Copland Technical Overview

Draft

Developer Press

Contents

About Copland

Copland is the development name for the next major release of the Mac OS. This document presents an overview of the design of Copland, describes many of the benefits that its implementation offers to the users of your software or hardware products, and discusses how your products can take advantage of Copland's new features.

With Copland, Apple Computer is redesigning the core of the Mac OS to provide concurrency, higher performance, and enhanced stability. Featuring a modular, microkernel design, this new OS provides

- preemptive task scheduling
- memory protection for critical system code and data, such as device drivers and file system extensions
- improved virtual memory
- a shared, dynamically loaded, file-mapped version of the Mac OS
- a concurrent and more flexible File Manager
- a new, higher performance, concurrent I/O architecture

Copland also delivers dramatic improvements in ease of use and new customizing capabilities. Significant improvements to the Macintosh user experience include

- coordinated sets of designs—affecting the appearance of onscreen human interface elements, such as windows, menus, fonts, and controls—with which users can personalize their computers
- the ability for multiple users to individually customize and share a single computer
- visual and behavioral consistency among commonly implemented human interface elements, such as floating windows, slider controls, and tear-off menus
- better access to documents and related information from within an application and from the Finder

- a system for helping users achieve their goals instead of teaching them merely how to manipulate applications and system software
- user assistance in automating repetitive tasks

Making refined new versions of essential Macintosh technologies available to all of its users, Copland includes as standard features

- the object-based flexibility of OpenDoc
- the integration and unification of the graphics, text, and printing capabilities of QuickDraw, QuickDraw GX, QuickDraw 3D, and WorldScript
- network transport transparency and independence through Open Transport
- integrated collaboration services with PowerTalk

In addition to improving user productivity with these technologies, Copland offers noteworthy speed increases to users of PowerPC-based computers, because its OS and Toolbox have been rewritten as native implementations on PowerPC processors.

Many of the benefits of Copland are available to your application (and its users) regardless of whether it adopts the new Copland APIs, because Copland offers a high-degree of compatibility with System 7 applications. However, for increased performance, additional functionality, and future extensibility in your product, you should consider adopting the Copland APIs as quickly as possible. (As you might guess, decreasing numbers of System 7 APIs will be supported in versions of the Mac OS beyond Copland.) So adopting Copland APIs in your application helps ensure a smooth development path to future versions of the Mac OS.

Just as Copland provides many improvements for users, so too does it offer new opportunities for you as a developer. For example, the modular design of the OS makes it much easier for Mac OS licensees to port Copland to their own hardware platforms, increasing the market for all other Mac-related products. Copland simplifies product development in many other significant ways. For example, the new, structured I/O architecture simplifies device driver development; the flexible new File Manager makes it easier for your product to support industry-standard volume formats in addition to an improved version of the hierarchical file system (HFS); and the new Toolbox supplies extensible user-interface features demanded by application developers.

The chapters in this document explain in general how the Copland architecture supports these features for users and developers. Note that because of the preliminary nature of this document, many of the terms and names used within it, as well as interface elements shown in its figures, are subject to change. As in the past, however, Apple Computer will strive to provide you with the most up-to-date information possible through its many developer products and offerings.

We look forward to helping you run your software products on Copland.

C H A P T E R 1

About the Copland Human Interface and Toolbox

Contents

This document describes the new human interface features in Copland and related changes to the Mac Toolbox.

The major human interface improvements introduced by Copland include

- a more consistent human interface
- more options for customization by users
- a new form of onscreen assistance that allows users to set up and perform complex tasks by answering a few simple questions
- improved access to information, including a flexible replacement for the Standard File Package

The Toolbox has been rewritten for Copland to support the new human interface features and run native on PowerPC processors. From a programmer's point of view, the most important architectural changes to the Toolbox fall into four categories:

- **Improved opacity.** Toolbox managers have been simplified and integrated to provide a complete programming model that doesn't require direct manipulation of underlying data structures.
- **Integrated support for globalized software.** The Toolbox takes advantage of the new systemwide text objects APIs to provide flexible support for multilingual text throughout the human interface.
- **Data extensibility.** The Toolbox uses the Collection Manager to support the addition of arbitrary tagged data to Toolbox data structures without manipulating the structures directly.
- **Design extensibility.** The Toolbox includes a comprehensive set of standard windows, menus, controls, and other human interface elements that can be used as is or extended by developers to support specialized application needs.

By supporting the Copland human interface and Toolbox, you can create applications that look and behave more consistently and reliably and take advantage of the new customization, assistance, and information access features.

Compatibility—Backward and Forward

Code written to the Copland APIs not only ensures compatibility with Copland but also lays the foundation for new capabilities that will be introduced in future Mac OS enhancements.

Like any major system software revision, Copland introduces features that aren't backward compatible with earlier systems. However, most applications written to System 7.5 APIs can run on Copland, even though they may not be able to take advantage of all its features. For example, clients of standard System 7.5 definition procedures (defprocs) work correctly and inherit the Copland human interface appearance. Custom defprocs written for System 7.5 also work correctly on Copland but do not inherit the Copland appearance.

Here are some general compatibility guidelines for System 7.5 applications:

- Don't assume that dialog box backgrounds are white. The Copland human interface supports a variety of background colors.
- For floating windows, use the standard floating window definition (ID 124) introduced in System 7.5. This window definition works correctly on Copland and inherits the Copland appearance.
- Don't hard code any assumptions about the precise locations of human interface elements such as close boxes, zoom boxes, and window titles within the noncontent areas of windows or dialog boxes.
- Don't hard code any assumptions about the precise locations of any human interface elements in the Save and Open dialog boxes. Use the relative position of the standard elements to determine the locations of new ones.
- Always use low-memory accessor functions.
- Use data structure accessor functions where they exist. For example, use `SetMenuItemText` and `GetMenuItemText` to manipulate menu item text rather than accessing the data structure directly.
- If data structure accessor functions aren't available, isolate the code that accesses data structures directly. Copland provides accessor functions for all data structures, and it is easier to take advantage of them if you have isolated the code that needs to be updated.

- Don’t manipulate the window list directly. Use the `BringToFront` and `SendBehind` functions instead.
- Compile your application with the `STRICT` compiler option to ensure that your application can be supported as Copland migrates to future versions of the Mac OS.

Design Goals for the Copland Human Interface

After careful analysis of customer priorities, Apple has established these design goals for the Copland human interface and related Toolbox capabilities:

- Provide a more consistent human interface for system software and for all Copland applications. For example, pop-up menus in different applications should work the same way and should match the appearance of other human interface elements.
- Support customization by individual users, including the appearance of human interface elements, the scaling (complexity) of available features, and separate work areas for users who share a single computer. For example, multiple users of one computer should be able to set up their own computing environments—including the details of systemwide appearance, application preferences, the organization of the desktop, and the complexity of available features—and let the computer handle the details of switching between one environment and the other.
- Support “do it for me” behavior and provide a new high-level assistance mechanism that allows users to concentrate on achieving goals rather than learning how to manipulate applications and system software.
- Provide better access to documents and related information from within an application and from the Finder.
- Provide more comprehensive APIs that are consistent across all Toolbox managers and shield applications from the details of the underlying system data structures.
- Provide better support for globalized software, including localization of human interface elements and multilingual text; making it possible, for example, to use multilingual text easily within a menu or a window title.

- Support a larger number of commonly used human interface elements, such as sliders, pop-up menus, modifier keys in menus, and tear-off menus, and facilitate the reuse of standard code for customized elements.

About the Copland Human Interface

Copland makes the Macintosh human interface more flexible from a user's point of view and at the same time makes it easier for a developer to implement. The major areas of change include a new approach to appearance and customization, new forms of assistance, and improved access to information.

Appearance and Customization

As Macintosh applications have become more powerful and more complex, developers have tended to circumvent various parts of system software to implement new capabilities. This has led to idiosyncratic solutions to common problems. For example, the appearance and behavior of pop-up menus, dialog boxes, icon buttons, and sliders tend to differ from one application to another.

Inconsistent human interface implementations lead to user frustration, higher training costs, and reduced productivity. Improved consistency is therefore one of the major design goals for the Copland human interface.

However, consistency should not come at the price of flexibility. Users want to be able to customize their computing environments to suit the specific tasks they perform. Users also want to be able to share their computers easily with others while retaining separate customized environments.

The Copland Toolbox addresses these demands for improved consistency and customization by supporting these key features of the Copland human interface:

- Users can select different *themes* or styles—that is, coordinated sets of human interface designs that determine the appearance and behavior of human interface elements on a systemwide basis. For example, a teacher might want to use a theme with bright colors, animated menus, and sounds, whereas a businessperson might prefer a more conservative, classic theme. Regardless of the theme, the core Macintosh experience remains the same, and users can switch themes without having to learn new human interface metaphors.

- Users can select different *workspaces* or desktops—that is, separate customized environments for a single computer. Each workspace can have its own theme, control panel options, desktop arrangement, screen saver settings, application preferences, and so on. For example, a user with a home office could set up one password-protected workspace for business tasks and several additional workspaces for use by other members of the family. If desired, each user's access to available system and application features can be controlled, or *scaled,* according to the user's skill level, and each workspace can be password-protected.
- Developers can use standard code and graphic designs for a variety of human interface elements in your application, including many elements that must be created from scratch in System 7.5. At the same time, it's much easier to customize the standard elements or, if necessary, to create new ones without replacing large chunks of code provided by the system. These capabilities depend on *interface definition objects (IDOs),* which are SOM-based replacements for the defprocs used in System 7.5.

IMPORTANT

The terms *theme, workspace,* and others noted elsewhere in this chapter are preliminary names for use during Copland development. These names are likely to change. ▲

The sections that follow introduce the Copland human interface capabilities from a user's point of view. For information about Toolbox support for these features, see "About the Copland Toolbox," beginning on page 1-25.

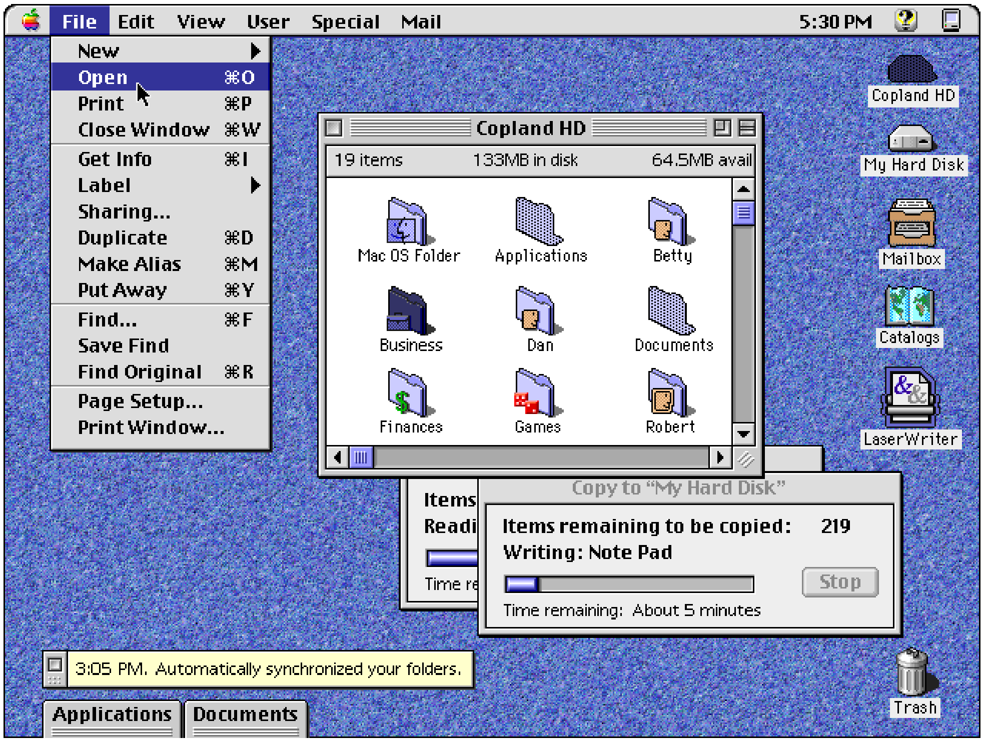
Themes

A **theme** is a coordinated set of designs that determines the appearance of all human interface elements on the screen, including alert icons, controls, background colors, dialog boxes, menus, screen savers, state transitions, system font, and windows. Apple plans to supply several standard themes with Copland and will encourage third-party theme development in the future. Users can choose among the themes available to the system with the Appearance control panel. Eventually users will be able to buy and install additional themes. One theme, the Apple Default theme, is permanently built into the system.

In addition to supporting user customization, themes and the underlying IDO mechanism insulate your application from future changes to the human interface. They free you from relying on hardwired appearances for standard elements while making it easier to create customized elements. Because Copland allows you to deal with appearance abstractions rather than specific details, your application can support not only the new human interface designs in Copland but also future design enhancements. Themes are designed to support future growth of the Mac OS as well as customization by users.

Figure 1-1 shows a preliminary design for the Apple Default theme. This design provides a standard 3-D appearance appropriate for a business or professional user. The Apple Default theme resembles the appearance of the System 7.5 human interface more closely than other themes supplied by Apple.

Figure 1-1 Preliminary design for the Apple Default theme

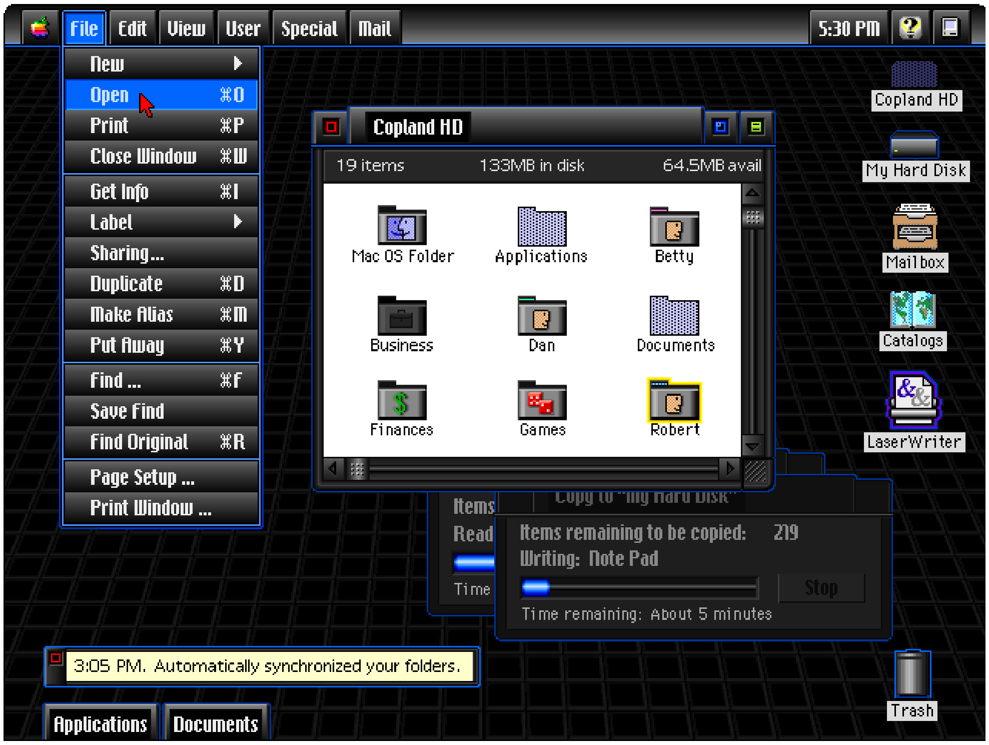


Items remaining to be copied: 219
Writing: Note Pad
Time remaining: About 5 minutes

3:05 PM. Automatically synchronized your folders.

Figure 1-2 shows another preliminary design for a theme.

Figure 1-2 Another potential theme



19 items 133MB in disk 64.5MB avail

Copy to "My Hard Disk"

Items remaining to be copied: 219

Writing: Note Pad

Time remaining: About 5 minutes

3:05 PM. Automatically synchronized your folders.

