected page range of a document. It is called a print loop because it loops to print each page in the range. The print loop and the Print dialog must use the same printing session object.

The pseudocode in Listing 2-1 (page 18) shows the calls a typical print loop might make, with the calls divided among Carbon Printing Manager functions, other Carbon functions, and application-defined functions. Not all error handling is shown, but the print loop should check for errors after each call that may return one.

Listing 1-1 Pseudocode for a print loop function

```
AppPagesInDoc (application-defined function to verify a valid page range)
PMGetFirstPage (gets the number of the first page to be printed)
PMGetLastPage (gets the number of the last page to be printed)
(now know how many pages to print in the print loop)
PMSetLastPage (sets the last page in the print job; used for status dialog)
PMSessionBeginDocument (begin a new print job)
    (for each page to be printed)
    PMSessionError (check for an error before starting a new page;
                    if error, break out of loop)
    PMSessionBeginPage (prepare to print the current page)
        PMSessionGetGraphicsContext (get the printing port)
        SetPort (set current drawing port to the printing port)
        AppDrawPage (application-defined function to draw one page)
        SetPort (restore saved port)
    PMSessionEndPage (finish printing the current page)
PMSessionEndDocument (end the print job)
AppPostPrintingErrors (application-defined function to display
                               error message, if any, to the user)
PMRelease (release the print settings and printing session objects)
```

You'll notice that very few of the functions called in the print loop are application-defined functions—most are Carbon Printing Manager functions or other Carbon functions. Your application supplies functions only to determine the maximum number of pages in its document, to draw the pages to be printed, and to display error messages (if any). See "Writing the Print Loop" (page 39) for more information.

Sequence, Scope, and Usage

The Carbon Printing Manager enforces a sequence of steps in the print loop, and defines a valid scope for each printing function. This means that your application must call certain functions before calling others. Functions used out of sequence return the result code `kPMOutOfScope`.

Here are the basic rules for sequence, scope, and usage:

- If you create an object, you must release it. You can pass any Carbon Printing Manager object to the function `PMRelease` to release it.
- A printing session object is created after a successful call to `PMCreateSession` and is released by calling `PMRelease`. Some Carbon Printing Manager functions must only be called between these two calls.

- Any function whose name begins with `PMSession` can only be called between the creation and release of a printing session object.
- Any function whose name includes `Begin` (such as `PMSessionBeginPage`) must be called before the corresponding `End` function (such as `PMSessionEndPage`).
- An `End` function must be called if the corresponding `Begin` function returns `noErr`, even if errors occur within the scope of the `Begin` and `End` functions.
- You can call the functions `PMSessionBeginPage` and `PMSessionEndPage` only within the scope of calls to `PMSessionBeginDocument` and `PMSessionEndDocument`.
- The function `PMSessionGetGraphicsContext` can be called only with the scope of calls to `PMSessionBeginPage` and `PMSessionEndPage`.
- If you want to use sheets, you must call the function `PMSessionUseSheets` before you call the functions `PMSessionPageSetupDialog` or `PMSessionPrintDialog`. (Sheets are available only in Mac OS X.)

Listing 2-2 (page 19) shows a typical calling sequence for code that sets print settings and sends a print job to a printer. The call to `PMRelease` applies to the printing session object created by the function `PMCreateSession`.

In general, functions may be called in any order with respect to other functions at the same or lower scope level (represented in Listing 2-2 by indentation). For example, you can call `PMSessionGetGraphicsContext` only within the scope of a call to `PMSessionBeginPage`, which in turn must be within the scope of a call to `PMSessionBeginDocument`. But within the scope of a call to `PMCreateSession`, you can call `PMSessionDefaultPageFormat` and `PMSessionDefaultPrintSettings` in any order.

Listing 1-2 Typical calling sequence for code that adjusts print settings and then prints a document

```
PMCreateSession
    PMSessionDefaultPrintSettings
    PMSessionValidatePrintSettings
    PMSessionDefaultPageFormat
    PMSessionValidatePageFormat
    PMSessionUseSheets
    PMSessionPrintDialog
    PMSessionBeginDocument
        PMSessionBeginPage
            PMSessionGetGraphicsContext
        PMSessionEndPage
    PMSessionEndDocument
PMRelease
```

The following list shows some of the printing functions that do not need to be called between the creation and release of a printing session object. Many of these functions are used in the sample code in "Printing Tasks" (page 25). Note however, that the functions with an asterisk (*) must be called before a call to the function `PMSessionBeginDocument` or before you display the Print dialog.

```
PMCreatePageFormat*
PMCreatePrintSettings*
PMFlattenPageFormat*
PMUnflattenPageFormat*
PMGetPageFormatExtendedData*
PMSetPageFormatExtendedData*
PMGetUnadjustedPaperRect
PMGetUnadjustedPageRect
PMGetOrientation
```

```
PMSetOrientation*
PMFlattenPrintSettings
PMUnflattenPrintSettings*
PMGetPageRange
PMSetPageRange*
PMGetResolution
PMSetResolution*
```

The functions `PMGetAdjustedPaperRect` and `PMGetAdjustedPageRect` also do not need to be called between the creation and release of a printing session object, but it is currently required that you call `PMSessionValidatePageFormat` before you call either function. Validating a page format object causes the printing system to calculate the adjusted page and paper rectangles.

Note: When a user makes any change that can affect the adjusted rectangles, such as changes to the orientation, scaling, or resolution, the changes are recorded in the page format object but the adjusted rectangles may not be immediately calculated. The adjusted rectangles may only be calculated when you call the function `PMSessionValidatePageFormat`.

You can find out more about each function, including when to use it, in the *Inside Mac OS X: Carbon Printing Manager Reference*.

Page and Paper Rectangles

The page and paper rectangles define the paper sheet size and imageable area for your application's printed page. If you are new to Mac OS X, you should read this section. (Mac OS X uses the same definitions for these rectangles as those used for the old Printing Manager in Mac OS 9.) This section provides the information you need to write a function that draws a single page of a document for your application. It describes the page and paper rectangles and the relationship between them, the adjusted page and paper rectangles, and the margins for the imageable area of the sheet.

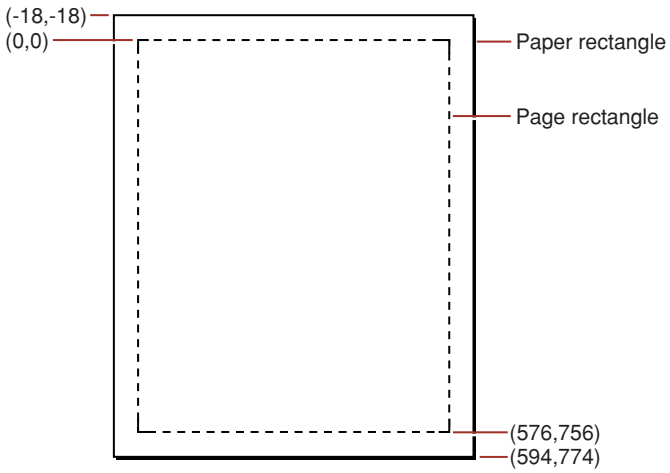
Page Rectangle

The page rectangle is the area of the page to which an application can draw. The coordinates for the upper-left corner of the page rectangle are (0,0), as shown in Figure 2-4 (page 21). The coordinates of the lower-right corner specify the maximum page height and width attainable on the given printer for this page format. Anything drawn outside of the page rectangle is clipped.

Note: The printing system records all drawing commands, even if they draw to the area outside the page rectangle. At print time items outside the page rectangle might be clipped by the printing system.

The width and height of the drawing area are determined by a number of factors—the user's settings for orientation and scaling as well as the resolution your application sets by calling the function `PMSetResolution`. In all cases, the size of the drawing area is limited by the lower-right corner of the page rectangle. Figure 2-4 shows the page rectangle for generic letter-size paper, at 72 dpi, 100% scaling, and portrait orientation.

Figure 1-4 Page and paper rectangles



Your application should always use the page rectangle sizes provided by the printer and should not attempt to change them or add new ones. If your application offers page sizes other than those provided by the printer, you risk compatibility problems.

Paper Rectangle

The paper rectangle gives the paper sheet size, defined relative to the page rectangle, with the same coordinate system. Thus, the upper-left coordinates of the paper rectangle are typically negative, and the lower-right coordinates are greater than those of the page rectangle. Figure 2-4 (page 21) shows the relationship of these two rectangles. The difference between the page and paper rectangles defines the area of the sheet that can't be printed on, sometimes referred to as the **hardware margin**. Assuming a resolution of 72 dpi, the hardware margin in Figure 2-4 is .25" (6.3 mm) in both dimensions.