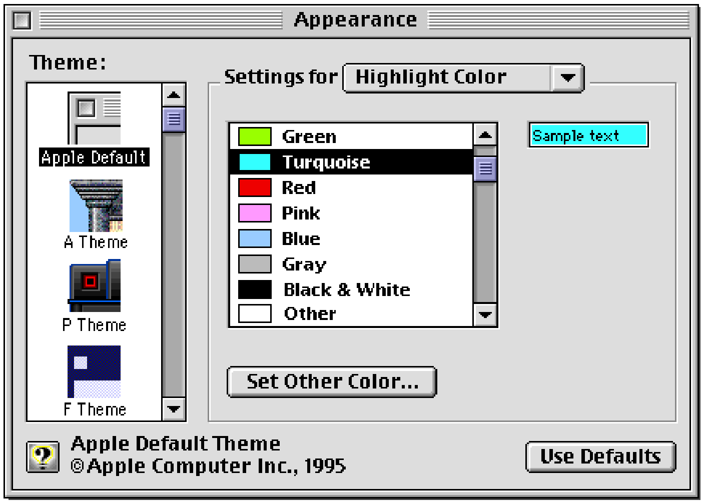
IMPORTANT
The designs shown in Figure 1-1, Figure 1-2, and other figures in this chapter are preliminary and likely to change. ▲

The Appearance Control Panel

The Appearance control panel allows users to specify the current theme and other aspects of their computing environment's appearance. It also replaces the Desktop Patterns and Color control panels in System 7.5.

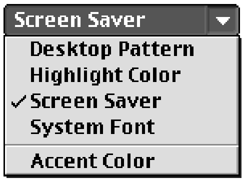
As shown in Figure 1-3, the Appearance control panel displays the themes that are currently installed in the system. It also displays several panels—that is, groups of related settings—that allow users to adjust specific appearance attributes of each theme.

Figure 1-3 Preliminary design for the Appearance control panel



Each theme supports four built-in panels and can also display additional theme-specific panels. The pop-up menu in Figure 1-4 lists the panels available for the Apple Default theme.

Figure 1-4 Pop-up menu listing the panels available for the Apple Default theme



Note
The panels available in the Appearance control panel, the Save dialog box, and elsewhere in the Copland human interface are in turn composed of smaller panels that encapsulate individual human interface elements. For more information, see “Panels,” beginning on page 1-33. ◆

Standard Human Interface Elements and Primitives

The Copland Toolbox provides a much broader variety of standard behaviors and standard human interface elements than System 7.5, including ready-made menus, dialog boxes, radio buttons, sliders, and other controls. Unlike standard elements in earlier versions of system software, those provided by Copland can be easily customized without wholesale duplication of system code. “Summary of New Toolbox Features,” beginning on page 1-29, introduces some of the standard elements provided by each Toolbox manager.

Copland also provides a mechanism for specifying bevels, fills, window headers, and other basic shapes that you can combine to create custom, theme-compatible visual elements for specialized purposes or to coordinate content areas (for example, background fills) with the current theme. These capabilities are described in “Appearance Manager,” beginning on page 1-29.

Whenever possible, you should use the standard windows, menus, controls, and so on provided by the Copland Toolbox. This is the easiest way to support themes. If you need to create custom elements, you have two choices:

- **Customize standard elements.** You can modify just those characteristics of a standard element that you wish to implement differently while maintaining compatibility with the current theme.
- **Implement your own theme-compatible elements.** You can assemble custom human interface elements from primitives and fills that maintain compatibility with the current theme.

An application's content areas are entirely under the application's control. Apple encourages developers to support themes wherever it makes sense for an application to be coordinated visually with the rest of the Macintosh human interface. For more information about Toolbox support for themes and custom human interface elements, see "Appearance Manager," beginning on page 1-29.

Multiple Workspaces

Many Macintosh computers are shared by multiple users. Although a single user can customize a single computer in many ways—including the arrangement of files and folders on the desktop, system preferences, and application preferences—System 7.5 can keep track of only one set of customizations at a time.

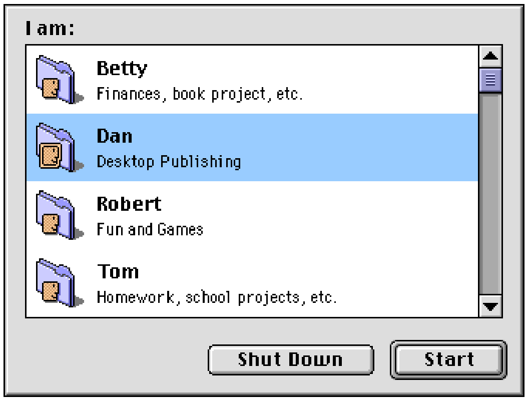
Copland allows users to set up several different workspaces for a single computer. A **workspace** maintains an entire user context, including

- application and system preferences
- icons (other than mounted volumes) on the desktop and open Finder windows
- Apple menu items
- control panels
- startup and shutdown items
- user's name and password
- user level, or **scaling**—that is, the complexity of available menus and features

Workspaces make it much easier for several users to share a single computer, thus meeting one of the primary design goals for the Copland human interface.

If a system is configured for more than one workspace, the user is asked to choose a workspace either at startup or when switching workspaces. During startup, for example, a dialog box like that shown in Figure 1-5 appears.

Figure 1-5 Dialog box for choosing a workspace


I am:

Betty
Finances, book project, etc.

Dan
Desktop Publishing

Robert
Fun and Games

Tom
Homework, school projects, etc.

Each workspace is associated with a specific user name. If the workspace has a password, the user is prompted for it after choosing a name. A similar dialog box allows a user to switch workspaces after startup. Switching workspaces doesn't require restarting the computer, but it does cause all currently running applications to quit.

Copland includes a Preferences Manager. Using the Preferences Manager hides the details of preference management from your application and enhances its capabilities in a multiple-workspace environment; for example, by providing user variables and global variables.

Active Assistance

The human interface for most personal computers tends to place the computer in a passive role. Users accomplish tasks primarily by manipulating user interface elements directly, and they tend to discover new features through trial and error. As a result many users don't take advantage of the most powerful capabilities of their software and hardware.

Apple Guide in System 7.5, the Copland Assistance Manager, and related Copland features represent the first steps along a new path for the Mac OS human interface, a path that will eventually transform the computer from a purely passive tool to a cooperative partner that actively helps the user get work done. Instead of focusing on how to manipulate applications, users should be able to focus on how to accomplish their goals. The Copland Toolbox provides integrated support throughout the system for **active assistance,** which extends the capabilities of Apple Guide to assist the user actively. For example, active assistance makes it possible to present high-level questions to the user and make decisions based on the answers.

Using the Copland Toolbox APIs as recommended by Apple helps to ensure that your application can support active assistance. If you are working on a System 7.5 application, providing a Guide file is an important first step toward supporting active assistance on Copland.

The Copland Assistance Manager supports two additional capabilities that underlie active assistance: automation and delegation. **Automation** is the ability to create a sequence of actions, or task, in a form that can be used to control operations automatically. Instructional designers and scripters can use various automation technologies to generate tasks for the user. **Delegation** is the ability to trigger tasks at a specified time or when a specified event occurs. Delegation allows the computer to be productive even when the user isn't present.

Copland features that support automation, delegation, and related assistance mechanisms include the following:

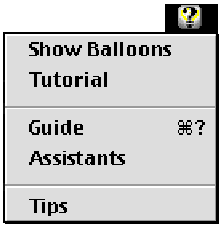
- **Quick Assist** provides user-defined key combinations that, as long as the specified keys are pressed, enable balloons or context-sensitive access to Apple Guide files.
- **Apple Guide 3.0** includes enhanced access and presentation windows, improved support for performing actions on the user's behalf, and support for interview sequences that collect information about the user's goals.

- **Task Lister** is an application for viewing and managing automated tasks.
- **Task Log** is a window for reviewing a log of automated task activities and messages.
- **Finder objects** related to assistance include a new Assistance folder and a new kind of file for storing information related to automated tasks.

Note
The terms *automation*, *delegation*, *Task Lister*, and *Task Log* are preliminary names that are likely to change. ◆

Figure 1-6 shows the Assistance menu, which extends the Guide menu in System 7.5 to support access to active assistance. The Guide menu item is always available and invokes the Guide file provided by the active application or by the Finder. The Assistants menu item provides access to specific kinds of high-level active assistance that also vary according to the context. The Tips menu item is similar to the Shortcuts menu item in System 7.5, except that it permits more user interaction and lets users choose to have tips present themselves in contexts where they are likely to be helpful.

Figure 1-6 The Assistance menu while the Finder is frontmost

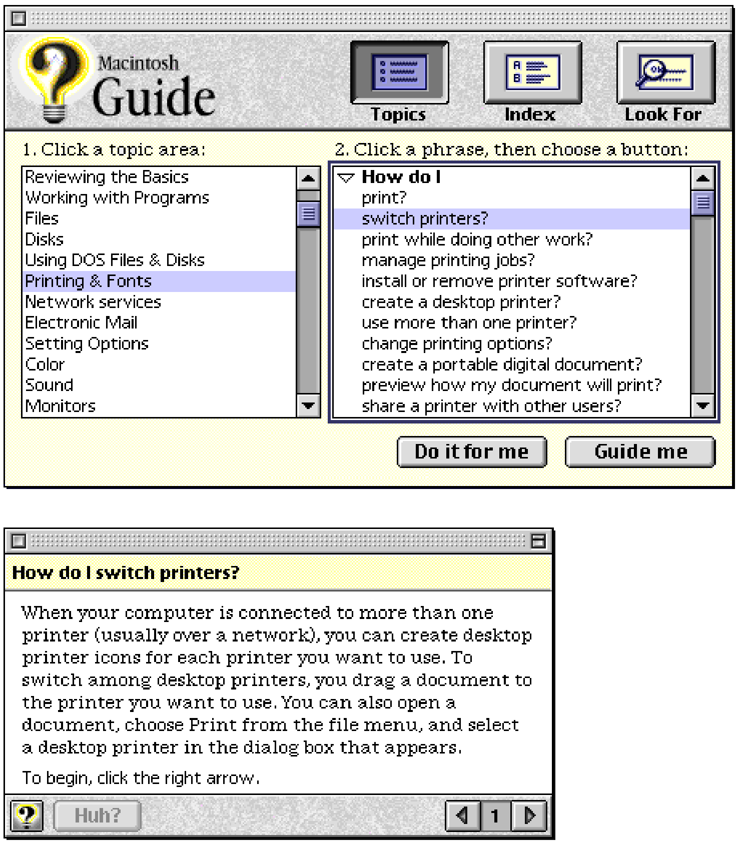


From a developer's point of view, the most important aspects of active assistance are the enhancements to Apple Guide and the mechanisms for delegating and automating tasks. These are discussed in the sections that follow.

Apple Guide Enhancements

As shown in Figure 1-7, an Apple Guide 3.0 access window presents a list of questions, problems, and tasks determined by the author of the guide file being displayed. Users can view potential topics with the aid of the Topics, Index, and Look For buttons and select the topic that best fits their needs, just as they can in earlier versions of Apple Guide.

Figure 1-7 Apple Guide 3.0 access and presentation windows



1. Click a topic area:

Reviewing the Basics
Working with Programs
Files
Disks
Using DOS Files & Disks
Printing & Fonts
Network services
Electronic Mail
Setting Options
Color
Sound
Monitors

2. Click a phrase, then choose a button:

How do I
print?
switch printers?
print while doing other work?
manage printing jobs?
install or remove printer software?
create a desktop printer?
use more than one printer?
change printing options?
create a portable digital document?
preview how my document will print?
share a printer with other users?

How do I switch printers?

When your computer is connected to more than one printer (usually over a network), you can create desktop printer icons for each printer you want to use. To switch among desktop printers, you drag a document to the printer you want to use. You can also open a document, choose Print from the file menu, and select a desktop printer in the dialog box that appears.

To begin, click the right arrow.

Apple Guide 3.0 also provides two buttons at the bottom of the Apple Guide access window that allow users to select from the types of assistance available for the selected task:

- **Guide me** initiates step-by-step instructions with coachmarking and context checking provided for each step. This is the style of instruction provided by Apple Guide in System 7.5. It is appropriate when users want to learn how to accomplish a task.
- **Do it for me** leads the user through a path that provides as much automation as possible, streamlining the steps required to complete the task. As the user moves between panels in an Apple Guide presentation window, Apple Guide automates as many steps as possible and takes the user directly to key interface elements when user input is required.

Additional enhancements to Apple Guide and to the Guide Maker authoring application include

- support for authoring interview sequences that gather the information required for assistants (introduced in the next section)
- context-sensitive filtering of topics in response to a Quick Assist key or from within Apple Guide
- support in Guide Maker for standard Copland panels, including buttons, PICT images, QuickTime movies, sounds, text input areas, pop-up menus, scrolling lists, and sliders (see "Panels", beginning on page 1-33)
- improved keyboard focus in Apple Guide windows
- support for printing the text of an individual task sequence

Delegated Tasks

Delegated tasks in Copland involve three key concepts:

- A **task** is a persistent representation of a sequence of actions that can be triggered programmatically, including actions within an application. The actions can be performed by running a script, sending a series of Apple events, executing a code fragment, or by combinations of these and other methods. A task is created from a task definition in a manner analogous to the way a document is created from an application.
- A **task definition** defines how a particular kind of task is to be performed. Combined with information about the parameters for a specific task, such as filenames or other details and a condition, a task definition can be used by an assistant or directly by a user to create a specific task. A single task definition can be used to create many tasks. You can create your own task definitions; for example, to create tasks that automatically fetch mail from an on-line service.
- An **assistant** conducts an interview to determine the user's high-level goals and, based on the user's replies, either performs one or more actions immediately or employs a task definition to create one or more tasks to be performed at a later time. You can create your own assistants, such as an assistant that uses a task definition to create a document layout.

You can provide task definitions and assistants for use either within a particular application or as stand-alone tools. For example, a spreadsheet application could include a charting assistant that helps the user create a chart quickly; and a network management assistant could be a separate product that helps administrators manage a network.

Tasks

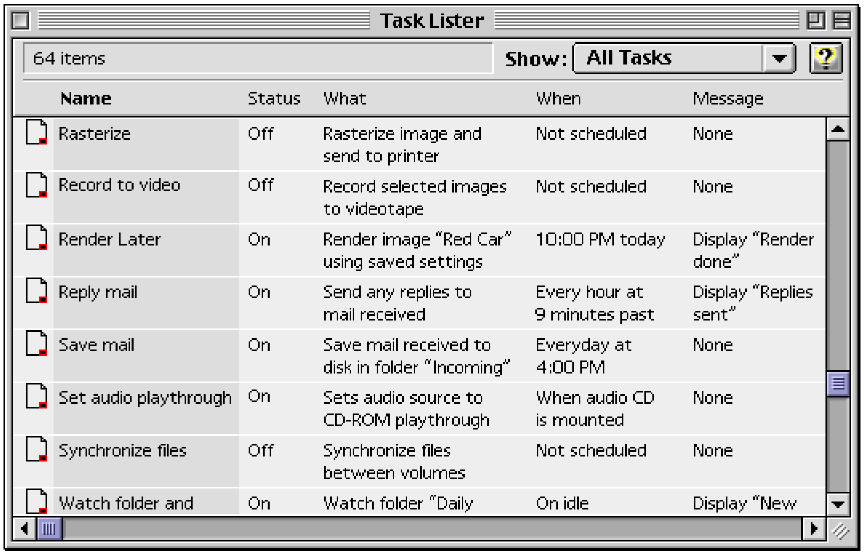
Every task includes three kinds of information:

- The **condition** for a task is the set of events or states that trigger it. Conditions can be based on specific times, time intervals, or a variety of other events or states that can be determined programmatically. You can specify custom conditions, such as a condition that triggers a task every time an e-mail application receives mail.

- The **action** for a task defines both the objects and the activities that make the task unique, such as the names of folders to back up, the name of a folder to watch on the network, or a list of people to whom a message should be sent. Tasks can perform virtually any actions that a user can perform manually.
- The **notification** for a task defines how to notify the user when the task runs and determines whether to log its execution for later review. You can create custom notifications, such as a notification that pages the user via a commercial paging system when a task is completed.

The Task Lister application (shown in Figure 1-8) allows users to view, duplicate, change, enable, disable, and remove the tasks that are currently available to the system. Task Lister also lets users modify the details of conditions, actions, and notifications for individual tasks. However, users are more likely to begin setting up tasks with the aid of assistants, which can make many detailed decisions about individual tasks on behalf of the user.