Adjusted Page and Paper Rectangles

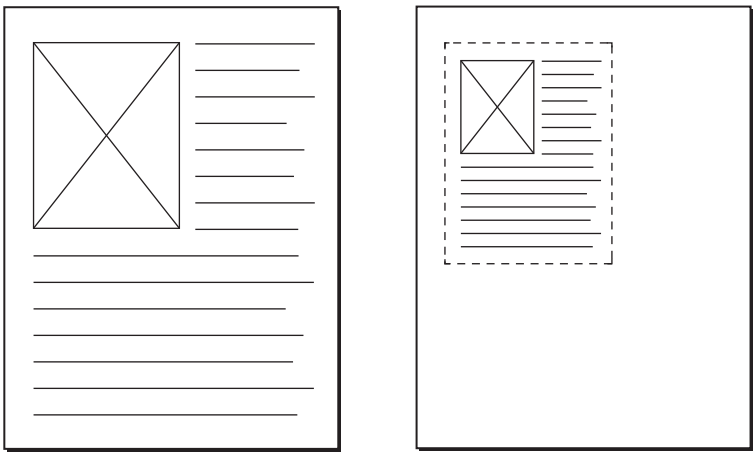
The adjusted page and paper rectangles are more important to your application than the unadjusted ones, as they define the drawing area after orientation, scaling, and resolution are applied. The printing system interprets your application's drawing relative to these coordinates, but the unadjusted rectangles also provide a reference for the application in case information (such as a print preview) needs to be displayed to the user.

By default, all rectangles specify the page and paper sizes in dots-per-inch (dpi) with a default resolution of 72 dpi. If there are no orientation or scaling changes, and the application resolution is 72 dpi, the adjusted and unadjusted page and paper rectangles are equal. When scaling, orientation, and resolution are taken into account, the adjusted rectangles can define a drawing area quite different from the original unadjusted rectangles.

Scaling increases the amount of content your application can draw to a page when the user specifies a percentage less than 100% and decreases the amount of content your application can draw to a page when the percentage is greater than 100%. For example, if the user specifies 50% scaling, the page and paper rectangles increase in size by a factor of 2 in each dimension relative to the content, as shown in Figure 2-5. While it may not be intuitive for the page and paper rectangles to increase in size when scaling is decreased,

increasing the size of the rectangles effectively shrinks the content. The converse is true. If the user specifies 200% scaling, the page and paper rectangles decrease in size by half in each dimension relative to the content, effectively increasing the content size by 200% in each direction.

Figure 1-5 100% scaling compared to 50% scaling



If the user chooses landscape mode, the adjusted page and paper rectangles become, for most paper sizes, wider than they are tall. Because some papers are naturally wider than they are tall (envelopes, for example), it's best not to calculate page orientation based on the difference between the height and width of the page. Instead, use the function `PMGetOrientation`. In most cases your application shouldn't need the orientation, just the dimensions of the adjusted page and paper rectangles.

When your application sets the drawing resolution to a value other than 72 dpi, the adjusted page and paper rectangles change accordingly. For example, setting the resolution to 600 dpi enlarges both rectangles by a factor of 8.3 in both dimensions—a previous page rectangle width of 576 units changes to 4800 units.

Application Margins

The default margins set by your application don't need to coincide with the page rectangle. Applications typically provide default margins within the page rectangle. For example, the margins for an application might be as much as 1.5" (38mm) even though the margins for the page rectangle are much smaller, such as .25" (6.3mm). An application may allow the user to specify application margins that allow content to fall outside the printable area for a given printer.

If You've Used the Old Printing Manager

If you've previously used the old Printing Manager, you'll find that the Mac OS X printing architecture and most of the underlying concepts are slightly different from what you've used in Mac OS 9 and earlier. This section outlines the main differences. If you have developed applications for Mac OS 9, you should read this section to find out how the old concepts relate to the new ones. If you want to convert an existing Mac OS 9 application to use the Carbon Printing Manager, make sure you also read "Adopting the Carbon Printing Manager" (page 49). If you are not modifying an existing application, you can skip this section.

A key aspect of the Mac OS X printing system is its robust support for Carbon applications. Because the Carbon Printing Manager is supported in Mac OS 8 and 9 as well as Mac OS X, a Carbon application is able to print as expected in both environments. For example, when running in Mac OS 8 and 9, the application utilizes the traditional user interface and drivers. In Mac OS X, the application automatically takes advantage of the new printing system's more consistent set of printing dialogs and its flexible printing architecture.

Prior to Mac OS X, printing was supported by the old Printing Manager interface. This interface is not used by Carbon applications. The Carbon Printing Manager interface is defined in the header files `PMApplication.h`, `PMCore.h`, and `PMDefinitions.h`.

If you need to convert existing printing code to use the Carbon Printing Manager, you should be aware of the following changes. You can find more details about converting your code in "Adopting the Carbon Printing Manager" (page 49).

- The old Printing Manager print record (`TPrint`) is replaced by two opaque objects: a print settings object (`PMPrintSettings`) and a page format object (`PMPageFormat`). You create these objects using the `PMCreatePrintSettings` and `PMCreatePageFormat` functions, each of which returns a reference that you can pass to other Carbon Printing Manager functions to obtain information stored in the objects.
- Your application must not make assumptions about the size or content of the print settings and page format objects. Your application can attach extended data to both objects using Carbon Printing Manager functions, but applications must not assume a specific size when storing or retrieving flattened versions of these objects with documents.
- The Carbon Printing Manager provides functions for flattening and restoring the print settings and page format objects. In most cases, your application should store only the flattened page format object. If older versions of your application store a print record (created by the old Printing Manager) with a saved document, you may continue to do so for backward compatibility. However, some information may be lost when converting a page format object into a old print record (or vice versa), due to limitations of the print record structure in the old Printing Manager.
- In Mac OS X, Carbon applications can create multiple printing session objects and run more than one print loop at a time. Each print loop is independent of other print loops. In Mac OS 8 and 9, applications are limited to a single printing session object, and therefore one print loop at a time.
- The Carbon Printing Manager enforces an order in which some functions should be called. Any function used out of order returns the result code `kPMOutOfScope`.
- The style dialog box is now called the Page Setup dialog, and the job dialog box is now called the Print dialog.
- If your application requires customizing the Page Setup or Print dialogs, you should consider creating a printing dialog extension (PDE). If you want your application's custom dialog to be displayed as a sheet in Mac OS X, you must create a printing dialog extension. For more information, see *Extending Printing Dialogs*.
- Low-level driver functions such as `PrLoadDriver` are not supported.

Printing Tasks

Chapter 2, "Printing Concepts for Carbon Developers," (page 9) discussed the high-level tasks required to support printing in a Carbon application: setting up the page format, setting up the print settings, and printing the print job. This chapter shows you how to implement each of those tasks for a document-based Carbon application, such as a text editor or drawing application. This chapter describes the following tasks:

- "Setting Up the Page Format" (page 26). Your application must make sure valid page format settings exist for a document.
- "Setting Up the Print Settings" (page 33). Your application must make sure valid print settings exist for a print job.
- "Printing the Job" (page 38). Your application must determine how many pages are in the print job and draw each page.
- "Handling Errors" (page 43). Errors can occur at many points during the printing process. Your application should provide a function to post error messages, should they occur.
- "Saving a Document as a PDF File" (page 44). Saving a document as a portable document format (PDF) file in Mac OS X uses code similar to the code you need to print a document to a printer. With very little effort on your part, your application can provide users with the ability to save their documents to a file that uses this popular format.
- "Printing One Copy" (page 46). Your application can provide a shortcut command (Print One Copy) to allow users to print a single copy of a document using default settings without requiring them to open the Page Setup and Print dialogs.
- "Printing Multiple Copies" (page 46). The printing system handles printing multiple copies for you.

This chapter contains sample code to illustrate each task you need to do to support printing. See Printing Carbon Sample Code for the sample application on which the code is based.

If you've installed the Mac OS X Developer Tools CD you can find additional sample applications in the following directory that show how to support printing in Mac OS X. You can compile and run the sample applications in Project Builder.

```
/Developer/Examples/Printing/App/
```

As you go through the sample code in this chapter, note that any function, data type, or constant your application must supply has the prefix `My`, except for global constants, which use the prefix `gMy`. Application-defined constants have the prefix `kMy`. Carbon Printing Manager functions use the prefix `PM`. If a line of code contains a commented number, an explanation for that line of code follows the listing.

Note: If your application has very simple printing needs and you are using Multilingual Text Engine (MLTE), you should investigate using the MLTE functions `TXNPageSetup` and `TXNPrint` instead of using Carbon Printing Manager functions. The MLTE functions call through to the Carbon Printing Manager, elminating the need for you to write the code described in this chapter.

Setting Up the Page Format

Most Carbon applications that print allow the user to choose the Page Setup command from the File menu to set options that control the page format for a document. This section shows you how to set up the page format in response to the Page Setup command. It discusses the following tasks that your application needs to perform to set up the page format for a document:

- "Responding to the Page Setup Command" (page 26)
- "Setting Up a Page Format Object" (page 30)
- "Handling Dismissal of the Page Setup Dialog" (page 31)

This section also shows how to flatten page format data so you can save it with a document and unflatten saved data to use it. See "Saving and Retrieving Page Format Data" (page 32) for information on how to perform these optional tasks.

Responding to the Page Setup Command

When the user chooses Page Setup from the File menu your application needs to perform the following operations:

1. Create a printing session object.
2. Make sure there is a valid page format object for the document by performing one of the following actions:

   If no page format object exists for the document, your application must create one and set it to default values for this session. (The page format object is not usable until it has default values.)

   If the page format object already exists, validate the object against the printing session object. Validation updates any fields in the page format object that need to be calculated, such as the adjusted page and paper rectangles.

   If the page format object doesn't exist, but the document has page format data saved with it, your application needs to unflatten the saved data to create a page format object that contains the saved settings.
3. Call the function `PMSessionUseSheets` to indicate that the Page Setup dialog should use sheets. Although sheets are not available in Mac OS 9 and earlier, you should call this function if your application runs in Mac OS X as well as Mac OS 9. In Mac OS 9, the function returns a result code that indicates sheets are not available.

4. Call `PMSessionPageSetupDialog` to display the Page Setup dialog so the user can specify settings such as paper size and orientation. In Mac OS X, the dialog is displayed as a sheet as long as you called the function `PMSessionUseSheets`.

5. Preserve page setup information. Users expect page setup information to persist with the document. Your application should save the page format data with the document.

6. Release the printing session object and handle errors, if any occur.

One of the major design decisions you must make for your application is how to handle the page format object for the document. The page format should persist with the document after the user has dismissed the Page Setup dialog. In other words, the next time the user opens the Page Setup dialog for the document, the settings should be the same as they were when the user last closed the Page Setup dialog for that document. The settings should persist when the user quits the application, launches it again, and then opens the document.

There are a variety of ways you can handle page format data for a document. The sample code in this chapter stores the page format object in a structure and attaches it to a document window as a property. The structure is simple; it contains only two items. Most document-based applications use a larger structure to store all information about the document, including printing-related information. You need to handle the page format data in a way that's best for your application.

The sample code assumes the following structure is defined:

```
typedef struct MyDocData
{
    PMPageFormat            pageFormat;
    PMPrintSettings         printSettings;
}MyDocData;
```

The structure `MyDocData` contains two fields, one of type `PMPageFormat` and another of type `PMPrintSettings`. Although it is important for the page format data to persist with the document, it is not recommended that print settings data persist. However, as you'll see in "Setting Up the Print Settings" (page 33), using the structure `MyDocData` provides a convenient way to pass the print settings object from one function to another; the print settings are not saved with the document after the print job is cancelled or sent to a print queue.

The function `MyDoPageSetup` in Listing 3-1 (page 27) shows how your application can respond to the Page Setup command. Following this listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-1 A function that responds to the Page Setup command

```
OSStatus MyDoPageSetup (WindowRef documentWindow,
                        MyDocData *docDataP)                    // 1
{
    OSStatus err = noErr;

    if (docDataP)                                               // 2
    {
        PMPrintSession printSession;
        PMPageFormat pageFormat = NULL;
        err = PMCreateSession (&printSession);                  // 3
        if (!err)
         {
```

```
            Boolean accepted;
            err = MySetupPageFormatForPrinting (printSession,
                                        docDataP, &pageFormat);          // 4
            if (!err)
             {
                Boolean sheetsAreAvailable = true;                       // 5
                err = PMSessionUseSheets (printSession, documentWindow,
                                        gMyPageSetupDoneProc);           // 6
                if (err == kPMNotImplemented)                            // 7
                {
                        err = noErr;
                        sheetsAreAvailable = false;
                }
                if (!err)
                {
                     err = PMSessionPageSetupDialog (printSession,
                                pageFormat, &accepted);                  // 8
                    if (err == noErr && !sheetsAreAvailable)
                    MyPageSetupDoneProc (printSession,
                                        window, accepted);               // 9
                }
            }
            if (err)
                (void) PMRelease (printSession);                         // 10
        }
    }
    MyPostPrintingError (err, kMyPrintErrorFormatStrKey);                // 11
    return err;
}
```

Here's what the code in Listing 3-1 (page 27) does:

1. Passes a reference to the document window and a pointer to the data structure that contains the page format object (`PMPageFormat`) for the document. Your application can get the pointer (`docDataP`) to pass to your `MyDoPageSetup` function by calling the Window Manager function `GetWindowProperty`. This assumes that you have already set up the data structure and set it as a property of the window using the Window Manager function `SetWindowProperty`.

2. Makes sure the pointer is valid.

3. Calls the Carbon Printing Manager function `PMCreateSession` to create a printing session object that is used for the page format code that follows.

4. Calls your application's function to set up a page format object. See "Setting Up a Page Format Object" (page 30). Pass the printing session, the pointer to the structure `MyDocData`, and a pointer to the local page format object.

5. Sheets are not available in Mac OS 8 and 9. If you plan to run your application in Mac OS 8 and 9 as well as in Mac OS X, you need to write your code so it acts properly in both situations. This code demonstrates how you can support each operating system. First, create a variable `sheetsAreAvailable` and set it to `true`.

6. Calls the Carbon Printing Manager function `PMSessionUseSheets` to specify that a printing dialog (in this case the Page Setup dialog) should be displayed as a sheet.

You need to pass the current printing session, the window that contains the document to be printed, and a pointer to your page setup done function. When using sheets in Mac OS X, the Carbon Printing Manager calls your function when the user dismisses the Page Setup dialog. This code assumes the application already declared a global variable:

```
gMyPageSetupDoneProc = NewPMSheetDoneUPP (MyPageSetupDoneProc);
```

If your application runs in Mac OS 8 or 9, calling the function `PMSessionUseSheets` returns the error `kPMNotImplemented`, and the function has no effect.

7. Checks for the error `kPMNotImplemented`. If it is returned, that means your application is running in Mac OS 8 or 9, and that you are responsible for calling your procedure to handle dismissal of the Page Setup dialog. Set the constant `sheetsAreAvailable` to `false`.

8. Calls the Carbon Printing Manager function `PMSessionPageSetupDialog` to display the Page Setup dialog. Pass the current printing session, the page format object that was set up in a previous step, and a pointer to a Boolean value. You call this function regardless of the version of the operating system. If your application is running in Mac OS X, the dialog appears as a sheet.

   The following is true if you are using sheets:

   - When the user dismisses the Page Setup dialog, the Carbon Printing Manager calls the function specified by the constant `gMyPageSetupDoneUPP` in the previous call to `PMSessionUseSheets`.
   - When using sheets, the `PMSessionPageSetupDialog` function returns immediately and the Boolean value returned in the `accepted` parameter is irrelevant because it is your Page Setup dialog done function that is called when the dialog is dismissed. If your application needs to perform additional tasks after the user dismisses the Page Setup dialog, it can do so in the `MyPageSetupDoneProc` function, which is called when the user dismisses the Page Setup dialog.
   - If the user clicks the OK button in the Page Setup dialog, the page format object is updated with the user's changes (if any) and the value `true` is returned to the `MyPageSetupDoneProc` function. If the user clicks the Cancel button, the page format object is unchanged and the value `false` is returned to the `MyPageSetupDoneProc` function.

   The following is true if you are not using sheets:

   - The `PMSessionPageSetupDialog` function does not return until the user dismisses the Page Setup dialog.
   - The Boolean value returned in the `accepted` variable is `true` if the user clicks OK and `false` if the user clicks Cancel.

9. If there is no error, and sheets are not available, then your application must call its Page Setup dialog done function (`MyPageSetupDoneProc`) to process the results of the Page Setup dialog and do any necessary clean up.

10. If an error is returned from the Page Setup dialog or prior to showing the dialog, then you must release the printing session here. Otherwise you should release the printing session in your Page Setup dialog done function (`MyPageSetupDoneProc`).

11. Calls your application's function to display an error message. See "Handling Errors" (page 43) for more information. If there is no error, the function does nothing.

Setting Up a Page Format Object

The function `MySetupPageFormatForPrinting`, shown in Listing 3-2 (page 30), makes sure there is a valid page format object for a document. The advantage to creating a separate function to take care of the page format object is that your application can call the function when it handles the Page Setup command and when it handles the Print command. "Responding to the Print Command" (page 33) describes how to handle the Print command.

Listing 2-2 A function that sets up a page format object

```
static OSStatus MySetupPageFormatForPrinting (
                                PMPrintSession printSession,
                                MyDocData *docDataP,
                                PMPageFormat *pageFormatP)                  // 1
{
    OSStatus status = noErr;
    PMPageFormat pageFormat = docDataP->pageFormat;                         // 2
    if (pageFormat == NULL)                                                 // 3
    {
        status = PMCreatePageFormat (&pageFormat);
        if (status == noErr)
        {
            status = PMSessionDefaultPageFormat (printSession,
                                                 pageFormat);
            if (status == noErr)
                docDataP->pageFormat = pageFormat;
            else
             {
                (void) PMRelease (pageFormat);
                pageFormat = NULL;
            }
        }
    }
    else                                                                    // 4
    {
        status = PMSessionValidatePageFormat (printSession, pageFormat,
                                        kPMDontWantBoolean);
        if (status)
        {
            docDataP->pageFormat = NULL;
            (void) PMRelease (pageFormat);
            pageFormat = NULL;
        }
    }

    *pageFormatP = pageFormat;                                              // 5
    return status;
}
```

Here's what the code does:

1. Passes the current printing session object (created either in `MyDoPageSetup` or `MyDoPrint`), a pointer to the `MyDocData` structure, and a pointer to a page format object (created either in `MyDoPageSetup` or `MyDoPrint`).