Figure 1-8 Preliminary design for the Task Lister application



Name	Status	What	When	Message
Rasterize	Off	Rasterize image and send to printer	Not scheduled	None
Record to video	Off	Record selected images to videotape	Not scheduled	None
Render Later	On	Render image “Red Car” using saved settings	10:00 PM today	Display “Render done”
Reply mail	On	Send any replies to mail received	Every hour at 9 minutes past	Display “Replies sent”
Save mail	On	Save mail received to disk in folder “Incoming”	Everyday at 4:00 PM	None
Set audio playthrough	On	Sets audio source to CD-ROM playthrough	When audio CD is mounted	None
Synchronize files	Off	Synchronize files between volumes	Not scheduled	None
Watch folder and	On	Watch folder “Daily	On idle	Display “New

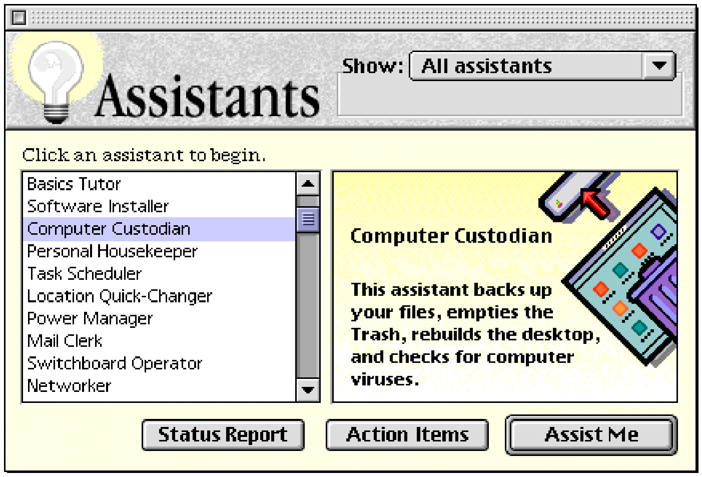
Note
The tasks listed in Figure 1-8 are preliminary examples of the kinds of tasks that may ship with Copland. They are likely to change. ◆

You can create dedicated task definitions for use by your application. The Assistance Manager manages task definitions, task files, and other aspects of active assistance. Task definitions can be registered with the Assistance Manager in two ways: as a file placed in the Assistance folder or as resources within an application that are registered via the Assistance Manager API.

Assistants

When a user chooses Assistants from the Assistance menu, a window like that shown in Figure 1-9 displays the assistants available for the current context. The user can view brief descriptions of each assistant and click the Assist Me button to initiate the one that's currently selected.

Figure 1-9 Preliminary design for the Assistants window



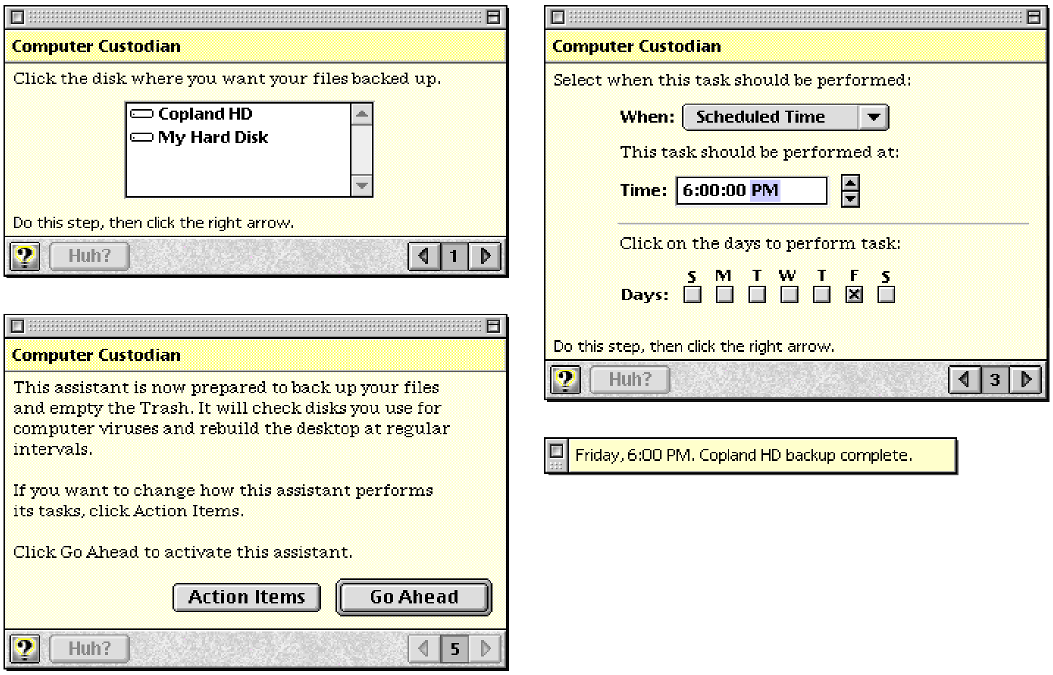
Computer Custodian

This assistant backs up your files, empties the Trash, rebuilds the desktop, and checks for computer viruses.

Note
The assistants listed in Figure 1-9 are preliminary examples of the kinds of assistants that may ship with Copland. They are likely to change. ◆

After the user clicks Assist Me, the selected assistant presents a series of windows like those shown in Figure 1-10 that let the user make high-level choices about the goals the assistant is designed to help achieve.

Figure 1-10 Excerpts from a preliminary design for an interview sequence



Computer Custodian

Click the disk where you want your files backed up.

Copland HD
My Hard Disk

Do this step, then click the right arrow.

Computer Custodian

Select when this task should be performed:

When: Scheduled Time

This task should be performed at:

Time: 6:00:00 PM

Click on the days to perform task:

Days: S ☐ M ☐ T ☐ W ☐ T ☐ F ☒ S ☐

Do this step, then click the right arrow.

Friday, 6:00 PM. Copland HD backup complete.

Computer Custodian

This assistant is now prepared to back up your files and empty the Trash. It will check disks you use for computer viruses and rebuild the desktop at regular intervals.

If you want to change how this assistant performs its tasks, click Action Items.

Click Go Ahead to activate this assistant.

The last window presented by an assistant (for example, the lower-left window in Figure 1-10) includes an Action Items button and a Go Ahead button. Some assistants don't create tasks; instead, clicking Go Ahead simply initiates the actions the assistant is designed to perform, and the Action Items button is dimmed. For assistants that create tasks, clicking Go Ahead creates and schedules one or more tasks; and clicking Action Items presents a list of those tasks as they appear in the Task Lister application, allowing the user to fine-tune the details of each task.

If requested by the user during the interview sequence, a message window like the lower-right example in Figure 1-10 appears when a task defined by the assistant runs successfully.

Assistants provide the foundation for a new **secondary human interface** that frees users from detailed decisions about how to make the computer perform specific actions. Instead, assistants make decisions on the user's behalf and use the application's **primary human interface**—its menus, windows, tools, and so on—to execute those decisions. Users still have the option of dealing directly with the primary interface, but assistants' default decisions are fine for many routine tasks.

In some respects the secondary interface provided by assistants permits more flexibility than the primary interface. For example, e-mail applications such as AppleLink often include a mechanism for automatically dialing the phone, establishing a connection, and retrieving messages, but such mechanisms require that the application already be running. When this kind of service is implemented as an assistant, the assistant can launch the application when necessary and combine mail retrieval with other tasks such as sorting mail or unpacking compressed files.

Copland provides a variety of ready-made Apple assistants to help users achieve common high-level goals. Those under consideration include assistants for installing software, sorting mail, performing housekeeping chores such as backups and virus checks, and establishing network connections. Developers can create additional assistants for a variety of purposes. This involves using Guide Maker and Apple Guide to create and present the interview sequence and the Assistance Manager to manage the scheduling and execution of tasks.

For application developers, assistants provide a way of performing complex operations without first requiring the user to learn complex details of the application's human interface. For scripters and consultants, assistants provide a way to automate specialized tasks involving multiple applications.

Information Access

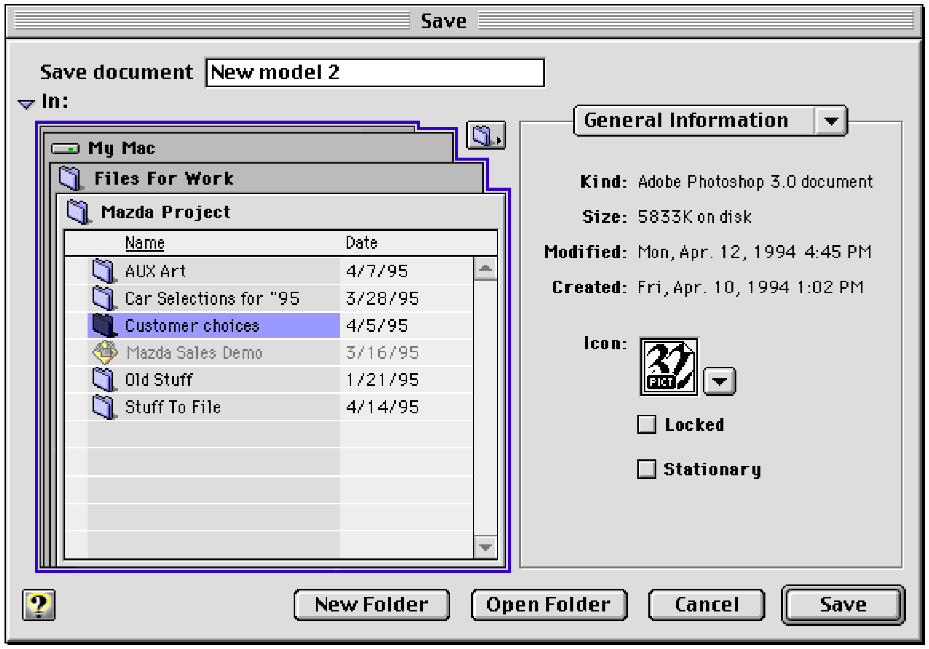
Copland includes a replacement for Standard File Package called Navigation Services. Unlike the Standard File Package, which provides limited browsing capabilities and is difficult to customize, Navigation Services provide a better browsing interface, more structured customization, and better access to document-specific information from within an application.

Navigation Services support these changes in the human interface for document management:

- The New command in the File menu for Copland applications should save a new document to disk when it first creates the document. Users can set the default location for new documents with the Default Location control panel. The first time they choose the Save command for a new untitled document, they are given an opportunity to change its name and location.
- The Save As command in the File menu should be replaced in Copland applications by the Save A Copy command, which subsumes the old Save As behavior with a more intuitive human interface.
- The Find Document command in the Copland File menu allows users to access the Copland Finder's improved Find facility from within an application.
- The Open, Save, and Save A Copy commands employ an intuitive Navigation Browser that you can easily customize. The Navigation Browser makes it easy for users to access favorite items quickly, create new folders, and perform other common tasks not supported by the Standard File Package.
- The new Document Info command in the Edit menu invokes a **document information panel,** similar to the current Get Info window in the Finder, that allows the user to manipulate document-specific information without switching contexts. The same document information panel also appears in the Open and Save dialog boxes.

Figure 1-11 shows the Navigation Browser and one of several document information panels as they appear in the Save dialog box. You can easily specify filters for the files displayed and let the user choose among available filters with a pop-up menu at the bottom of the Navigation Browser. You can also add specialized document information panels for your application's documents that permit editing and display of document information such as file type, author, keywords, colors, and dimensions.

Figure 1-11 Standard Navigation Browser and General Information panel in the Save dialog box



Save document: New model 2

Name	Date
AUX Art	4/7/95
Car Selections for '95	3/28/95
Customer choices	4/5/95
Mazda Sales Demo	3/16/95
Old Stuff	1/21/95
Stuff To File	4/14/95

Kind:	Adobe Photoshop 3.0 document
Size:	5833K on disk
Modified:	Mon, Apr. 12, 1994 4:45 PM
Created:	Fri, Apr. 10, 1994 1:02 PM
Icon:	

☐ Locked

☐ Stationary

The Navigation Services API also makes it much easier for applications to display an Open or Save dialog box with just one call, specifying options by means of parameters rather than with separate calls.

About the Copland Toolbox

Copland simplifies the Toolbox architecture by providing consistent APIs that shield you from implementation details and facilitate the use of standard human interface elements. This introduction to the changes in the Copland Toolbox begins with an overview of the architectural changes that affect all the Toolbox managers, followed by brief descriptions of some of the new managers and of specific changes to some of the existing managers.

Overview of the Copland Toolbox

The Copland Toolbox lays the foundation for future versions of the Mac OS human interface. It repairs a number of inconsistencies and weaknesses in the System 7.5 APIs and provides built-in, extensible support for a variety of features that previously required extensive coding, such as slider controls, modifier key equivalents of all kinds for menu items, and live control tracking.

The Copland architecture affects most Toolbox managers in similar ways. From a developer's point of view, the most important Toolbox-wide changes involve more consistent APIs and improved opacity with respect to underlying data structures; support for international text; extensible data structures; and ready-made, extensible designs.

Opacity and Consistency

In addition to the Toolbox APIs, Apple has in the past published descriptions of system data structures and other implementation details of the Toolbox managers. It's therefore possible to manipulate data structures directly, and many developers have done so to implement features that aren't supported by the APIs.

The fact that developers often bypass the high-level APIs has created revision locking problems for Apple. This practice also creates problems for developers and users. Custom implementations take a lot of time to develop, are inconsistent with similar implementations in other applications, and tend to work less reliably with the rest of the system than standard implementations based on the high-level APIs.

Copland addresses this problem by providing comprehensive APIs that work consistently across all Toolbox managers, including

- accessor functions for all data structures
- blind references such as `WindowRef` and `DialogRef` that replace `WindowPtr`, `DialogPtr`, and similar pointer-based data types
- high-level APIs for capabilities that are available only via low-memory accessor functions in System 7.5
- support for a much wider variety of standard and custom human interface elements

Integrated Support for International Text

Copland introduces a new systemwide text data type, called a text object, that encapsulates the details of text encoding. Text objects replace both Pascal and C strings, but they are not intended to serve as the data model for text editors. They allow applications to manipulate multilingual text transparently without dealing with the details of character encoding, which can be based on Unicode, ASCII, traditional Macintosh, and other encoding systems.

The Copland Toolbox supports text objects for all uses of strings in the Mac OS human interface, including menu item text, buttons, and window titles. This pervasive support has several ramifications:

- You can display multilingual text (in multiple scripts) in the same menu, dialog box, or other human interface element.
- Because you don’t have to keep track of the details of individual scripts and encoding systems, localization of interface elements is greatly simplified.
- Text objects are easier to maintain through several versions of a program than text related to human interface elements has been in the past.

In addition to supporting text objects throughout the Toolbox APIs, Copland provides standard human interface elements, such as left-growing windows and automatically resizable dialog boxes, that support specific international needs.

For more information about text objects, see “About Copland Imaging.”

Extensible Data Structures

As mentioned earlier, system software limitations have frequently caused developers to bypass the Toolbox APIs and manipulate data structures directly. In many cases, such strategies stem from the need to associate application-specific data with system data structures that don't allow for it.

The Copland Toolbox eliminates the need for this kind of hardwired modification of system data structures by providing a new mechanism that lets developers attach arbitrary data to virtually any human interface element. Based on the Collection Manager, which originally shipped with QuickDraw GX, this mechanism can be used to associate data with a tag and ID, attach that data to any Toolbox data structure, and retrieve it when necessary.

The Collection Manager is described in *Inside Macintosh: QuickDraw GX: Environment and Utilities.* The Copland Toolbox supports the use of tagged collection items throughout all its APIs and uses the same mechanism to associate standard properties such as colors and icons with a variety of human interface elements. You can easily extend existing collection items or add new ones to any human interface elements.

Extensible Designs

Extensible human interface designs also help to ensure the consistency and opacity of the Copland Toolbox. Copland provides two mechanisms for extending human interface designs: IDOs and panels.

The standard definition procedures, or *defprocs,* used by the System 7.5 Toolbox provide some modularity, but their programmatic interfaces are difficult to customize or enhance. Developers must often duplicate the code in system defprocs and then make small changes to achieve a desired appearance. For example, the System 7.5 Menu Manager doesn't support strike-through menu items or modifier keys other than the Command key. To display these items in menus, you must write your own `'MDEF'` resource, duplicating the code in the system `'MDEF'` and copying the current system appearance. This not only encourages inconsistent and often buggy implementations but also leads to applications that are locked into a particular interface appearance.

The Copland Toolbox provides an enhanced defproc mechanism that allows you to customize the appearance of standard objects while shielding you from the details of how they are drawn on the screen. IBM's System Object Model (SOM) provides the language-independent, object-oriented interface that makes this possible. The Copland defprocs, called **interface definition objects (IDOs),** replace the `'WDEF'`, `'CDEF'`, and `'MDEF'` defprocs used in System 7.5. The Appearance Manager provides the underlying support for IDOs, and the Window Manager, Menu Manager, and Control Manager provide standard IDOs that implement both graphic design and behavior for their respective human interface elements. You can use these standard elements in your application just by calling the appropriate manager's API.

Like System 7.5 defprocs, IDOs are essentially drawing engines that use the inheritance characteristics of SOM to simplify the creation and customization of human interface elements. They do not encapsulate data or track content in any way and do not need to be reinstantiated for every interface object. For example, the Window Manager can use single IDO to draw any number of identical windows. To store specific data in an interface element, use the Copland collection interface described in "Extensible Data Structures" on page 1-27.

The Copland Toolbox also supports lightweight panels that provide a universal, SOM-based interface for human interface elements. This mechanism is described in "Panels," beginning on page 1-33.

By supporting standard IDOs and panels, your application inherits not only the Copland human interface designs but also new human interface designs in future versions of the Mac OS. IDOs and panels also allow you to customize the standard designs or create new ones without sacrificing compatibility. The rest of this chapter introduces some of the extensible standard designs provided by individual Copland Toolbox managers.

Summary of New Toolbox Features

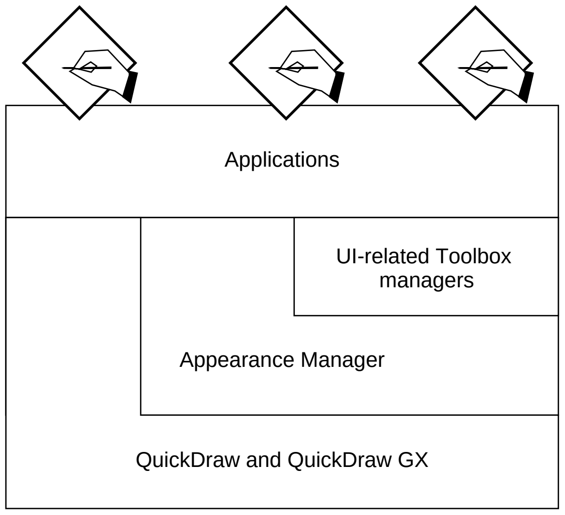
This section introduces some of the new high-level Toolbox managers and APIs and the new features supported by the Copland revisions of existing Toolbox facilities.

Appearance Manager

The Appearance Manager manages all aspects of themes and theme switching, including a set of APIs for theme developers, the Appearance control panel, support for a variety of color data (RGB colors, pixel patterns, and so on), and support for animation and sound. It also supersedes System 7.5 color tables such as `'cctb'` and `'mctb'` with a more abstract mechanism that allows you to coordinate colors with the current theme.

Figure 1-12 illustrates the relationships among QuickDraw and QuickDraw GX, the Appearance Manager, Toolbox managers such as the Window Manager and Menu Manager, and applications.

Figure 1-12 High-level architecture of the Appearance Manager

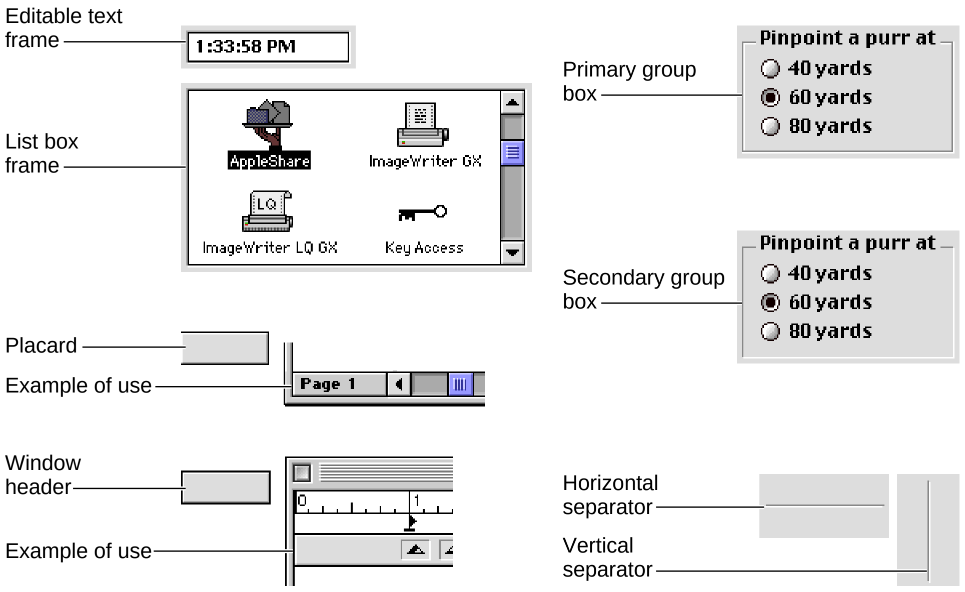


The Appearance Manager keeps track of all IDOs specified by the current theme. When your application asks the Window Manager, Menu Manager, or Control Manager to draw a particular human interface element, the manager asks the Appearance Manager for the corresponding IDO and thenceforth uses that IDO whenever it needs to draw that kind of element. When the user switches themes, the Appearance Manager informs the manager that it needs to request new IDOs.

You need to use the Appearance Manager directly only if you're creating custom human interface elements or if you want to draw in your application's content area in a manner that matches the current theme. The Appearance Manager provides drawing utilities, drawing primitives, and patterns for such purposes. For example, if you want to draw a line through a standard menu item, you can ask the Appearance Manager for the current color of the menu item text so you can use the same color for the line. Similarly, you can ask the Appearance Manager for the current background color so you can coordinate the appearance of your application's content area with the current theme.

Figure 1-13 shows preliminary designs for some of the primitives provided by the Appearance Manager as they might appear in the Apple Default theme. If none of the standard human interface elements provided by a Toolbox manager suit your purposes, you can use these primitives to assemble custom objects that the Appearance Manager draws according to the current theme.

Figure 1-13 Preliminary designs in the Apple Default theme for Appearance Manager primitives



Desktop Animation Manager

The Desktop Animation Manager allows you to write **desktop animation modules,** which can be invoked to draw the display when no keyboard or mouse events occur within a specified period of time. From the point of view of a desktop animation module, drawing to a screen-saving window is no different from drawing to the desktop.

A desktop animation module can restrict its usage to desktop mode only (that is, its only function is to maintain the appearance of the desktop background exclusive of the menu bar and Finder icons), or to screen saver mode only (that is, its only function is to act as a screen saver at the appropriate time), or it can work in both modes. When working in desktop mode, the module draws a static pattern `PixMap` specified with a `'ppat'` resource.

To help manage desktop animation modules, Copland provides a mechanism that allows the user to install multiple modules and select individual modules for different purposes. When the user configures a desktop animation module for both modes, the Desktop Animation Manager is responsible for notifying the module that it needs to switch its mode. When the user configures a different module for each mode, the Desktop Animation Manager is responsible for switching between the modules at the appropriate time.

The Desktop Animation Manager cooperates with the Appearance Manager to establish and maintain user preferences for a module. User preferences for a module are stored in a module property. When the user changes a preference for the running module via the Appearance control panel, the module receives a message that tells it to comply with the change immediately.

The Desktop Animation Manager allows users of computers with multiple monitors to select the module that runs on a particular monitor or to specify a single module that spans all monitors. Users can configure modules using the Appearance control panel.

The Desktop Animation Manager also supports screen saver demo capabilities and password protection.

Note
The term *desktop animation module* is preliminary and likely to change. ◆

Panels

The Copland Toolbox includes support for **panels,** which are lightweight objects that provide a SOM-based interface for human interface elements. Panels are entirely responsible for their own behavior and thus don't require a traditional Toolbox manager.

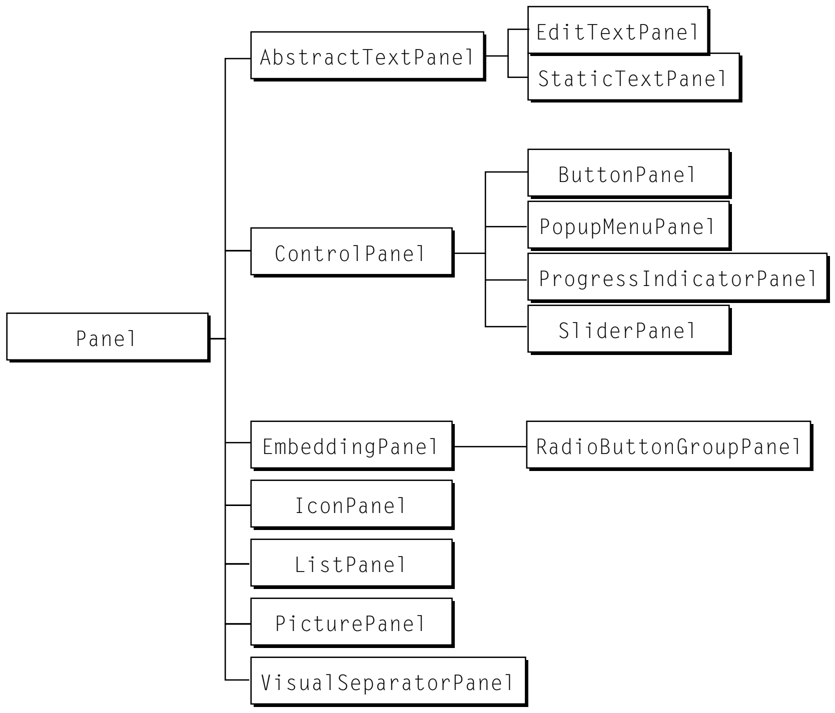
Panels support several useful features for Copland developers:

- **Object-oriented behavior.** You can use standard panel classes as is or subclass them and override just the parts you need to change for your own application.
- **Uniform programming interface.** You can use the same programming interface to communicate with any human interface element that uses the panel mechanism.
- **Binary compatibility.** As SOM objects, future versions of any panel can be released without breaking existing versions.
- **Embedding.** Panels that inherit from the `EmbeddingPanel` subclass can automatically handle keyboard navigation and other standard behavior for any other panels that they contain. This greatly simplifies the development of sophisticated panels composed of multiple subpanels. For example, radio button groups are `EmbeddingPanel` objects that contain multiple embedded panels, all coordinated and maintained as a single unit.
- **Keyboard navigation.** Panels know how to react to keystrokes and how to display themselves with and without keyboard focus.
- **Collection items.** Clients can attach arbitrary collection items to panels.
- **Drag Manager support.** You can support drag and drop for any panel just by overriding a few methods.

In addition to supporting the general panel mechanism, the Copland Toolbox includes a set of concrete panel classes that developers can instantiate and use as is. Developers, scripters, and consultants can also use standard object-oriented programming (OOP) techniques to develop custom panels for their own use or to sell to customers or other developers.

Figure 1-14 shows the class hierarchy for the standard Copland panels. These panels can be used with the corresponding human interface elements to create self-contained human interface elements that can be easily integrated and maintained. You can use multiple inheritance techniques to subclass custom panels for more specialized purposes.

Figure 1-14 Class hierarchy for standard panels



Although panels can be used within OpenDoc parts, they aren't intended to be as large or as powerful as parts. Instead, they facilitate the assembly of integrated human interface elements from smaller, simpler objects.

The Copland Dialog Manager has been rewritten to take advantage of panels. However, panels are independent of the Dialog Manager and can be used anywhere within an application. For example, you can place panels in any window, not just in a dialog box, and combine them to create toolbars and other custom elements.

The Dialog Manager provides a convenient API for managing panels in a dialog box. If you use panels directly, for example in a document window, you must handle events yourself, keep track of the panels' locations, and perform other housekeeping tasks.

Menu Manager

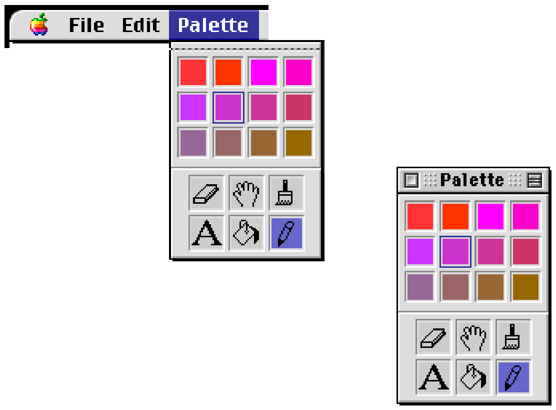
The Copland Menu Manager allows you to

- make any application menu or submenu a tear-off menu
- provide custom content on an item-by-item basis rather than for an entire menu at a time
- choose from a rich set of standard menu item types and menu templates, many of which aren't supported in System 7.5
- use collection items to add nonstandard menu types to any menu item
- create custom grid menus containing colors, icons, and so on
- use text objects to display any font in any language using any script in a single menu item
- display any modifier-key equivalent for any menu item
- show and hide the menu bar

All Copland menus automatically support a "sticky menu" mode that allows users to leave a menu or submenu open and choose menu items by clicking them or from the keyboard.

Figure 1-15 shows a tear-off menu created with standard types and templates. Standard types include pattern and color swatches, text items based on text objects, and dividers. Standard templates support menu sections and grid menus. It's also possible to create custom types and templates.

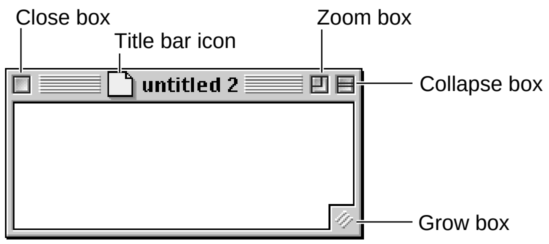
Figure 1-15 Preliminary design for a tear-off menu created with standard Menu Manager types and templates



Window Manager

The Copland Window Manager's standard definition for a document window produces a window like that shown in Figure 1-16 when the Apple Default theme is selected.

Figure 1-16 Preliminary design for an active document window in the Apple Default theme



The window in Figure 1-16 includes

- A title bar icon that users can drag to a new volume to copy the document to that volume. The behavior of the title bar icon is determined by you within guidelines to be provided by Apple. The Window Manager determines its appearance only.
- A collapse box in the upper-right corner that integrates the WindowShade behavior provided by System 7.5. You can also choose to replace the collapse box with a gadget icon that implements application-specific behavior.
- A zoom box next to the collapse box. The Copland Window Manager includes built-in support for Finder-style zooming with multiple monitors.

The Window Manager also supports left-growable windows and multilingual text in window titles (based on text objects), both of which greatly simplify globalization.

Window content regions remain in the application domain. However, you can use a pattern supplied by the Appearance Manager to define the content region's background appearance according to the current theme.