2. Declares a local variable to hold a page format object and set its value to the document's page format.

3. If the local page format is `NULL`, calls the Carbon Printing Manager function `PMCreatePageFormat` to allocate the page format object, and then do the following:
    - Sets the newly allocated page format object to default values by calling the Carbon Printing Manager function `PMSessionDefaultPageFormat`.
    - If setting up defaults for the local page format object is successful, then sets the document's page format object to the local page format object. If it isn't successful, you need to release the local page format object and set it to `NULL`.
4. If the local page format is not `NULL`, then validates the local page format object within the context of the current printing session. Validating updates any values in the page format object that need to be calculated, such as the adjusted page and paper rectangles.

    If the validation does not succeed, you need to set the document's page format object to `NULL`, release the local page format object, and set the local page format object to `NULL`.
5. If the code is successful so far, you need to assign the local page format object to the storage (`pageFormatP`) you passed to your `MySetupPageFormatForPrinting` function.

Handling Dismissal of the Page Setup Dialog

You need to provide a function to handle dismissal of the Page Setup dialog. If your application uses sheets, the Carbon Printing Manager calls this function when the user dismisses the Page Setup dialog. Otherwise, your application needs to call this function.

At a minimum this function should release the current printing session object, which shouldn't be saved after the user dismisses the Page Setup dialog. If your application has more complicated printing needs, it may need to include code to perform other operations here. For example, your application may need to reformat pages to reflect changes the user made to scaling or paper size options in the Page Setup dialog.

The function `MyPageSetupDoneProc` in Listing 3-3 (page 31) shows how you can handle dismissal of the Page Setup dialog in your application. The function takes three parameters: the current printing session object, a reference to the document window, and a Boolean to indicate whether the user accepted or cancelled the Page Setup dialog. The function does two things: releases the printing session object and calls your application's function to post a printing error, should one occur. See "Handling Errors" (page 43) for information on the error-posting function.

Listing 2-3 A function to handle dismissal of the Page Setup dialog

```
static pascal void MyPageSetupDoneProc (PMPrintSession printSession,
                        WindowRef documentWindow,
                        Boolean accepted)
{
    #pragma unused (documentWindow, accepted)

    OSStatus err = PMRelease (printSession);
    if (err)
        MyPostPrintingError (err, kMyPrintErrorFormatStrKey);
    return;
}
```

Saving and Retrieving Page Format Data

When the user saves a document, most applications save the page format data with it, which consists of choices made by the user in the Page Setup dialog. Because a page format object is an opaque data type, you need to flatten the object before you can save it. When you want to use the data, you need to unflatten it to restore the original page format object.

Listing 3-4 (page 32) shows an example of how you can flatten a page format object so it can be saved with the document. The `MyFlattenAndSavePageFormat` function assumes the caller passes a validated page format object. The function `PMFlattenPageFormat` flattens a page format object to a handle before the application writes it to a document or other location.

Listing 2-4 Saving page format data

```
OSStatus MyFlattenAndSavePageFormat (PMPageFormat pageFormat)
{
    OSStatus    status = noErr;
    Handle      flatFormatHandle = NULL;

    if (pageFormat != kPMNoPageFormat)
    {
         status = PMFlattenPageFormat (pageFormat, &flatFormatHandle);
         if (status == noErr)
            //  In this sample code we simply put it in a global variable.
            //  Replace this line with your code to write the data to a file.
            gflatPageFormat = flatFormatHandle;
    }
    return status;
}
```

Listing 3-5 (page 32) shows a function—`MyLoadAndUnflattenPageFormat`—that gets flattened page format data and returns a page format object. The function `PMUnFlattenPageFormat` converts the flattened data to a page format object.

Listing 2-5 Retrieving page format data

```
OSStatus    MyLoadAndUnflattenPageFormat (PMPageFormat* pageFormat)
{
    OSStatus    status;
    Handle  flatFormatHandle = NULL;

    //  This sample code copies flattened data from a global.
    //  Replace this line with your code to obtain the flattened data
    //  from your document.
    flatFormatHandle = gflatPageFormat;
    status = PMUnflattenPageFormat (flatFormatHandle, pageFormat);

    return status;
}
```

Setting Up the Print Settings

Most Carbon applications that support printing allow the user to choose Print from the File menu to set options that control how a document is printed. This section shows you how to respond to the Print command issued when the user chooses Print from the File menu. It discusses the following tasks that your application needs to do to set up print settings for a document:

- "Responding to the Print Command" (page 33)
- "Handling Dismissal of the Print Dialog" (page 37)

The sample code assumes the following structure is defined:

```
typedef struct MyDocData
{
    PMPageFormat            pageFormat;
    PMPrintSettings         printSettings;
}MyDocData;
```

See "Setting Up the Page Format" (page 26) for more information about this structure.

Responding to the Print Command

When the user chooses Print from the File menu, your application needs to do the following:

1. Create a printing session object.
2. Check for a valid page format object; if there isn't one, create it.
3. Create a print settings object and set it to default values for this session. A print settings object (`PMPrintSettings`) is an opaque object that stores information such as the number of copies and range of pages. An application creates an instance of this object by calling the function `PMCreatePrintSettings`. Apple recommends that you do not save the print settings object with the document, as it is intended to describe print settings for a specific printing session.
4. Call the function `PMSessionUseSheets` to indicate that the Print dialog should use sheets. Although sheets are not available in Mac OS 9 and earlier, you should call this function if your application runs in Mac OS X as well as Mac OS 9. In Mac OS 9, the function returns a result code that indicates sheets are not available.
5. Call the Carbon Printing Manager function `PMSessionPrintDialog` to display the Print dialog so the user can specify settings such as page range and number of copies before printing. In Mac OS X, the dialog is displayed as a sheet as long as you called the function `PMSessionUseSheets`.
6. Release the printing session object, set the print settings object to `NULL`, and handle errors, if any occur. Apple recommends that your application does not save print settings from one session to the next, which is why you need to set the print settings object to `NULL`.

The function `MyDoPrint` in Listing 3-6 (page 34) shows how your application can respond to the Print command. Following this listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-6 A function that responds to the Print command

```
OSStatus MyDoPrint (WindowRef documentWindow,
                            MyDocData *docDataP)                          // 1
{
    OSStatus status = noErr;                                              // 2
    PMPrintSettings printSettings = NULL;
    PMPageFormat pageFormat = NULL;
    UInt32 minPage = 1, maxPage;
    PMPrintSession printSession;

    status = PMCreateSession (&printSession);                             // 3
    if (status == noErr)
     {
        status = MySetupPageFormatForPrinting (printSession,
                                    docDataP, &pageFormat);               // 4
        if (status == noErr)
        {
            status = PMCreatePrintSettings (&printSettings);              // 5
            if (status == noErr)
            {
                status = PMSessionDefaultPrintSettings (printSession,
                                    printSettings);                       // 6
                if (status == noErr)
                {
                    CFStringRef windowTitleRef;
                    status = CopyWindowTitleAsCFString (documentWindow,
                                &windowTitleRef);                         // 7
                    if (status == noErr)
                    {
                        status = PMSetJobNameCFString (printSettings,
                                                windowTitleRef);          // 8
                        CFRelease (windowTitleRef);                       // 9
                    }
                }
            }
            if (status == noErr)
            {
                maxPage = MyGetDocumentNumPagesInDoc (docDataP);          // 10
                status = PMSetPageRange (printSettings,                   // 11
                                    minPage, maxPage);
            }
            if (status == noErr)
            {
                 Boolean accepted;
                 Boolean sheetsAreAvailable = true;                       // 12
                 docDataP->printSettings = printSettings;                 // 13
                 status = PMSessionUseSheets (printSession,
                                    documentWindow,
                                    gMyPrintDialogDoneProc);              // 14

                if (status == kPMNotImplemented)                          // 15
                {
                    status = noErr;
                    sheetsAreAvailable = false;
                }
                if (status == noErr)
```

```
                {
                    status = PMSessionPrintDialog (printSession,
                                                   printSettings,
                                                   pageFormat,
                                                   &accepted);                  // 16
                     if (status == noErr && !sheetsAreAvailable)                // 17
                                    MyPrintDialogDoneProc (printSession,
                                                   parentWindow, accepted);
                }
            }
        }
        if (status != noErr)                                                    // 18
         {
            if (printSettings)
             {
                docDataP->printSettings = NULL;                                 // 19
                (void) PMRelease (printSettings);
            }
            (void) PMRelease (printSession);                                    // 20
         }
    }
    MyPostPrintingError (status, kMyPrintErrorFormatStrKey);                    // 21
    return status;
}
```

Here's what the code in Listing 3-6 (page 34) does:

1. Passes a reference to the document window and a pointer to the data structure that contains the page format and print settings structures for the document. Your application can get the pointer (`docDataP`) to pass to your `MyDoPrint` function by calling the Window Manager function `GetWindowProperty`. This assumes that you have already set up the data structure and set it as a property of the window using the Window Manager function `SetWindowProperty`.

2. Sets up the local variables you need for this function. You need to declare local variable to hold print settings and page format objects and set them to `NULL`. You need two variables—`minPage` and `maxPage`—for getting and setting the page range. You need to declare a variable to hold a printing session object.