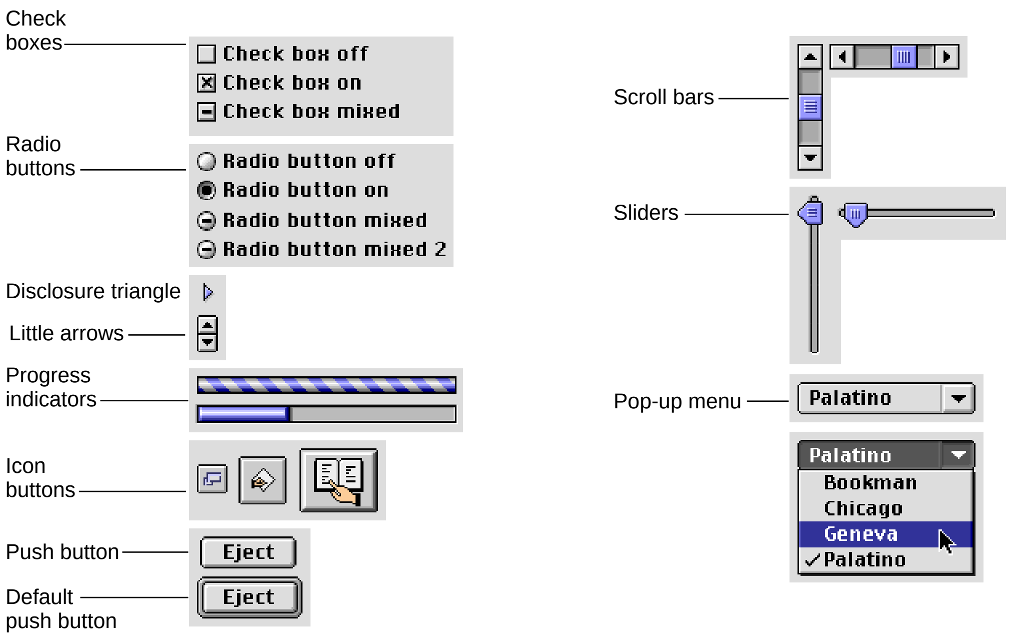
Other changes to the Window Manager that affect developers include the following:

- **Layer management.** The Copland Window Manager takes care of all layer management automatically. To support this change, all windows are classified as modal, floating, or normal.
- **New APIs for floating and modal windows.** The Copland Window Manager takes care of tracking floating and modal windows, which are much easier to implement and customize than in System 7.5.
- **The window list.** Copland Window Manager APIs that move windows in the window list support floating windows. Essentially they work as they always have except that they move a window only within the part of the window list that contains windows of the same type.
- **Modal state.** The Copland Window Manager takes responsibility for determining when to call the Menu Manager to set the menu bar to a modal state.

Control Manager

Figure 1-17 shows some of the standard controls provided by the Copland Control Manager as they appear in the Apple Default theme when they are active. These standard controls are more numerous and easier to create than the standard controls provided in System 7.5.

Figure 1-17 Preliminary designs for some standard controls as they appear in the Apple Default theme



Note that the radio buttons and check boxes in Figure 1-17 support a mixed state between off and on.

The standard Copland controls simplify many tasks that previously required custom coding:

- Pop-up menus are much easier to implement and can be torn off and navigated from the keyboard.
- Live tracking is much easier to support because you can get a series of control values back from a control while a user is still manipulating it. For example, you can get control values back from a scroll bar that allow your application to redraw the window's contents while the user is dragging the scroll box (live scrolling), or you can change the sound volume while the user is still manipulating the slider rather than waiting until the user releases it.
- Control values are 32-bit values, thus avoiding the need to scale the values used with large drawing areas.

All standard controls support keyboard focus, the use of text objects for international text, and collection items. The Copland Control Manager also makes it easier to create custom controls.

Dialog Manager

The Copland Dialog Manager includes a suite of standard elements, such as radio button groups, visual separators, and lists, that make it much easier to construct complex dialog boxes. The API has changed little compared to System 7.5, but the underlying code has been rewritten as an interface layer for manipulating panels and groups of panels. Although the filter functions used with earlier versions of the Dialog Manager are still supported for compatibility with System 7.5 code, filter functions have been superseded by panels.

The elements provided by the Dialog Manager support tracking of keyboard focus, management of groups of radio buttons, and other standard behavior. To implement specialized behavior, you subclass from the standard elements and override the appropriate SOM methods using OOP techniques.

Panels are completely independent of the Dialog Manager and can be used anywhere within an application, not just in dialog boxes. When you use the Dialog Manager to manage panels, it takes care of all event handling. When you manage panels directly, you must take care of event handling yourself. For more information, see "Panels," beginning on page 1-33.

Scrap Manager, Clipboard Manager, and Drag Manager

The Scrap Manager used in System 7.5 has changed little since it was first created as part of the software for the original Macintosh computer. It was originally designed to handle a few lines of text or a 1-bit picture being copied and pasted between MacPaint and MacWrite, not the large pieces of data, such as QuickTime movies, sounds, and blocks of formatted text, that are common today.

Copland replaces the original Scrap Manager with a new Clipboard Manager and introduces an entirely new Scrap Manager that provides a generic transport mechanism used by both the Clipboard Manager and the Drag Manager. The Copland Scrap Manager permits the transport of data of any size. It also supports collection items at three levels: the scrap as a whole, individual items within the scrap, and the data within each scrap item. The Clipboard Manager and the Drag Manager control access to their respective scraps, while the Scrap Manager always controls reading from and writing to the scrap.

There is only one Clipboard scrap, and you use the Clipboard Manager to lock it. You then use the Scrap Manager to read and write information to the Clipboard scrap and unlock the scrap when you're through.

The Clipboard Manager also supports the concept of *promises*. If the data copied or cut is large, you can choose to post a promise to the Clipboard scrap instead of the data itself. When the user pastes, the Scrap Manager uses the promise to retrieve the data from the application that did the copying. This mechanism avoids data transfer until it is actually needed for a paste. It also allows the original application to transfer the data using just the format requested by the pasting application rather than duplicating the data in a variety of possible formats.

Only one application can access the Clipboard scrap at a time. Because the contents of this single Clipboard scrap is in global memory, not application memory, you don't need to worry about the Clipboard suddenly taking over all your free memory. This arrangement takes care of many common copy and paste problems and also supports background copying, for example by a script that's running in the background.

Because the Copland Clipboard Manager is closely modeled on the Copland Drag Manager and both managers use the generic Copland Scrap Manager, you can use the same source code to implement both copy and paste and drag and drop. From the Scrap Manager’s point of view, references to the Clipboard scrap and the Drag scrap are identical.

The Clipboard Manager takes care of synchronizing the behavior of applications that use the System 7.5 Scrap Manager and Copland-savvy applications that use the new mechanism, although the former won’t be able to take advantage of the Copland’s improved performance and consistency.

Most of the System 7.5 Drag Manager API is no longer needed in Copland, although it is still supported for applications that aren’t Copland-savvy. Like the Copland Clipboard Manager, the Copland Drag Manager uses the Scrap Manager API for all data transport. The Copland Drag Manager also displays translucent icons during dragging rather than simple outlines.

Icon Utilities

The Copland Icon Utilities provide an extended API that allows you to obtain the icon currently displayed in the Finder for a given file much more easily than you can in System 7.5. Instead of having to examine a variety of possible locations for standard application icons, custom icons, edition file icons, alias icons, and so on, you just specify the file whose icon you want and the Icon Utilities return the appropriate icon. Because the Copland Finder uses the same mechanism to manage icons, you can be certain that the icon returned is the same one displayed in the Finder.

Icon Utilities also provide systemwide icon caching, copying each icon into the cache as it is requested. If the icon you request happens to be in the cache already, it doesn’t have to be read into memory again. This approach avoids reading duplicate icons into memory and thus improves performance.

C H A P T E R 2

About Copland Imaging

Contents

This chapter introduces Copland imaging, including graphics and text. Copland streamlines the Mac OS graphics architecture by integrating all the Macintosh graphics models. Copland offers a single set of graphics APIs that share underlying code for increased system efficiency. Because the APIs and code base for QuickDraw GX graphics and text and for QuickDraw 3D graphics are included in Copland graphics, any development you do today using QuickDraw GX and QuickDraw 3D will run without modification on Copland.

Compatibility—Backward and Forward

The Copland graphics architecture uses the same base of graphics code to implement QuickDraw, QuickDraw GX, and QuickDraw 3D. All of the functions in all three APIs remain available. Thus, all of your graphics code runs without modification on Copland while its performance improves due to the increased efficiency of the new unified code base. This integration of graphics models will continue on versions of the Mac OS beyond Copland, so your graphics routines will continue to run and their performance will continue to improve.

Although Copland continues to support the Printing Manager APIs for backward compatibility, all native printing on Copland uses the QuickDraw GX print model. In order to get the best performance on Copland and to ensure that your print routines continue to work on versions of the Mac OS beyond Copland, you should use QuickDraw GX for all your print routines now and in the future.

You can continue to use your C and Pascal-string text routines on Copland. However, by switching to the new text-object model, you can include encoding information with the text, alleviating the problem of having to store information about text encoding (that is, the script system) and text content separately. The new text model makes it much easier to support international, multilingual text. In addition, if you switch to the new text-object model now, the transition to some new text encoding method in the future will be transparent to your text-handling routines, requiring no recoding or recompiling when the change occurs. In the future, text objects might contain additional information, such as language and text-to-speech hints.

Design Goals for Copland Imaging

The design goals for Copland graphics include graphics system integration, the implementation of text objects, and improved handling of fonts.

In System 7.5, you choose from among several graphics models when writing your application:

- QuickDraw for rendering onscreen graphics, drawing text, and printing documents
- Color QuickDraw for rendering color graphics
- QuickDraw GX for rendering onscreen graphics, drawing text, and printing documents
- QuickDraw 3D for defining and rendering three dimensional graphics
- WorldScript for managing text and font subsystems

The APIs for these existing graphics models are maintained in Copland. In fact, they become standard, simplifying the implementation of these various graphics models in system software. What's more, Copland combines the code from these graphics systems wherever possible, thus reducing redundancy, increasing efficiency, and decreasing the size of the operating system. Therefore, Copland provides a single print driver model, a single graphics code base, and a single font cache. In addition, Copland provides a new text model, based on text objects rather than Pascal or C strings. The new text model greatly improves text encoding, making internationalization much easier.

Graphics Systems Integration

The various graphics systems available in System 7.5 each include code to perform certain basic functions, such as displaying data on the screen and caching fonts. Wherever the graphics code was redundant, Copland combines the code so that all of the graphics systems use a common code base. At the same time, Copland uses native code for the features that make each graphics system unique. Each graphics system runs as efficiently as possible using native code; none are emulated.

Copland's integration of graphics implementations offers several advantages to your application.

- A single, ever-available set of APIs eliminates the need for redundant code paths in your application.
- A shared graphics code base providing services to QuickDraw, QuickDraw GX, and QuickDraw 3D reduces system memory requirements and improves performance.
- A single print driver model (the QuickDraw GX model) allows the Finder to present your users with a consistent metaphor for printing documents.
- A unified font rendering and caching mechanism improves overall system performance and minimizes RAM requirements. Furthermore, it allows any font to be used with any API.
- Integration of WorldScript I and II and the Language Kit Extensions into the operating system simplifies international application development and provides better international text support.
- An improved version of TextEdit offers all users, regardless of their script systems, improvements in systemwide text manipulation.
- The powerful QuickDraw GX line layout technology is now part of the standard system software installed on all users' systems.

Using Copland's unified graphics system in your application requires negligible work on your part. Aimed at increasing performance—especially in the areas of hardware acceleration and RAM resource allocation—most of Copland's improvements to the font cache and the graphics code base are at a level below the public API. With Copland's desktop printer icons, the single (QuickDraw GX) print driver model improves your user's experience in the Finder. In addition, the QuickDraw GX–style print dialog boxes, which are standard with Copland, improve the user's experience with your application.

Text Objects

Although System 7.5 provides extensive support for international text, the information about the encoding (that is, the script system) of the text must be maintained separately from the text content. Programmers must introduce additional complexity in their code to keep track of this separate data so that the text is processed correctly by the system.

Apple's goal is to shift the Mac OS from the current multiscript, multiple-encoding model to one in which all scripts are handled in a consistent fashion. To facilitate this transition, Copland introduces a new systemwide text data type, called a text object, that hides the details of text encoding from the programmer.

A **text object** is an opaque data structure that contains information about both text content and text encoding. Supplanting both Pascal and C strings, text objects become the fundamental text data type in Copland. The use of text objects greatly simplifies the handling of text encoding and removes most limitations of both Pascal and C strings.

Text objects extend the capabilities of the basic string data type by allowing arbitrary annotations to be made to the text. For example, in the future, text objects might be annotated with pronunciation hints for text-to-speech conversion. Text objects use a Unicode converter extension to manage conversions between different text encodings. (Unicode is a fixed-width 16-bit character-encoding system—part of the international standard ISO 10646—that provides a code for every character in every major writing system.) The details of these conversions will be handled by the text object, so most programmers do not have to deal with the Unicode converter API directly.

Text objects allow applications and data to use multiple text encodings, including Unicode and WorldScript.

Text objects are not intended to serve as the data model for text editors. Rather, they allow applications to manipulate multilingual text transparently without dealing with the details of character encoding, which can be based on Unicode, ASCII, traditional Macintosh, or other encoding systems.

Font Handling

Copland introduces an open font architecture that allows developers to write their own font scalers. Therefore, a font provider can provide a scaler that makes a new font architecture available in a way that is completely transparent to users. From the user's point of view, all font types, whether Type 1, TrueType, bitmapped, or some new type, are installed and used in exactly the same way.

On Copland, font scalers run in their own memory partitions and are therefore protected from application errors.

Preparing Your Application for Copland

On Copland, every user has QuickDraw GX and QuickDraw 3D as part of their basic system. Therefore, you should start using these powerful graphics models now to make their features available to your users.

Because all native printing on Copland uses the QuickDraw GX print model, and because later versions of the Mac OS will not support the Printing Manager APIs, you should use QuickDraw GX for all your print routines now and in the future.

The Copland text-object model makes it much easier for you to internationalize your application and data. You should switch to the new text-object model now, not only to take advantage of its ease of use, but also because as new text encoding methods become available in the future, your text routines will automatically benefit from the change without additional effort on your part.

C H A P T E R 3

About Copland Processes

Contents

The Copland **microkernel** is the part of the operating system that manages processes. A **process** consists of a set of one or more **tasks**—which are the basic units of program execution in Copland—and the memory and other operating system resources allocated to them by the microkernel. To share the processor among all tasks, the microkernel can preempt the execution of one task and start—or resume—the execution of another. (This form of processor sharing is called **preemptive multitasking**.)

This chapter describes task and process management in Copland. You should read this chapter if you are writing any kind of software product for Copland. This chapter will help you understand and efficiently use Copland's task and process management features.

There are two types of processes in Copland:

- cooperative processes
- server processes

The **Process Manager** is the part of Copland that launches, manages the cooperative scheduling of, and terminates **cooperative processes.** They are called *cooperative* because their tasks cooperate to share processor time. All Mac OS applications that relinquish control of the processor by calling the `WaitNextEvent`, `GetNextEvent`, or `EventAvail` function run as cooperative processes. From all cooperative processes, the Process Manager allows only one task at a time to have access to the processor. The microkernel, in turn, preemptively schedules this task for execution.

Note
Because System 7 supports only one kind of process, the cooperative process, the *Inside Macintosh* suite of books simply refers to cooperative processes as *processes*. ◆

The **Server Manager**, new in Copland, launches and terminates server processes. In Copland, a **server process** is software that provides a service to other processes on the same computer or a connected computer. The microkernel can preempt the execution of a task within a server process without the task calling a function like `WaitNextEvent` to yield control.

System 7 Application Compatibility

Copland fully supports the Process Manager and Thread Manager APIs from System 7.5. Copland also provides full compatibility with the System 7 methods of interprocess communication.

Design Goals for Copland Process Management

The design goals for Copland process management are to

- increase the responsiveness and performance of Macintosh applications
- increase system robustness

Copland increases application responsiveness and performance by providing a preemptive multitasking environment rather than the System 7 cooperative multitasking environment.

Copland increases system robustness by restricting applications and completion routines from writing over critical system data. Copland also protects code from corruption by preventing any software from writing over it.

Processes

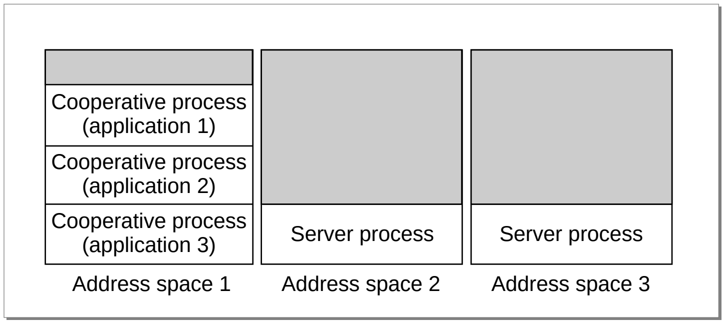
Copland uses processes to track resource allocation and reclamation. When a process is destroyed—for example, when an application exits or terminates abnormally—the microkernel reclaims all associated operating system resources, including stacks and underlying system data structures. (In this chapter, use of the term *resource* includes all physical and abstract resources, such as memory, disk space, tasks, and timers; it is not limited to Resource Manager resources.)

Each process has a set of memory locations and associated values, collectively called an **address space.** In Copland, all cooperative processes share a single address space, and therefore are not inherently protected from each other.

Every server process, on the other hand, runs in its own address space, thereby protecting it from applications and other server processes (and, conversely, protecting applications and other servers from it). Figure 3-1 illustrates how all cooperative processes reside in one address space, and how each server process has its own address spaces. (While processes in one address space have no direct access to any other address spaces, all address spaces include globally shared code and data, as described later.)

Note
In future versions of the Mac OS beyond Copland, every application will have its own address space as well. ◆

Figure 3-1 Copland processes and their address spaces


Cooperative process (application 1)
Cooperative process (application 2)
Cooperative process (application 3)
Server process
Server process
Address space 1
Address space 2
Address space 3

The microkernel deletes a process when all tasks within the process have terminated. After it deletes the process, the microkernel reclaims all associated operating system resources.

Cooperative Processes

Like a System 7 process, a cooperative process in Copland is an instance of an application being executed. When the Process Manager launches an application, Copland

1. creates a new cooperative process
2. loads the application into the new process

3. creates an application heap and stack
4. opens the application resource fork
5. creates a task, the *primary task*, within the process; this task runs the application's main routine, which can create additional tasks called *secondary tasks*

Cooperative processes present a human interface by using Toolbox managers, such as the Window Manager and Control Manager. To maintain compatibility with System 7 applications, the Toolbox managers are not reentrant. (A **reentrant service** can be safely called by several tasks at the same time, but a non-reentrant service like the Window Manager can be safely called by only one task at a time.)

Because only one task should use a Toolbox manager at time, the Copland Process Manager ensures that from among all cooperative processes, only one primary task is eligible to run at any one time. By rotating this eligibility among all primary tasks, the Process Manager gives each primary task the opportunity to be preemptively scheduled by the microkernel.

As in System 7, an application must yield control of the CPU to other cooperative processes at frequent intervals by calling `WaitNextEvent`, `EventAvail`, or `GetNextEvent`.

Cooperative processes can terminate either normally or abnormally. Cooperative process terminate normally when the following occurs:

1. The application performs the usual cleanup done in System 7, such as checking for outstanding I/O requests, removing VBL tasks, removing any remaining timers, and closing any network connections.
2. The application terminates any of its secondary tasks.
3. The primary task calls the `ExitToShell` function.

If your application does not clean up properly, the system won't crash as it can in System 7. Instead, because the Copland microkernel tracks all resources (such as, tasks, timers, and areas), it can reclaim them when it terminates the process associated with your application.

When terminated abnormally, the primary task never calls the `ExitToShell` function. For example, if a process does not properly handle an exception (that is, an access fault or some other special error detected by the microprocessor), the process terminates without calling `ExitToShell`. However, even when a process terminates abnormally, the microkernel terminates all tasks within the process and reclaims all associated resources.

Server Processes

In Copland, a server process is software that provides a service to other processes on the same computer or a connected computer. Copland uses many server processes, such as the File Share server and Events server. Copland also provides an API that allows you to create server processes of your own.

Server processes run in their own address spaces, separate from cooperative processes and other server processes. This protection makes them more reliable service providers. For example, the File Share Server provided by Copland can continue to provide file access to remote users even when an application has crashed on the computer that is serving the files.

Server processes should never use the non-reentrant services of the Toolbox managers. Therefore, server processes should not perform any human interface operations. Server processes may, however, interact with the user indirectly by communicating with cooperative processes, which do perform human interface operations. For example, a server process can communicate with a cooperative process through Apple events.

Server processes may also perform minimal interaction with the user through the Notification Manager. For example, a mail server can use the Notification Manager to notify the user of incoming mail. The user must then use a mail application to actually read and respond to mail. In System 7, you can provide this service through a combination of an application (for the user interface) and a system extension of type `'INIT'` (for the background-only application). In Copland, you provide this service by using a cooperative process and a server process.

The Copland Server Manager is the part of the Mac OS that launches and terminates server processes. The Copland microkernel starts the Server Manager when all other system initialization tasks are complete. After it is started, the Server Manager launches all server processes associated with a workspace. Workspaces, described in the chapter "About the Copland Human Interface and Toolbox," allow multiple users to customize their work environment on a single Macintosh.

When the Server Manager launches a server process, the Copland microkernel creates the following items that are associated with that process:

- an address space
- a task within the process
- a stack for use by the task

Tasks

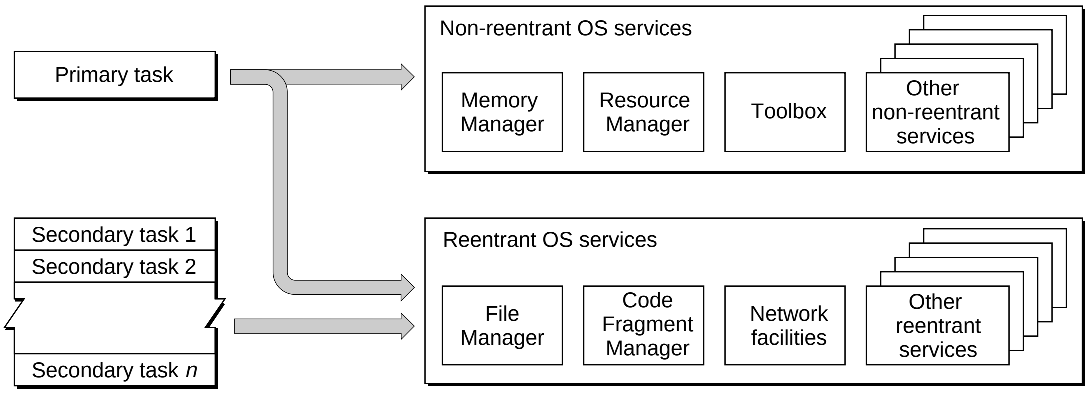
Processes are passive—that is, they do not execute instructions. Tasks execute instructions. Each task has a stack and a set of registers. A task can share an address space with many other tasks. The microkernel provides additional processing resources to a task. These resources include general-purpose registers (such as R0 and FP0) and special-purpose registers (such as CR and FPSCR).

Tasks belong to processes. A **primary task** corresponds to the execution of a cooperative process. There is one primary task for each instance of an application being executed. (Server processes do not have primary tasks.) Only primary tasks should call `WaitNextEvent` and such non-reentrant services as the Toolbox managers, the Memory Manager, and the Resource Manager.

Your application or server software can create one or more **secondary tasks.** They should use only a restricted set of operating system services—those that are reentrant, including all microkernel services (such as synchronization and messaging), the File Manager, the Code Fragment Manager, the Server Manager, the Notification Manager, the Apple Event Manager, device drivers, and network facilities. In particular, secondary tasks should not use services that deal with the human interface or that are graphical, such as QuickDraw and the Window Manager.

Figure 3-2 shows the services that primary and secondary tasks should access.

Figure 3-2 Operating system services accessible to primary and secondary tasks



You can improve the responsiveness and productivity of your application by having the primary task perform user interface tasks and using secondary tasks to process data and perform time-consuming I/O and compute-intensive operations.

Because Copland runs many secondary tasks in addition to the set of primary tasks associated with cooperative processes, it is crucial that primary tasks avoid using “idle-time” scheduling models. If your application must perform critical, computationally intensive execution, you should use a secondary task, instead of having a primary task call `WaitNextEvent` with a timeout of zero.

Cooperative Threads and Preemptive Tasks

In Copland, there are two ways for you to use multiple paths of execution in your software: you can use the preemptively scheduled tasks previously described in this chapter, or you can use the cooperatively scheduled **threads** first offered by the System 7.5 Thread Manager.

A task containing threads may call Toolbox managers from any of its threads, because the Thread Manager, using the Process Manager, cooperatively schedules access to the non-reentrant services of the Toolbox. This allows both primary and secondary tasks to create threads.

Note that while thread execution—under the control of the Thread Manager—is scheduled within a task, task scheduling is made systemwide by the microkernel. Task scheduling increases system throughput and allows a fine degree of parallelism not available with threads. Secondary tasks are more suitable than threads for performing I/O that is synchronous to the execution of your application.

Task Scheduling

Copland's microkernel schedules all tasks preemptively, based on their priority and on other microkernel multitasking rules. A task is eligible for execution whenever it is not waiting for some operation to complete, such as a synchronous I/O operation or a page fault. Many tasks can be eligible for execution, but only one can be executing at a time.

Under Copland, the highest priority task that is eligible for execution is guaranteed to be the task that is executing. A task's **priority** is based on its relative importance. The Process Manager assigns the same priority to all primary tasks. The code creating a secondary task assigns that task's priority. The Copland microkernel has varying priorities for servers, applications, drivers, and real-time operations.

Tasks that are not eligible for execution are said to be *blocked* on some event. (For example, the Process Manager blocks all primary tasks but one. This ensures that only one primary task is eligible for access to the Toolbox managers.)

When a task is blocked on some event and that event occurs, the task becomes eligible for execution again. If that task has a priority greater than the currently executing task, the microkernel performs a **task switch,** where the execution of one task is suspended while the execution of a different task is resumed from the point at which it was blocked. A task switch saves the processor state of the former task and restores the processor state of the latter task.

The microkernel performs a task switch when a task with a priority greater than the currently executing task becomes eligible for execution or when the currently executing task becomes blocked for some reason.

If several tasks have the highest priority and are all eligible for execution, the microkernel allows each task to execute for an internally specified time called a **time slice.** When a time slice expires, the microkernel switches to the next task with the same priority. The microkernel uses this time-slice form of scheduling to give each task at this highest priority access to the CPU on a first-in, first-out (FIFO) basis. No single task can starve the others unless it is the only task at the highest priority.

The microkernel never uses time-slicing over its priority-based scheduling algorithms; it uses time-slicing only when several tasks are all eligible for execution at the same priority and no higher-priority tasks are eligible. If a higher-priority task becomes eligible for execution, it will always get immediate access to the CPU.

Task Communication

Tasks cannot move between processes. However, tasks can communicate with each other, even when the tasks belong to different processes. Copland supports communication between tasks in both cooperative and server processes, and Copland provides full compatibility with the System 7 methods of interprocess communication.

Copland provides a layered model for task communication, from high-level communication services to low-level synchronization services. Depending on your software's needs, you can use any of the following:

- **Apple events.** In Copland you can use Apple events to communicate between all tasks in the system. This means that Apple events can be used for communication between cooperative and server processes, between cooperative processes, between server processes, as well as between tasks within a process. In System 7, only applications (or processes that call `WaitNextEvent`) can use Apple events. Copland extends these services to secondary tasks. In Copland, the Apple Event Manager provides secondary tasks with the ability to send and receive Apple events, and secondary tasks can take full advantage of the Apple Event Manager.
- **High-level events.** Copland supports high-level events, but secondary tasks cannot send high-level events other than Apple events. In most cases you should use Apple events to communicate with other applications.
- **Events.** Copland fully supports the System 7 Event Manager APIs. By calling `WaitNextEvent`, an application's primary task receives notice of such events as update events, mouse events, and keyboard events.
- **PPC Toolbox services.** Primary tasks can use the PPCBrowser mechanism and the PPC Toolbox functions. Secondary tasks cannot use the PPCBrowser

mechanism but can call all other PPC Toolbox functions. However, before using the PPC Toolbox to send data between tasks, you should consider whether any of the other interprocess communication methods are more appropriate for your needs.

- **Microkernel messaging services.** The messaging services can be used by tasks to communicate with each other. If two tasks communicate using the messaging services, they must agree on the protocol and conventions of the information they exchange. The microkernel does not interpret the data, it simply provides a mechanism for transporting the data from one task to another. Both primary and secondary tasks can use the messaging services. However, in most cases a primary task should use Apple events rather than the messaging services. (Apple events provide for rich data content and network capability, and thus are more suitable for communication between applications.)

  The messaging services are well-suited for use by server processes that provide shared libraries or that provide other communication services. For example, a client might use the API provided in a shared library; the actual implementation of the API may use the messaging services to transport data between the client and a server. In this case, the use of the messaging services is transparent to the client.

- **Task synchronization.** The microkernel provides various services that allow you to synchronize access to shared resources by multiple tasks. These services are described in the next section.

Task Synchronization

Because the Copland microkernel schedules tasks preemptively, it also supplies services you can use to synchronize their execution to protect and control access to shared resources.

Most applications don't need to deal with synchronization in System 7, where the current application can always assume that—except for code running at interrupt and deferred task level—the application controls the computer. Even when your application code needs to synchronize with code running at interrupt level, such as during VBLs or completion routines, it can do so by spin-looping on a global variable or by disabling interrupts.

Neither of these mechanisms provides advantages over Copland's preemptively multitasking microkernel. If your application takes advantage of Copland's microkernel services, such as preemptive tasks and messaging, you need to be aware of synchronization issues and know how to protect your code from synchronization errors.

Copland offers new APIs that provide the following synchronization services:

- **Atomic operations.** Tasks can use atomic operations to modify a single, 32-bit aligned word of memory in any way without interference from other tasks and interrupts. Atomic operations are simple subroutines that use the processor's built-in synchronization mechanisms; examples include operations that increment, decrement, test and set, and compare and swap.
- **Read/write locks.** Read / write locks protect resources that your application can share some of the time and lock exclusively only some of the time. This type of lock can have one writer and any number of readers.
- **Simple locks.** Simple locks, a subset of read / write locks, ensure that an area of memory or some other resource is accessed exclusively by one task.
- **Event groups.** An event group consists of a set of 32 flag bits. Used for synchronizing operations among tasks, these flags are similar to semaphores, which have traditionally been available on other multitasking systems.
- **Kernel queues.** Similar in use to System 7 deferred tasks, kernel queues are first-in, first-out queues of three-word entries. Your application can use event queues as simple messaging mechanisms or as the lowest-level synchronization mechanisms for indicating completion of asynchronous events.
- **Secondary interrupts.** Device drivers can use secondary interrupts, available only in supervisor mode, as a sychronization mechanism because they are always serialized. A secondary interrupt can be thought of as non-reentrant code shared across the entire system.
- **Disabling hardware interrupts.** Disabling hardware interrupts is one of the few synchronization options available in System 7. This method is not recommended in Copland and should be used only in device drivers when no other mechanism works. In Copland, your application or server run in user mode and therefore cannot disable hardware interrupts.

Note
For more information about task communication and synchronization, see *Microkernel White Paper* included on this disk. ◆

Copland Execution Environments

Copland explicitly defines execution environments to minimize interrupt latency, to maximize responsiveness, and to allow greater I/O throughput. An execution environment refers to

- the circumstances under which code executing in a given environment is invoked
- the routines that can be called
- the type of memory access that is permitted

Copland supports four execution environments:

- task level
- software interrupt level
- secondary interrupt level
- hardware interrupt level

Each execution environment is partly characterized by the processor mode, either supervisor or user.

Supervisor mode is the state of operation for the processor that allows software to gain access to all memory, processor registers, and other critical resources. When software executes while the processor is in supervisor mode, the software can write to, and therefore corrupt, any address space.

IMPORTANT
Code running in supervisor mode has no access to the Toolbox or to A-trap emulation; such code has access to reentrant services only. ◆

User mode is the state of operation for the processor that allows software, typically application software, to execute in an environment that protects certain critical resources, such as portions of memory and certain processor registers. When the processor is in user mode, software can write only to its own address space.

Task Level

All tasks (primary and secondary tasks in either supervisor mode or user mode) run at task level. Nearly all code is executed at task level—application code, the Copland microkernel, and device drivers.

The processor executes task-level code whenever it is not executing at one of the interrupt levels. The microkernel suspends task-level execution while any hardware or secondary interrupt handlers execute. Then, the microkernel schedules tasks preemptively according to their priority and internal rules for time-slicing. When a task is eligible to continue executing, its normal processing can be interrupted to run any software interrupt handlers sent to it.

Code executing at task level can call nearly all microkernel, OS, and Toolbox services and it is allowed access to pageable memory.

To protect the rest of the system, you should design your software so that it runs as much as possible at task level.