3. Calls the Carbon Printing Manager function `PMCreateSession` to create a printing session object.

4. Calls your application's function to set up a page format object. See "Setting Up a Page Format Object" (page 30). Pass the printing session, the pointer to the structure `MyDocData`, and a pointer to storage for the local page format object.

5. Calls the Carbon Printing Manager function `PMCreatePrintSettings` to allocate a local print settings object.

6. Sets default values for the print settings object for the current printing session.

7. Calls the Window Manager function `CopyWindowTitleAsCFString`. This example uses the document's window title as the name of the print job.

8. Calls the Carbon Printing Manager function `PMSetJobNameCFString` to set the name of the print job.

9. Makes sure you call the Core Foundation Base Services function `CFRelease` to release the `CFStringRef` you created for the document's window title.

10. Calls your application's function to determine the number of pages in the document. See "Calculating the Maximum Number of Pages to Print" (page 42).

11. Calls the Carbon Printing Manager function `PMSetPageRange` to specify the actual range of pages in the document. In Mac OS X, the minimum allowable page (`minPage`) appears in the From field in the Copies & Pages pane of the Print dialog and the maximum allowable page (`maxPage`) appears in the To field. If the user enters a value outside of this range in the Print dialog the Carbon Printing Manager displays an alert message. The page range cannot be enforced automatically in Mac OS 8 and 9.

12. Sheets are not available in Mac OS 8 and 9. If you plan to run your application in Mac OS 8 and 9 as well as in Mac OS X, you need to write your code so it acts properly in both situations. This code demonstrates how you can support each operating system. First, create a variable `sheetsAreAvailable` and set it to `true`.

13. Writes the print settings object to the document's data structure to allow the object to be accessed by your application's Print dialog done function.

14. Calls the Carbon Printing Manager function `PMSessionUseSheets` to specify that a printing dialog (in this case the Print dialog) should be displayed as a sheet.

    You need to pass the current printing session, the window that contains the document to be printed, and a pointer to your Print dialog done function. The Carbon Printing Manager calls your Print dialog done function when the user dismisses the Print dialog. This code assumes the application already declared a global variable:

    ```
    gMyPrintDoneProc = NewPMSheetDoneUPP (MyPrintDoneProc);
    ```

    If your application runs in Mac OS 8 or 9, calling the function `PMSessionUseSheets` returns the error `kPMNotImplemented`, and the function has no effect.

15. Checks for the error `kPMNotImplemented`. If it is returned, this means your application is running in Mac OS 8 or 9, and that you are responsible for calling your procedure to handle dismissal of the Print dialog. Set `sheetsAreAvailable` to `false`.

16. Calls the Carbon Printing Manager function `PMSessionPrintDialog`, to display the Print dialog. Pass the current printing session, the print settings and page format objects that were set up in previous steps, and a pointer to a Boolean value. You call this function regardless of the version of the operating system. If your application is running in Mac OS X, the dialog appears as a sheet.

    The following is true if you are using sheets:

    - When the user dismisses the Print dialog, the Carbon Printing Manager calls the function specified by `gMyPrintDialogDoneProc` in the previous call to `PMSessionUseSheets`.
    - When using sheets, the `PMSessionPrintDialog` function returns immediately and the Boolean value returned in the `accepted` variable is irrelevant since it is your Print dialog done function that is called when the dialog is dismissed. If your application needs to perform additional tasks after the user dismisses the Print dialog, it can do so in the `MyPrintDoneProc` function, which is called when the user dismisses the Print dialog.
    - If the user clicks the OK button in the Print dialog, the print settings object is updated with the user's changes (if any) and the value `true` is returned to the `MyPrintDoneProc` function. If the user clicks the Cancel button, the print settings object is unchanged and the value `false` is returned to the `MyPrintDoneProc` function.

    The following is true if you are not using sheets:

- The `PMSessionPrintDialog` function does not return until the user dismisses the Print dialog.
- The Boolean value returned in the `accepted` variable is `true` if the user clicks OK and `false` if the user clicks Cancel.

17. If there is no error, and sheets are not available, then your application must call its Print dialog done function to handle dismissal of the Print dialog. See "Handling Dismissal of the Print Dialog" (page 37).
18. If an error is returned from the Print dialog, then you must release the printing session and print settings objects here. Otherwise you should release them in your Print dialog done function.
19. If there is an error, sets the print settings object stored in the document's data structure to `NULL`.
20. If there is an error, releases the local printing session object.
21. Calls your application's function to post a printing error. See "Handling Errors" (page 43) for more information. If there is no error, the function does nothing.

Handling Dismissal of the Print Dialog

There are a number of operations your application needs to do when the user dismisses the Print dialog. You should handle these in a Print dialog done function. This function is called by the Carbon Printing Manager if your application uses sheets. Otherwise, your application must call this function after it calls the function `PMSessionPrintDialog`.

The function `MyPrintDialogDoneProc` in Listing 3-7 (page 37) shows how you can handle dismissal of the Print dialog in your application. It contains the minimum amount of code necessary to handle the dismissal, including calling the application's print loop if the user accepts the Print dialog. If your application has more complicated printing needs, it may need to include code to perform other operations here. A detailed explanation for each line of code that has a numbered comment appears following the listing.

Listing 2-7 A function to handle dismissal of the Print dialog

```
static pascal void MyPrintDialogDoneProc (PMPrintSession printSession,
                                  WindowRef documentWindow,
                                 Boolean accepted)                          // 1
{
    OSStatus status = noErr, tempErr;
    MyDocData *docDataP = MyGetWindowProperty (documentWindow);             // 2
    if (docDataP)
    {
        if (accepted)                                                       // 3
            status = MyDoPrintLoop (printSession,
                                    docDataP->pageFormat,
                                    docDataP->printSettings,
                                    docDataP);
        tempErr = PMRelease (ourDataP->printSettings);                      // 4
        if (status == noErr)
            status = tempErr;
         docDataP->printSettings = NULL;                                    // 5
    }
    tempErr = PMRelease (printSession);                                     // 6
    if (status == noErr)
```

```
        status = tempErr;
    MyPostPrintingError (status, kMyPrintErrorFormatStrKey);                    // 7
}
```

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

1. The function takes three parameters: the current printing session object, a reference to document window, and a Boolean to indicate whether the user accepted or cancelled the Print dialog.
2. Calls your application's function to retrieve a pointer to the structure `MyDocData`. If you set the pointer as a property of the document window using the Window Manager function `SetWindowProperty`, you can retrieve it using the Window Manager function `GetWindowProperty`. See the Window Manager documentation for more information.
3. If the user accepts the Print dialog (`accepted` has the value `true`), calls your application's `MyDoPrintLoop` function to print the pages in the selected range. See "Writing the Print Loop" (page 39).
4. Calls the Carbon Printing Manager function `PMRelease` to release the print settings object. Releasing an object decrements its reference count, causing it to be deallocated when the count reaches 0. By not saving print settings between calls to the Print dialog, you ensure that the dialog displays with the appropriate default settings, which is the recommended behavior.
5. Sets the value of the print settings object to `NULL` to indicate it is no longer valid.
6. Calls the Carbon Printing Manager function `PMRelease` to release the printing session object.
7. Calls your application's function to post a printing error. See "Handling Errors" (page 43) for information on the error-posting function.

Printing the Job

This section shows you how to write the code that actually creates a print job. Printing can occur only after valid page format and print settings objects are set up. In a document-based application, the printing code is typically called when the user clicks Print in the Print dialog. An application usually calls the printing code from its print dialog done procedure. See the call to the application-defined function `MyDoPrintLoop` from the function `MyPrintDialogDoneProc` (in "Handling Dismissal of the Print Dialog" (page 37)).

This section discusses the following tasks that your application needs to do to print a document:

- "Writing the Print Loop" (page 39)
- "Calculating the Maximum Number of Pages to Print" (page 42)
- "Drawing a Page" (page 42)

Writing the Print Loop

An application's print loop code does most of its work by calling Carbon Printing Manager functions. The code loops over the page range specified by the user. Each pass through the loop, your application needs to call its page drawing function to draw one page. At each step through the print loop, your code should check for errors and take appropriate action if an error occurs.

The function `MyDoPrintLoop` in Listing 3-8 (page 39) shows how your application can implement the print loop. You should be able to adapt this print loop code for applications with more sophisticated printing requirements. Following this listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-8 A function that implements a print loop

```
static OSStatus MyDoPrintLoop (PMPrintSession printSession,
                    PMPageFormat pageFormat,
                    PMPrintSettings printSettings,
                    const MyDocData *docDataP)                              // 1
{
    OSStatus err = noErr;                                                   // 2
    OSStatus tempErr = noErr;
    UInt32 firstPage, lastPage,
            totalDocPages = MyGetDocumentNumPagesInDoc (docDataP);          // 3
    if (!err)
        err = PMGetFirstPage (printSettings, &firstPage);                   // 4
    if (!err)
                err = PMGetLastPage (printSettings, &lastPage);             // 5
    if (!err && lastPage > totalDocPages)                                   // 6
             lastPage = totalDocPages;
    if (!err)
             err = PMSetLastPage (printSettings, lastPage, false);          // 7
    if (!err)
        {
         err = PMSessionBeginDocument (printSession, printSettings,
                                pageFormat);                                // 8
        if (!err)
         {
            UInt32 pageNumber = firstPage;
             while (pageNumber <= lastPage && err == noErr &&
                            PMSessionError (printSession) == noErr)         // 9
            {
                err = PMSessionBeginPage (printSession,
                                    pageFormat, NULL);                      // 10
                if (!err)
                 {
                    GrafPtr oldPort = NULL;
                    void *printingContext = NULL;
                    GetPort (&oldPort);                                     // 11

                    err = PMSessionGetGraphicsContext (printSession,
                            kPMGraphicsContextQuickdraw,
                            (void **) &printingContext);                    // 12
                     if (!err)
                    {
                        Rect pageRect;
```

```
                    SetPort ((CGrafPtr) printingContext);                   // 13
                    GetPortBounds (printingContext, &pageRect);             // 14
                    err = MyPageDrawProc (docDataP, &pageRect,
                                          pageNumber)                       // 15
                    SetPort (oldPort);                                      // 16
                }
                tempErr = PMSessionEndPage (printSession);                  // 17
                if(!err)err = tempErr;
            }
            pageNumber++;                                                   // 18
        } // end while loop
        tempErr = PMSessionEndDocument (printSession);                      // 19
        if (!err)
                err = tempErr;
        if (!err)
                err = PMSessionError (printSession);                        // 20
        }
    }
    return err;
}
```

Here's what the code in Listing 3-8 (page 39) does:

1. The function takes four parameters: the current printing session object, a page format object, a print settings object, and pointer to the structure `MyDocData` (described in "Setting Up the Page Format" (page 26).

2. Declares two variable to keep track of error codes. This ensures an error that could occur in one part of the print loop won't overwrite an error that occurs in another part.

3. Declares variables for the page numbers of the first and last pages to be printed and the number of pages that it is possible to print. Call your application's function (`MyDetermineNumberOfPagesInDoc`) to figure out the maximum number of pages that can be printed. See "Calculating the Maximum Number of Pages to Print" (page 42) for more information.

4. Calls the Carbon Printing Manager function `PMGetFirstPage` to obtain the page number entered by the user in the From field in the Copies & Pages pane of the Print dialog. If the user does not enter a value, the function returns the value of the previous call to `PMSetFirstPage`, if any, or the default value.

   Note: You must use 32-bit unsigned containers to obtain the first page and last page values because in some cases the `PMGetFirstPage` and `PMGetLastPage` functions may return very large values from the print settings data structure.

   You should not use the constant `kPrintAllPages` in your print loop. That constant is used only with the `PMSetLastPage` and `PMSetPageRange` functions to specify a last page. It is not returned by the `PMGetLastPage` function and your code should not look for it here.

5. Calls the Carbon Printing Manager function `PMGetLastPage` to obtain the page number entered by the user in the To field in the Copies & Pages pane of the Print dialog. If the user did not enter a value, the function returns the value of the previous call to `PMSetLastPage`, if any, or the default value.

6. If the user specified a last page number greater than the number of pages in the document, assigns the actual last page in the document to the variable `lastPage`.

7. Calls the Carbon Printing Manager function `PMSetLastPage` to set the last page of the page range in the print settings object for this print job. Setting the last page provides information used by the progress dialog that is shown during printing.

8. Calls the Carbon Printing Manager function `PMSessionBeginDocument` to establish a new print job.

   Note: If no error results from this call, the ensuing code always calls `PMSessionEndDocument` to end the print job regardless of any intervening errors.

9. Sets up a `while` loop over the range of pages selected for printing. Note that the loop terminates if any function returns an error (that is, if the variable `status` has a value other than `noErr`) or if the Carbon Printing Manager function `PMSessionError` returns an error.

10. Calls the Carbon Printing Manager function `PMSessionBeginPage` to inform the printing system that the drawing code which follows is part of a new page.

    Note: If no error results from this call, the ensuing code always calls `PMSessionEndPage` to finish the current page regardless of any intervening errors.

11. Calls the QuickDraw function `GetPort` to preserve the current graphics port.

12. Calls the Carbon Printing Manager function `PMSessionGetGraphicsContext` to obtain the QuickDraw graphics printing port for the page being printed.

13. Calls the QuickDraw function `SetPort` to set the graphics port to the port obtained in the previous step. You must do this before calling the document's function to draw one page.

14. Calls the QuickDraw function `GetPortBounds` to get the page rectangle for the current printing context. Your application may prefer to use the functions `PMGetAdjustedPageRect` and `PMGetAdjustedPaperRect` for its drawing.

15. Calls your application's function to draw the current page. See "Drawing a Page" (page 42) for more information.

16. Calls the `SetPort` function again to restore the port to the one you preserved previously.

17. Calls the Carbon Printing Manager function `PMSessionEndPage` to end the current page. You should use a temporary variable (`tempErr`) to get the status for this function so it doesn't overwrite an existing, prior error. This approach ensures that the current page is always finished and that if any error occurs the loop terminates.

18. Increments the page count within the page-printing loop.

19. On completion of the page-printing loop, call the Carbon Printing Manager function `PMSessionEndDocument` to signal the completion of the print job.

20. Calls the Carbon Printing Manager function `PMSessionError` to determine if any printing error has occurred. If the user cancels the print job, the result code is `kPMCancel`.

Calculating the Maximum Number of Pages to Print

The number of pages in a document is likely to depend on a number of factors, including document-specific changes by the user (such as adding text or changing font size), as well as Page Setup and Print dialog settings. Some applications might require a separate version of this function for each kind of document they print. Typically, you'd call the page calculation function from your print function (`MyDoPrint`). See Listing 3-6 (page 34).

When you write your page calculation function, you should consider including code to do the following:

1. Call the Carbon Printing Manager function `PMGetAdjustedPaperRect` to obtain the paper size, taking into account orientation, application drawing resolution, and scaling settings. Applications can use this information to determine the number of pages in the document based on the current page format.

   If your application formats pages based on the page rectangle, you should instead call the Carbon Printing Manager function `PMGetAdjustedPageRect`.

2. Call the Carbon Printing Manager function `PMGetAdjustedPageRect` to obtain the page size (the imageable area), in points, taking into account orientation, application drawing resolution, and scaling settings.

3. Return the computed number of pages to print in the document.

Drawing a Page

The code you write to draw a page of a document needs to be tailored to support the specific needs of your application. You can call the Carbon Printing Manager function `PMGetAdjustedPageRect` to obtain the page size (the imageable area), in points, taking into account orientation, application drawing resolution, and scaling settings.

Regardless of how you implement page drawing, your application should check for errors after attempting to draw each page, and take the appropriate action should an error occur. Typically, you'd call your page drawing function from your print loop function. See Listing 3-8 (page 39).

Listing 3-9 (page 42) shows a function that does some very simple text drawing.

Listing 2-9 A function that draws one page of a document

```
OSStatus MyPageDrawProc (const MyDocData *docDataP,
                         const Rect *drawingRectP,
                         UInt32 pageNumber)
{
    #pragma unused (docDataP, drawingRectP)
    OSStatus err = noErr;
    Str255 pageNumberString;

    MoveTo (72,72);
    TextFont (kFontIDHelvetica);
    TextSize (24);
    DrawString ("\pDrawing Page Number ");
    NumToString (pageNumber, pageNumberString);
    DrawString (pageNumberString);
```

```
    return err;
}
```

Handling Errors

Handling errors is critical to providing printing support in any application. An application should handle any errors it gets, making sure it displays localized error strings to the user. Providing localized strings in Mac OS X is fairly easy if you define them in a separate file named `Localizable.strings` and follow the Mac OS X convention to put localized versions of the file into the appropriate language-specific folder.

You can create a `Localizable.strings` file in Project Builder by choosing New File from the File menu. Then, create an empty file named `Localizable.strings` and add it to the Resources group of your project. See *Inside Mac OS X: System Overview* for detailed information on localizing strings, string file syntax, functions for retrieving strings, and information on generating a string file automatically.

Listing 3-10 (page 43) shows a function that displays an alert to notify the user that a printing error has occurred. The alert message includes a localized string and the error number. A detailed explanation for each line of code that has a numbered comment appears following the listing.

The function assumes a `Localizable.strings` file exists and contains a formatting string. A typical localized formatting string for a printing application would be:

```
"Print error format" = "There is an error in the printing code. Error number:
%d.";
```

The string `"Print error format"` is the key. It is the string you pass to the function in Listing 3-10 as the `errorFormatStringKey` parameter. The string `"There is an error in the printing code. Error number: %d."` is the localized formatting string; it uses printf-style formatting.