Software Interrupt Level

Software interrupt level, unlike the other execution environments, is distinguished only by the circumstances under which code (a task or an interrupt handler) is invoked, not the routines or memory to which the code has access. Software interrupt handlers execute at software interrupt level. A software interrupt handler is task-level software.

The conditions under which a completion routine in System 7 and a software interrupt handler in Copland are invoked are very different. In System 7, a completion routine is called as a result of a hardware interrupt and it executes at hardware interrupt level.

Suppose a task makes an asynchronous I/O request. When it makes the request, the task can create a software interrupt, specify a software interrupt handler, and ask that the software interrupt be issued when the request completes. The software interrupt contains the task's identifier.

In servicing the request, hardware interrupt handlers and secondary interrupt handlers are invoked. Only when they have completely finished servicing the request is the software interrupt issued. When no hardware interrupt handlers or secondary interrupt handlers are waiting to execute, the microkernel once again schedules tasks to run. When the task that created the software interrupt is eligible to run, its software interrupt handler is invoked.

Running a software interrupt handler in a task is like forcing the task to call a specific subroutine immediately. When the handler exits, the task resumes what it was doing. A software interrupt handler affects only the task in which it is run; the task running the handler can still be preempted so that other tasks can run. Those tasks in turn can run their own software interrupt handlers. A task running a software interrupt handler can also be interrupted by hardware interrupt handlers or secondary interrupt handlers. All software interrupt handlers for a particular task are serialized—they don't interrupt other software interrupt handlers for the same task.

The software interrupt handler executes in the task using it: the handler runs in the task's address space, uses its stack and registers, has full access to the Toolbox, can cause page faults, and so on. The system state is identical in the task and in the software interrupt handler.

Both supervisor-mode and user-mode tasks can use software interrupt handlers.

Secondary Interrupt Level

Secondary interrupt level is similar to the deferred task concept in System 7. Hardware interrupt handlers that need to perform certain actions, but that choose to defer the execution of those actions to minimize hardware interrupt level execution, can append secondary interrupt handlers to the queue for subsequent execution. Supervisor-mode tasks can also use secondary interrupt handlers for synchronization purposes.

A **secondary interrupt handler** is a supervisor-mode routine that runs with hardware interrupts enabled, but task switching disabled. Although hardware interrupt handlers preempt secondary interrupt handlers, secondary interrupt handlers cannot preempt one another. The secondary interrupt handler queue is always processed in FIFO order and the execution of the queued handlers is always serialized. The secondary interrupt handler queue is always emptied prior to running any task-level software.

Secondary interrupt handlers can use only a subset of microkernel and OS services. No Toolbox services are available to them. Furthermore, they can only access memory that is physically resident; page faults at secondary interrupt level are illegal and system fatal.

Because all task execution is blocked while secondary interrupt handler routines are running, your software shouldn't remain at secondary interrupt level for long.

Hardware Interrupt Level

Hardware interrupt level execution happens as a direct result of a hardware interrupt request. When a device presents an interrupt to the system, the microkernel calls a hardware interrupt handler, which always runs in supervisor mode. Device drivers provide hardware interrupt handlers for their devices. **Hardware interrupt handlers,** such as parts of device drivers, run at hardware interrupt level.

Hardware interrupt handlers can use only a subset of kernel and OS services. No Toolbox services are available to them. Furthermore, they can access only memory that is physically resident; page faults at hardware interrupt level are illegal and system fatal.

The microkernel provides an interface to the processor's interrupt vectors. For a PowerPC processor, the microkernel does not prioritize interrupts. As a result, all hardware interrupt handlers on PowerPC-based computers are serialized because hardware interrupts are disabled during execution of an interrupt handler.

In System 7, I/O completion routines, VBLs, and Time Manager tasks run at either hardware interrupt level or as deferred tasks. Copland runs them at user-mode task level instead. Therefore, most code on the system, including completion routines, is run at task level, and less at interrupt level or with interrupts disabled, which provides the following benefits:

- The microkernel doesn't allow application code to be run at interrupt level, because the code could cause page faults. Interrupt time is minimized. Because applications never disable interrupts, interrupt latency is minimized and the time available for applications to run is increased. **Interrupt latency** is the time between when an interrupt is generated and the associated interrupt handler is executed.

- Page faults become invisible to application code, including completion routines.

Exceptions

Each task, secondary interrupt handler, and hardware interrupt handler should have its own exception handler for kernel and hardware detected exceptions.

Note
This is not the complete, high-level exception mechanism available in the C++ language. ◆

The microprocessor detects **exceptions**—that is, errors or other special conditions like addressing errors, arithmetic overflows, and illegal instructions—in the course of program execution. When one of these exceptions occurs, the microkernel tries to call a handler. The handler performs its action, then the microkernel resumes execution from where the exception occurred or transfers control as indicated by the exception handler.

If there's no handler for the exception, the microkernel's actions depend on the execution environment. A debugger is called if one's installed. If not, and the exception occurred in a task or a software interrupt handler, the task is terminated.

If no debugger is installed and the exception occurred in a secondary interrupt handler or hardware interrupt handler, the exception is fatal to the system. Therefore, secondary interrupt handlers and hardware interrupt handlers should always have exception handlers if they might conceivably get an exception—even if the handlers only jump to a safe exit point.

About the Copland Runtime Environment

Contents

This chapter describes the new runtime environment available with Copland. A **runtime environment** is a set of conventions that determine how code is loaded into memory, where data is stored and how it is addressed, and how functions call other functions and system software routines. The Mac OS software and your software together determine the runtime environment available on a particular Mac-compatible computer.

The Copland runtime model, based on fragments, consolidates System 7's many diverse mechanisms for loading and executing code. A **fragment** is a block of executable code and its associated data. Fragments are created by your development system's linker. The **Code Fragment Manager (CFM)** loads fragments into memory and prepares them for execution.

The Copland runtime environment is an evolution of the one introduced with System 7.1.2 for PowerPC-based Mac-compatible computers (described in *Inside Macintosh: PowerPC System Software*). While the use of fragments is evolutionary, Copland introduces a memory addressing and allocation model that is entirely new on the Mac OS. This new architecture provides robust memory protection at the operating-system level and expands the role of virtual memory.

You should read this document if you are designing new software for Copland or if you have existing software that you want to run on Copland. An understanding of the Copland runtime environment can help you design your software to take maximum advantage of the Copland OS.

Compatibility—Backward and Forward

Because it is fragment-based, PowerPC native code compiled for System 7 is supported by the Copland runtime environment. All Code Fragment Manager–based calling conventions in Copland remain consistent with those of System 7. The Copland runtime environment also supports System 7 software based on the use of the A-trap table—developed for the original 68K runtime environment—by running this software under emulation on the PowerPC processor.

Copland does *not* support the use of system extensions of type `'INIT'`. To support replacements for software of this type, Copland provides enhanced system services, many of which also eliminate the need for the patching that your application might have done in System 7.

For System 7 compatibility, Copland supports the existing mechanism for patching system software routines (for example, using the `GetTrapAddress` and `SetTrapAddress` routines) with local effect. However, many system services have been revised in Copland, and therefore your patch is not guaranteed to produce the results you intend. Copland does not support the use of the existing mechanism for boot-time or global patching.

Copland applications use the Memory Manager when allocating and releasing memory space in their heaps, which are still used by the Toolbox. The Copland Memory Manager fully supports the System 7 Memory Manager APIs.

Due to the number of changes in the addressing model introduced by Copland, software that circumvents the System 7 Memory Manager functions may require revision to run compatibly with Copland's virtual memory.

Copland supports all System 7 Virtual Memory Manager functions except `LockMemoryContiguous`.

Design Goals for the Copland Runtime Environment

The Copland runtime environment is designed to

- consolidate the mechanisms for loading and executing code
- optimize software that uses the Code Fragment Manager
- support extensibility
- organize memory in a manner that offers flexible address mapping, protection of allocated memory, and new facilities for sharing memory

The Copland runtime environment uses fragments as a unified mechanism for loading and executing software. System 7 requires that you understand and use many diverse code loading and executing mechanisms, including Code Fragment Manager containers, HyperCard extensions (stored in resources of type `'XCMD'`), Component Manager components, and driver resources (stored in resources of type `'DRVR'`). You could be required to use several of them to write a single System 7 application. Copland simplifies the knowledge base required by having only one mechanism—fragments—to understand.

The Copland runtime environment optimizes the performance of software based on the Code Fragment Manager over the performance of software based on the use of the A-trap table. (By comparison, System 7 for PowerPC-based computers emphasizes compatibility over performance.)

The design of the Copland runtime environment promotes extensibility in two ways. First, many system services have extensibility mechanisms built into them. You can easily extend them to meet the needs of your application. Second, Copland provides the Patch Manager, which enables you to patch the system when the extensibility of the system services doesn't meet the needs of your application.

Copland improves the mapping of virtual memory to physical storage by allocating storage from a paging device only when software needs additional memory.

Copland provides memory protection by assigning access permissions to allocated memory and by placing all server processes in their own address spaces, which are separate from the address space used by all cooperative processes.

Fragments

In Copland, all executable code is packaged in Code Fragment Manager fragments, hereafter referred to simply as fragments. The basic unit of executable code in Copland is a **fragment**. Each fragment consists of its code, static data, imported symbols, and exported symbols. Fragments that export functions and variables to other fragments are called **shared libraries**. Because all fragments are potentially sharable (although not all are actually shared), the terms *fragments* and shared *libraries* are often used interchangeably. In general, a shared library is used to resolve imported symbols during linking and also during the loading and preparation of some other fragment.

A shared library that is dynamically linked at execution time is called a **dynamically linked library**. A dynamically linked library exports code or data that can be referenced by another fragment. For example, during the linking process, an application fragment can import a math library and the Window Manager library. At execution time, those libraries are dynamically bound to the application.

Do not confuse the use of the term *shared library* in this document with the shared libraries specified by the Apple Shared Library Manager, which are not part of the Copland runtime environment.

Using shared libraries for software development has many benefits, including the following:

- Having software in separate pieces simplifies development. For example, if you need to enhance the spell-checking module of your application, you need only to change and replace that shared library. You can distribute the enhanced shared library as a replacement, instead of distributing a new version of the application or a patch.
- When two or more applications use the same shared library, memory is saved because only one copy of the code is in memory.

Before the code or data in a fragment can be used, it must be loaded into memory and prepared for execution. This process is usually handled automatically by the Code Fragment Manager.

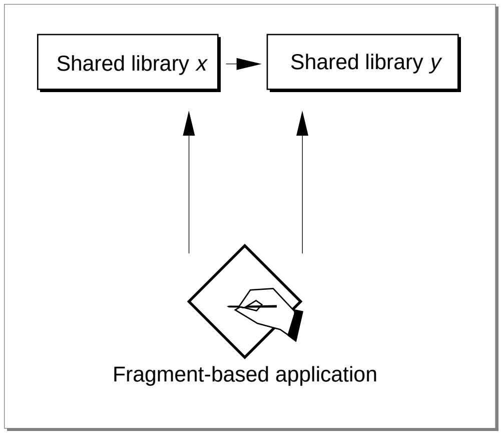
Copland supports the System Object Model (SOM), a new model for developing and packaging object-oriented software. It makes object-oriented shared libraries viable by providing release-to-release binary compatibility, compiler and language independence, and a basic level of dynamic language support. SOM is implemented as a layer on top of the Code Fragment Manager.

Shared Libraries

Copland provides all system services, including system software, through shared libraries. In Copland, all fragment-based software gains access to system services by directly calling shared libraries. System services also access each other directly, one shared library to another.

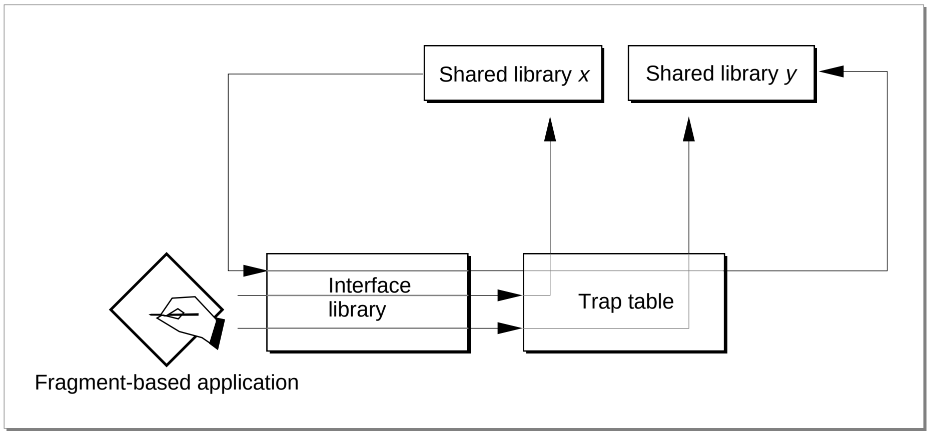
For example, Figure 4-1 shows a fragment-based Copland application accessing two shared libraries, *x* and *y*, directly. The figure also shows shared library *x* accessing shared library *y* directly.

Figure 4-1 Access to system services in Copland



In System 7, all software accesses system services through the trap table. A-trap-based software uses the trap table directly; fragment-based software calls an interface library, which in turn, uses the trap table. Figure 4-2 shows calls from a fragment-based application in System 7 to two shared libraries, shared library *x* and shared library *y*, going through the interface library and trap table. It shows the same process for shared library *x* calling shared library *y*.

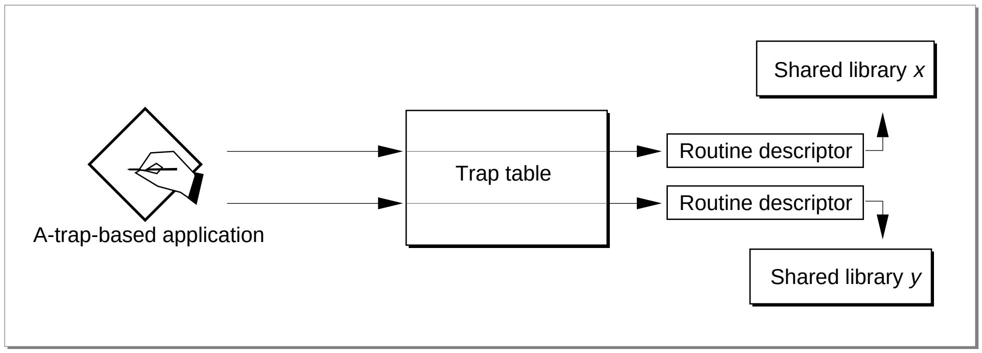
Figure 4-2 Access to system services in System 7



Software compiled as fragments for either a 68K or PowerPC processor can run only on that processor. All native PowerPC software is fragment-based. However, the Copland runtime environment also supports software that is based on the use of the A-trap table. This software was developed for the original 68K runtime environment. This software can run under emulation on the PowerPC processor.

The Copland runtime environment supports A-trap-based software by providing a trap table filled with routine descriptors that reference the system shared libraries. A **routine descriptor** is a data structure that describes the address of a routine, its parameters, and its calling conventions. Copland provides A-trap-based software support, shown in Figure 4-3, as an addition to the runtime environment instead of making it the focus of the runtime environment as in System 7.

Figure 4-3 Copland support for A-trap-based software



Copland takes full advantage of the capabilities of the Code Fragment Manager by allowing system services to use a process's static data. (Historically, this per-process static data has been referred to as an application's *global variables*.) There is a separate copy of per-process static data in each process that uses a shared library, and the static data of all libraries used by a process is shared by all of the tasks in that process, making concurrent access possible. By comparison, in System 7 the shared libraries of the operating system use only shared static data.

Memory Organization

Unlike previous versions of the Mac OS, Copland supports multiple address spaces and introduces functions for creating, deleting, and controlling areas of memory. An **address space** is the set of addresses that a process can reference.

System 7 and Copland both use a 32-bit address space, so that any address between 0x0000 0000 and 0xFFFF FFFF is a valid logical address. Whereas System 7 runs all software in a single 32-bit address space, Copland supports multiple address spaces. As explained in the chapter "About Copland Processes," all cooperative processes share one address space and server processes run in their own protected address spaces.

In Copland, at least 1 gigabyte (out of a maximum of 4 GB) of contiguous logical memory is available in each address space. The remaining logical memory is used for global allocation or reserved for other uses, such as slot space.