Listing 2-10 A function to post a printing error alert

```
void MyPostPrintingError (OSStatus status,
                          CFStringRef errorFormatStringKey)                 // 1
{
    CFStringRef formatStr = NULL,
                printErrorMsg = NULL;
    SInt16      alertItemHit = 0;
    Str255      stringBuf;

    if ((status != noErr) && (status != kPMCancel))                         // 2
    {
        formatStr =  CFCopyLocalizedString (errorFormatStringKey, NULL);    // 3
        if (formatStr != NULL)
        {
            printErrorMsg = CFStringCreateWithFormat(
                        NULL, NULL,
                        formatStr, status);                                 // 4
            if (printErrorMsg != NULL)
            {
                if (CFStringGetPascalString (printErrorMsg,
                            stringBuf, sizeof (stringBuf),
                            GetApplicationTextEncoding()))                  // 5
                 {
```

```
                StandardAlert(kAlertStopAlert, stringBuf,
                                NULL, NULL, &alertItemHit);          // 6
            }
            CFRelease (printErrorMsg);                               // 7
        }
        CFRelease (formatStr);                                       // 8
    }
}
}
```

Here's what the code in Listing 3-10 (page 43) does:

1. The function takes two parameters: a result code (`status`) and a string (`CFStringRef`) that specifies a key associated with the localized format string.
2. Checks whether the passed error should be displayed. Any error except `kPMCancel`, indicating the user cancelled printing, should be displayed.
3. Calls the Core Foundation Bundle Services macro `CFCopyLocalizedString` to search the default strings file (`Localizable.strings`) for a localized format string that indicates how the error should be formatted for display. You need to pass the key (`errorFormatStringKey`) for the localized string you wish to retrieve. The second parameter is optional—it is a comment (`CFStringRef`) to help translators by giving them context or other hints about how the string is used or how to translate it. If you don't provide a comment, you should pass `NULL`.
4. If no error occurs, calls the Core Foundation String Services function `CFStringCreateWithFormat` to create a copy of the error string that includes the error number. The first parameter is a reference to an allocator to be used to create the `CFString` object. You can pass `NULL` to request the default allocator. The second parameter refers to an undocumented feature, so pass `NULL`. The third parameter is a reference to a `CFString` object that contains a string with printf-style specifiers. The remaining parameters are arguments for the printf-style string. In this case there is only one, a status code.
5. Calls the Core Foundation String Services function `CFStringGetPascalString` to get a copy of the string in a format that can display in a standard alert dialog.
6. Calls the Dialog Manager function `StandardAlert` to display the error message.
7. Calls the Core Foundation Base Services function `CFRelease` to release the string (`printErrorMsg`) created by the call to `CFCopyLocalizedString`.
8. Calls the function `CFRelease` to release the string (`formatStr`) created by the call to `CFStringCreateWithFormat`.

Saving a Document as a PDF File

This section provides information on what you need to do in your code to implement a command to save a document as a PDF file. In Mac OS X saving a document as a PDF file is simple. Your application needs to set the printing destination to a file location instead of to a printer. You set the destination by calling the function `PMSessionSetDestination`.

Note: The function `PMSessionSetDestination` is available in Mac OS X version 10.1 and later.

Listing 3-11 shows a code fragment that sets the destination to a PDF file. You need to supply the Carbon Printing Manager constants `kPMDestinationFile` to specify that the destination is a file and `kPMDocumentFormatPDF` that the document format is PDF. The parameter `saveURL` specifies the location to save the PDF file.

When you call the function `PMSessionSetDestination` you must

- call the function `PMSessionSetDestination` after a call to `PMCreateSession` and before you release the printing session object.
- use the same printing session object for the function `PMSessionSetDestination` as you use when you set defaults for the print settings object (`printSettings`).
- set the destination before you call your print loop code.

Listing 2-11 Setting the destination as a PDF file

```
if (status == noErr){
        status = PMSessionSetDestination (printSession,
                                printSettings,
                                kPMDestinationFile,
                                kPMDocumentFormatPDF,
                                saveURL);
 }
```

Note: If you want the user to specify the location, you need to write functions that call the Navigation Services API. You call your printing code after the user has specified the location in the Navigation Services dialog.

In addition to setting the destination as a PDF file, you need to decide how your application handles printing dialogs (Page Setup and Print) and whether you want the printing system to display the printing status dialog.

When users save a document, they expect to see a dialog in which they can specify a filename and file location. They do not expect to see Page Setup or Print dialogs. In most cases, your application should not display either of the printing dialogs to the user in response to the Save As PDF command. You need to write your printing code so that the print settings object is set up programmatically rather than by the user. Your application should use the current page format that is associated with the document.

The printing status dialog is automatically shown by the printing system when an application calls these functions: `PMSessionBeginDocument`, `PMSessionEndDocument`, `PMSessionBeginPage`, and `PMSessionEndPage`. The printing status message informs the user of the page being printed (Printing Page 1, Printing Page 2, and so forth). A printing status message is probably not appropriate to show for file saving so you should suppress the printing status dialog.

To suppress the printing status dialog, your application needs to use the "No Status Dialog" versions of these functions. Specifically, you call

- `PMSessionBeginDocumentNoDialog` instead of `PMSessionBeginDocument`
- `PMSessionEndDocumentNoDialog` instead of `PMSessionEndDocument`
- `PMSessionBeginPageNoDialog` instead of `PMSessionBeginPage`

- `PMSessionEndPageNoDialog` instead of `PMSessionEndPage`

Otherwise, the code you use to implement the print loop is the same as what you'd use to print a print job triggered by a user clicking Print in the Print dialog.

Listing 3-12 (page 46) shows a code fragment that assigns function names to an application-defined structure based on whether the status dialog should be shown. Then, the application passes a pointer to the structure to its print loop function (`MyDoPrintLoop`). You would need to modify your application's print loop function to require a parameter that specifies a pointer to the application-defined structure containing the function names and call the appropriate function from the structure.

Listing 2-12 Setting up a print loop to use the No Dialog functions

```
 if (status == noErr)
 {
#if NO_STATUS_DIALOG
        PrintingProcs myPrintingProcs = {PMSessionBeginDocumentNoDialog,
                        PMSessionEndDocumentNoDialog,
                        PMSessionBeginPageNoDialog,
                        PMSessionEndPageNoDialog};
#else
        PrintingProcs myPrintingProcs = {PMSessionBeginDocument,
                        PMSessionEndDocument,
                        PMSessionBeginPage,
                        PMSessionEndPage};
#endif

                status = MyDoPrintLoop(printSession,
                            pageFormat,
                            printSettings,
                            documentDataP,
                            &myPrintingProcs);
 }
```

Printing One Copy

The Print One Copy command is often provided by applications as a shortcut to allow a user to print a single copy of a document using default settings. Because default print setting values are used, there is no need for your application to display the Print dialog. Instead your application should set up the print settings objects programmatically and should use the current page format that is associated with the document. Otherwise, the code you use to implement the print loop is the same as what you'd use to print a print job triggered by a user clicking Print in the Print dialog.

Printing Multiple Copies

The Mac OS X printing system automatically handles printing multiple copies. Your application does not need to perform any tasks other than specifying the number of copies in the printing session object.

Note: In non-Carbon printing in Mac OS 9 and earlier your application had to iterate through the print loop for each copy you wanted to print. This is no longer true. If you use this approach in Mac OS X, you will get unsatisfactory results.

Adopting the Carbon Printing Manager

This chapter provides information that will help you convert printing code that uses the old Printing Manager so that your code uses the Carbon Printing Manager. The Carbon Printing Manager was designed to let Carbon applications take advantage of new features in Mac OS X, while still working correctly in previous versions of the Mac OS.

If you want to port an application that runs in Mac OS 9 and earlier versions of the Mac OS to one that runs in Mac OS X, you should read *Inside Carbon: Carbon Porting Guide*. The guide covers all porting topics, except for porting printing code, which is covered in this chapter.

Updating your application to use the Carbon Printing Manager is a straightforward process. These are the basic steps:

1. Remove all references to the `Printing.h` header file from your project.
2. If you are building your application so that is runs only in Mac OS X, make sure you link against the Application Services and Carbon frameworks. If you are building your application so that it can run in Mac OS 9 as well as in Mac OS X, you also need to add the `CarbonLib` library to your project.
3. Convert your code to use the new printing functions and opaque data types, as described in the following sections.

New and Changed Printing Functions

Because the Carbon Printing Manager replaces all of the functions in the old Printing Manager, the first step in converting your code is to locate and replace your old Printing Manager function calls with their equivalents from the Carbon Printing Manager. In most cases there is a one-to-one mapping between the new functions and the original functions they replace. Table 4-1 (page 49) lists the Carbon equivalents for old Printing Manager functions.

Table 3-1 Carbon replacements for functions in the old Printing Manager

Classic function	Carbon Printing Manager function
`PrOpen`	`PMCreateSession`
`PrClose`	`PMRelease`
`PrOpenDoc`	`PMSessionBeginDocument`
`PrCloseDoc`	`PMSessionEndDocument`
`PrOpenPage`	`PMSessionBeginPage`

Classic function	Carbon Printing Manager function
PrClosePage	PMSessionEndPage
PrintDefault	PMSessionDefaultPrintSettings
	PMSessionDefaultPageFormat
PrValidate	PMSessionValidatePrintSettings
	PMSessionValidatePageFormat
PrJobInit	PMSessionPrintDialogInit
PrJobDialog	PMSessionPrintDialog
PrStlInit	PMSessionPageSetupDialogInit
PrStlDialog	PMSessionPageSetupDialog
PrDlgMain	PMSessionPrintDialogMain
	PMSessionPageSetupDialogMain
PrGeneral	PMSessionGeneral Specific accessors replace common calls, see the Carbon Manager Reference documentation.
PrSetError	PMSessionSetError
PrError	PMSessionError
MyDoPrintIdle	PMIdleProcPtr
MyPrDialogAppend	PMPageSetupDialogInitProcPtr
	PMPrintDialogInitProcPtr

Some of the functionality provided by the old Printing Manager is no longer supported. Table 4-2 (page 50) lists the old functions that are not supported by the Carbon Printing Manager.

Table 3-2 Old Printing Manager functions that are not supported in Carbon

PrPicFile
PrPurge
PrNoPurge
PrLoadDriver
PrDrvrDCE
PrDrvrOpen
PrDrvrClose

PrDrvrVers
PrCtlCall
PrJobMerge

The PrPicFile function was removed because the "deferred" printing style is no longer supported. All print records must use "draft" style, and printer drivers must perform their own despooling or, in Mac OS 8 and 9, use the Desktop Printer Spooler. Refer to the old Printing Manager documentation for information about draft and deferred printing styles. The Desktop Printer Spooler is described in Tech Note 1097.

A direct replacement for the PrJobMerge function is unnecessary because the PMPageFormat and PMPrintSettings objects can be used independently. You can print multiple documents, each with their own saved page format, using a single PMPrintSettings object.

New Data Types and Accessor Functions

The Carbon Printing Manager replaces the old Print Manager print record (TPrint) with two opaque data types, the PMPrintSettings object and the PMPageFormat object. All references to elements of the TPrint record in your code must be updated to refer to one of these objects, using the new accessor functions provided.

Because the PMPrintSettings and PMPageFormat objects are opaque, your application must not make assumptions about their size or internal structure. You will need to update any code that loads, stores, or directly manipulates the TPrint record.

Table 4-3 (page 51) lists the accessor functions you can use to examine elements of the PMPrintSettings and PMPageFormat objects.

Table 3-3 Carbon accessor functions for the old Print Manager print record (TPrint) fields

Data structure	Element	Accessor function
TPrint	iPrVersion	Not supported (Use PMPrinterGetDriverReleaseInfo to obtain a driver's version strings. The version strings are not normally needed.)
	prInfo	See TPrInfo
	rPaper	PMGetAdjustedPaperRect
	prStl	See TPrStl
	prInfoPT	Not supported
	prXInfo	not supported
	prJob	See TPrJob
	printX[19]	Not supported

Data structure	Element	Accessor function
TPrInfo	iDev	Not supported
	iVRes	PMGetResolution
	iHRes	PMGetResolution
	rPage	PMGetAdjustedPageRect
TPrStl	wDev	PMGetOrientation
	iPageV	PMGetUnadjustedPaperRect
	iPageH	PMGetUnadjustedPaperRect
	bPort	Not supported
	feed	Not supported
TPrJob	iFstPage	PMGetFirstPage
	iLstPage	PMGetLastPage
	iCopies	PMGetCopies
	bJDocLoop	Not supported
	fFromUsr	Not supported
	pIdleProc	PMSessionSetIdleProc This does nothing in Mac OS X; only used for Mac OS 8 and 9.
	pFileName	Not supported
	iFileVol	Not supported
	bFileVers	Not supported
	bJobX	Not supported

When you use these accessor functions, be sure to pass the appropriate constant for any parameters you do not want to pass to the function or receive from it. For example, pass the kPMDontWantData constant in place of a parameter that returns data you're not interested in.

Your application must dispose of any structures, references, or other data returned by Carbon Printing Manager functions. Your application should also release the PMPrintSettings and PMPageFormat objects it creates when they are no longer needed.

Supported PrGeneral Opcodes

The Carbon Printing Manager provides the `PMSessionGeneral` function as a replacement for the old Printing Manager function `PrGeneral`. However, Apple suggests that you reduce your reliance on these functions because they are not currently supported by all printer drivers, and because they are not likely to be supported in future versions of the Mac OS.

Table 4-4 (page 53) lists `PrGeneral` opcodes that are supported in Carbon in Mac OS 8 and 9, but not in Mac OS X. For opcodes that have an associated accessor function, you use that function instead of passing the opcode to `PMSessionGeneral`. For example, use `PMGetOrientation` instead of passing the `getRotnOp` constant to `PMSessionGeneral`. The `PMSessionGeneral` function returns the result code `kPMNotImplemented` for any unsupported opcodes, and for opcodes that have Carbon accessor functions.

Table 3-4 Carbon support for PrGeneral opcodes

Opcode	Value	Accessor function
`GetRslDataOp`	4	`PMPrinterGetIndexedPrinterResolution`
`SetRslOp`	5	`PMSetResolution`
`DraftBitsOp`	6	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`NoDraftBitsOp`	7	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`getRotnOp`	8	`PMGetOrientation`
`NoGraySc1`	9	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`GetPSInfoOp`	10	`PMSessionGeneral` supports this opcode in Mac OS 8/9; `PMPrinterGetLanguageInfo` supported in Mac OS 8/9 and Mac OS X
`PSIntentionsOp`	11	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`EnableColorMatchingOp`	12	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`PSAdobeOp`	14	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`PSPrimaryPPDOp`	15	`PMPrinterGetDescriptionURL`
`kLoadCommProcsOp`	16	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`kUnloadCommProcsOp`	17	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`kPrinterDirectOpCode`	20	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`kPrVersionOp`	22	`PMPrinterGetDriverCreator`
`kGetPrinterInfo`	23	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`kIsSamePrinterInfo`	24	`PMSessionGeneral` supports this opcode in Mac OS 8/9
`kSetDefaultPrinterInfo`	25	`PMSessionGeneral` supports this opcode in Mac OS 8/9

Opcode	Value	Accessor function
kPrEnablePartialFonts	26	PMSessionGeneral supports this opcode in Mac OS 8/9

Supported Picture Comments

Table 4-5 (page 54) lists picture comments supported by the Carbon Printing Manager. Except where noted, these have the same behavior in Mac OS X as they do with LaserWriter 8 in Mac OS 8 and 9. Note that by default PostScript picture comments are ignored in Mac OS X; they are supported only when printing with LaserWriter 8 compatibility mode.

Table 3-5 Picture comments supported by the Carbon Printing Manager

Name	Value	Data size	Data handle	Description
Text picture comments				
TextBegin	150	6	TTxtPicRec	Begin text function
TextEnd	151	0	NULL	End text function
StringBegin	152	0	NULL	Begin string delimitation
StringEnd	153	0	NULL	End string delimitation
TextCenter	154	8	TCenterRec	Offset to center of rotation for text
LineLayoutOff	155	0	NULL	Turn printer driver's line layout off
LineLayoutOn	156	0	NULL	Turn printer driver's line layout on
ClientLineLayout	157	16	TClientLLRec	Customize line layout error distribution; not supported in Mac OS X
Graphics picture comments				
PolyBegin	160	0	NULL	Begin special polygon
PolyEnd	161	0	NULL	End special polygon
PolyIgnore	163	0	NULL	Ignore following polygon data
PolySmooth	164	1	TPolyVerbRec	Mark that the polygon should be smoothed and indicate whether to frame, fill, and/or close

Name	Value	Data size	Data handle	Description
PolyClose	165	0	NULL	Mark the polygon as closed
RotateBegin	200	8	TRotationRec	Begin rotated port
RotateEnd	201	0	NULL	End rotation
RotateCenter	202	8	TCenterRec	Offset to center of rotation
Line-drawing picture comments				
DashedLine	180	Size of a TDashedLineRec record	TDashedLineRec	Draw following line as dashed
DashedStop	181	0	NULL	End dashed lines
SetLineWidth	182	4	TLineWidthHdl	Set fractional line widths
PostScript picture comments are provided only for LaserWriter 8 compatibility.				
PostScriptBegin	190	0	NULL	Set driver state to PostScript
PostScriptEnd	191	0	NULL	Restore QuickDraw state
PostScriptHandle	192	Size of the PostScript data in handle	a handle	PostScript data referenced by handle
PostScriptFile	193	Size of the PostScript data in handle	a handle	Filename referenced by handle
TextIsPostScript	194	0	NULL	If compatible with LaserWriter8, QuickDraw text is sent as PostScript
PSBeginNoSave	196	0	NULL	Set driver state to PostScript
ColorSync picture comments				
cmBeginProfile	220	variable	version 1 profile data	Begin ColorSync profile; not supported in Mac OS X
cmEndProfile	221	0	NULL	End ColorSync profile; not supported in Mac OS X

Name	Value	Data size	Data handle	Description
`cmEnableMatching`	222	0	`NULL`	Begin ColorSync color matching
`cmDisableMatching`	223	0	`NULL`	End ColorSync color matching
`cmComment`	224	variable	a handle	Contents are a selector with data following

Document Revision History

This table describes the changes to *Supporting Printing in Your Carbon Application*.

Date	Notes
2004-08-31	Fixed formatting for Listing 3-8 (page 39).
2002-12-03	Added information on printing multiple copies.
	Updated formatting.
2002-03-01	Added an index. Corrected several minor typographical errors.
2001-01-01	First version of this document. Chapter 4, "Adopting the Carbon Printing Manager," (page 49) is a revision of the standalone document of the same name, with content updated to reflect the latest Carbon Printing Manager API.

Document Revision History

Index

N

O

P

S

T