A **memory area** is a range of addresses, within an address space, that share common attributes, including:

- access permissions for software running in either user or supervisor mode
- whether the microkernel must hold the memory area resident in physical RAM
- whether the microkernel should allocate pageable memory on-demand, or preallocate it for the entire area
- whether the microkernel should initialize the contents of the memory area by clearing it

You can create your own memory areas and specify their attributes to suit your software's needs.

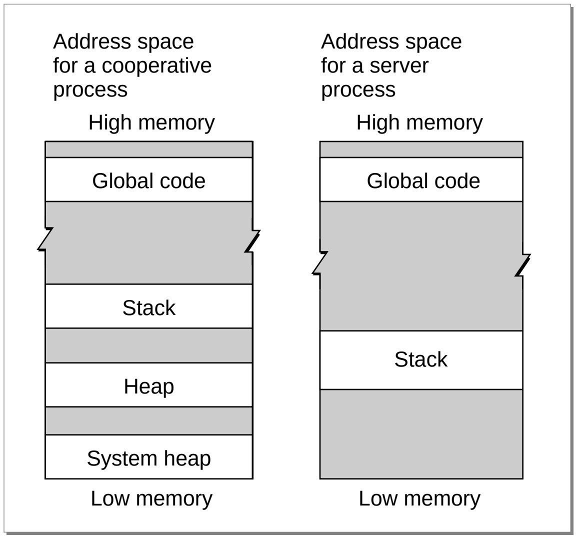
Copland maps **global memory areas** across all address spaces. In other words, each address space can address all global areas. Global areas, such as those occupied by system code, appear at the same location in every address space.

Figure 4-4 illustrates two different address spaces. The address space on the left belongs to cooperative processes; the stack and heap for an application's primary task is shown in this map. The address space on the right belongs to a server process; because secondary tasks have no heaps, only its stack is shown. Both address spaces share identical global areas—for example, the code of the shared libraries of the Copland File Manager. Both the application and server software can quickly call these shared libraries without leaving their own address spaces.

IMPORTANT

For illustrative purposes, the figures in this chapter show global areas near the top of address spaces. However, global areas are not necessarily in high memory, and you should make no assumptions about their locations. ▲

Figure 4-4 A global area across two address spaces



Memory Protection

Copland's facilities for memory protection reduce the possibility of a task crashing the entire system.

A task cannot directly reference an address in an address space other than its own. Therefore, software in different address spaces must use special system services to communicate across address spaces. This protects server software (as well as the microkernel) from applications.

Even within a single address space, Copland protects memory by assigning one of three access levels to each memory area:

- **read/write,** where read, write, and instruction-fetch operations are allowed
- **read-only,** where only read and instruction-fetch operations are allowed
- **excluded,** where no access is allowed

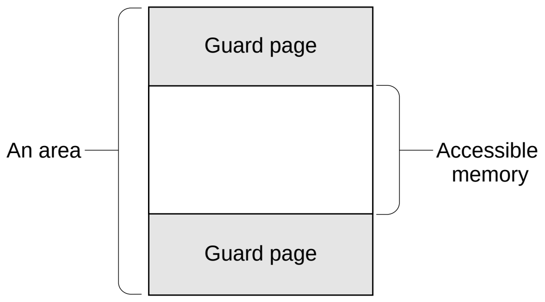
When Copland allocates a memory area, it assigns separate user mode and supervisor mode permissions for that area. For example, memory areas allocated for a device driver may allow read/write access to supervisor-mode software but read-only access to user-mode software. If code attempts to access memory to which it has insufficient access privileges, the processor generates an exception. (Exception handling is described in the chapter "About Copland Processes.")

Access permissions for global memory areas are of particular concern because global areas are visible to every task in the system. Executable code always resides in global memory areas that are assigned read-only access for all software. There are instances where a global area is assigned read/write permission for both user-mode and supervisor-mode software; for example, the video RAM is writable by all software.

This protection scheme does not cover all memory protection needs. In particular, it cannot protect a memory area that has been created with read/write access to user-mode software. Because all cooperative processes run in the same address space, Copland cannot prevent one application from corrupting another.

However, when Copland creates a memory area, it can place **guard pages** of memory at the beginning and end of the area. Copland allows no access whatsoever to these guard pages; neither user nor supervisor mode software can write to or read from these pages. Figure 4-5 illustrates a memory area created with guard pages. If any software, even the software residing in the area itself, attempts to access a guard page, the processor generates an exception. This makes it possible for Copland to detect conditions like stack overflows before they adversely affect surrounding areas.

Figure 4-5 A memory area with guard pages



Virtual Memory

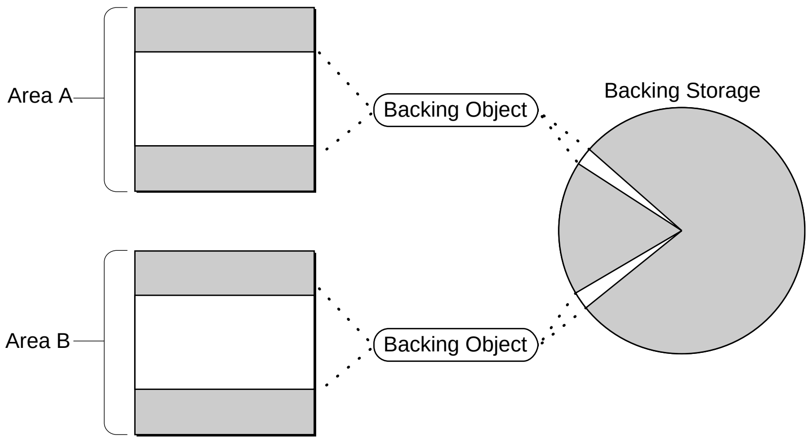
Virtual memory is always present in Copland, and it operates transparently to applications and other software executing in user mode. In Copland, virtual address space is dynamically allocated when needed and is released when no longer needed.

Note
Even while virtual memory is always in effect, any call to the `Gestalt` function on Copland using the System 7 selector `gestaltVMAttr` shows virtual memory to be off. ◆

Copland separates the recognition of a **page fault** (that is, the need to page in memory from backing storage to physical memory) from the software that resolves the fault. The **backing provider** is a server process responsible for transferring data between backing storage (for example, a hard disk) and physical memory in response to page faults. Copland supplies a backing provider to back physical memory to disk files.

In System 7, the user configures virtual memory through the Memory control panel. The user must allocate the size of virtual memory in this panel and then restart the computer for the changes to take effect. As shown in Figure 4-6, backing providers in Copland can allocate memory from the backing storage on an as-needed basis.

Figure 4-6 Virtual memory scheme on Copland



The microkernel signals the occurrence of a page fault in a message to a backing provider; a backing provider transfers the data into physical RAM. This separation allows the backing provider to use media other than local hard disks. In fact, the backing provider need not use actual backing store at all. For example, if a task needs to maintain a lookup table for a trigonometric function in memory, it might be faster and more efficient for the backing provider responsible for that memory allocation to recalculate the values of the table and place them directly in physical memory to satisfy a page fault rather than read them in from disk-based backing storage.

Backing messages include the following types of information from the microkernel:

- creation and deletion of memory areas
- reading from and writing to backing providers
- page aging and relinquishment

When the microkernel notifies a backing provider of the creation of a memory area, it allows the area's information to be checked by the backing provider before the microkernel completes the area creation. For example, if the backing storage used by a backing provider cannot be used as scratch storage (that is, it is read-only storage), then the backing provider should check the potential area's information to ensure that the microkernel is requesting nonscratch storage and read-only access permissions.

The kernel notifies a backing provider of memory area deletion, meaning that backing storage can be deallocated.

The microkernel notifies a backing provider when memory needs to be read from or written to backing storage. The backing provider is supplied with the starting physical memory address, the starting location in the backing provider, and the number of bytes involved.

A page aging message indicates that pages have remained untouched long enough that they appear unlikely to be referenced in the near future. The provider decides when to actually replace aged pages. A request for page relinquishment signals the microkernel's need to obtain physical memory from the backing provider; the need can be signaled as mild or urgent.

Be aware that device drivers, interrupt handlers, and any other software that cannot tolerate page faults must use Copland services to ensure that segments of code or data remain resident in physical memory. Copland provides functions to allocate physically resident memory and to change previously allocated memory from pageable to physically resident.

Heap Management

Copland introduces a reentrant, pointer-based memory allocator called the **Pool Manager.** Your software can use the Pool Manager to allocate memory for your data. The areas from which you can allocate memory are called **pools**.

Similar to heaps and temporary memory, pools provide applications and server software with memory. Unlike Memory Manager heaps, you can dynamically increase the size of pools. Unlike temporary memory in System 7, memory that you obtain from a pool can remain allocated throughout the lifetime of your process without degrading Copland's performance.

The Copland Memory Manager is a peer of the Pool Manager; both allocate memory from areas. Applications continue to use the Memory Manager when allocating and releasing memory space in their heaps, which are still used by the Toolbox. Copland fully supports the System 7 Memory Manager APIs.

Secondary tasks, which don't use the Toolbox, should always use the Pool Manager instead of the Memory Manager. Advantages of the Pool Manager are that

- it's fast
- its pools can be dynamically grown
- it's pointer-based (therefore, you don't need to lock and unlock handles)
- it's reentrant

Copland provides several preallocated pools to support use by cooperative processes, server processes, device drivers, and Toolbox managers.

In System 7, system software, drivers, desk accessories, system extensions, and applications have only the system heap and the application heap as alternatives for the allocation of dynamic memory. System 7 uses a temporary memory scheme useful mainly for emergency memory allocations that prevent applications from crashing.

The kernel creates three memory pools for the system upon startup:

- system resident pool
- system pageable pool
- system global pool

Device drivers and other supervisor-mode software commonly use the **system resident pool** when the software cannot tolerate page faults. The kernel holds memory allocated from this pool in physical RAM at all times. Only code running in supervisor mode can allocate memory from this pool. The data stored in the system resident pool is read-only for all user-mode software.

Supervisor-mode software that can tolerate page faults allocates memory from the system pageable pool. The **system pageable pool** acts as the default pool for supervisor-mode software. This pool, too, is read-only for all user-mode software.

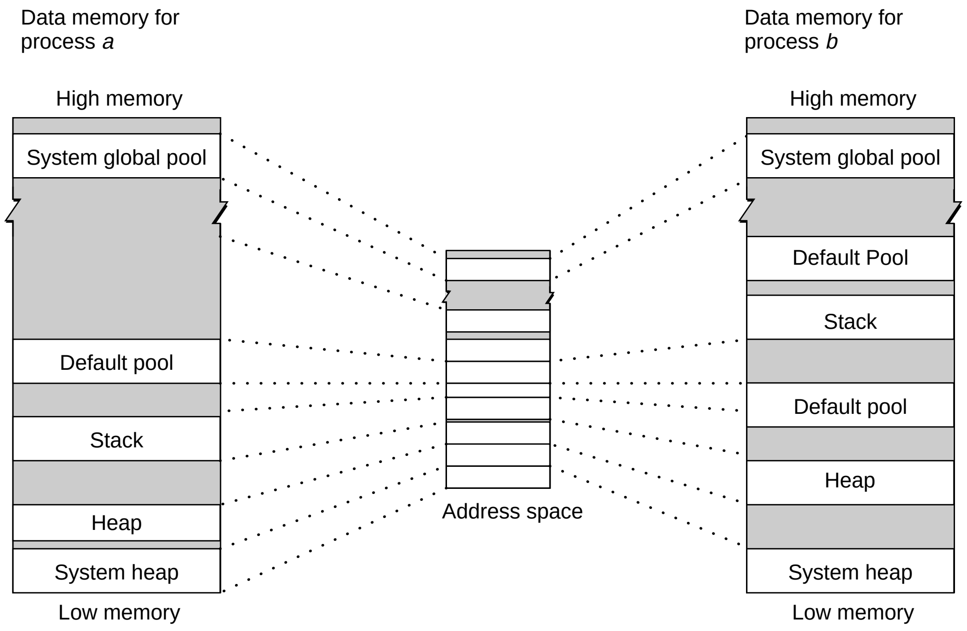
User- and supervisor-mode software can use the **system global pool** to allocate memory that must be globally accessible to all code in every address space. Use this pool sparingly—any code in the system can corrupt its contents.

Copland also creates a **default pool** for each process. User mode software can allocate from this pool at will. You can think of the default pool as being analogous to the application heap; it is pageable and nonglobal.

To allocate per-process static data, the Code Fragment Manager allocates one copy of a library's static data from the default pool for each process that uses that library.

Figure 4-7 illustrates how the data used by cooperative processes might be arranged. In this figure, the white boxes represent allocated areas.

Figure 4-7 Data memory areas for two cooperative processes

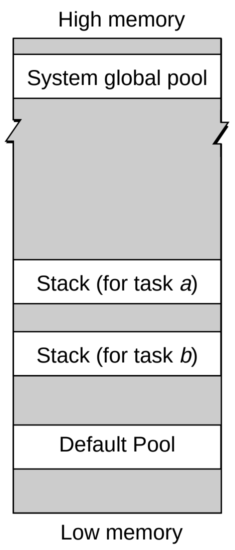


Because the cooperative processes shown in this figure share the address space of other processes that call the Toolbox, there is a system heap. Each process has its own heap and stack, and each has access to the system global pool, as shown in the figure. Copland also provides each process with a default pool from which to allocate memory. Notice that the default pool for process *b* consists of two discontiguous areas. Pools can be dynamically grown; when they grow, they may appear in discontiguous areas.

You may notice that the A5 worlds associated with A-trap-based code are absent from this address space. The information from those worlds is either not needed by fragment-based applications, or is maintained elsewhere (usually in the process's heap). Any software that makes assumptions about the organization of an A5 world will not work in Copland. For information about the new locations of the information formerly stored in A5 worlds, see *Inside Macintosh: PowerPC System Software*.

Every server process has its own address space; there are no heaps in the address space for a server process. There is a separate stack for every task in the process, and a default pool shared by all the tasks in the process, as illustrated in Figure 4-8.

Figure 4-8 Data memory areas for a server process with two tasks



The preallocated pools will generally meet your software's needs. Each pool uses a default grow function that provides a means for expanding the pool should it become exhausted. However, if none of the preallocated pools suit your software's needs, you can use the Pool Manager to create a new pool.

Using the Pool Manager’s creation functions, you can create pools with

- a specific size
- specialized grow functions

You can use a specialized grow function to grow your pool when it runs out of space; the default grow function allocates a new area for a pool that runs out of space.

Your software can create its own memory areas and specify attributes suitable for its needs. When you use the Pool Manager to create pools, they inherit the attributes of the areas from which they were created.

Extending the System

Copland provides a variety of services that let you extend the system to meet the specific needs of your application. Extensibility is incorporated into many of the Copland operating-system services so that you will no longer need to use system extensions or patches. For example, you can use preemptively scheduled tasks, discussed in the chapter “About Copland Task and Process Management,” to get system time instead of patching a regularly called system routine such as `SystemTask`.

Many Copland services provide support for application extensions. These application extensions use the Code Fragment Manager or SOM. Because SOM supports object-oriented software, it is especially useful for software that requires inheritance features.

If the extensibility of the Copland operating system doesn’t meet your needs, Copland provides a controlled mechanism for patching, called the Patch Manager. This new API enables you to perform such operations as naming, ordering, enabling, and disabling patches. Also, the Patch Manager allows the system to understand the relationships between patches and to determine where conflicts might arise.

For compatibility with System 7 software, Copland supports system software patching with the `GetTrapAddress` and `SetTrapAddress` functions. However, for an application making such a patch, this will have an effect only within that application’s process, and the execution of the patch will cause a performance penalty.

Note
The `GetTrapAddress` and `SetTrapAddress` functions will be removed from future versions of the Mac OS. Indeed, even if you use the Copland Patch Manager, Apple cannot ensure the future compatibility of your products. ◆

CHAPTER 5

About the Copland I/O Architecture

Contents

About the Copland I/O Architecture

This chapter provides an overview of the Copland I/O architecture. The Copland I/O architecture is designed to improve the user experience by providing superior performance, better responsiveness, and increasingly robust systems, and by supporting the advancements inherent in a microkernel-based operating system. It improves the developer experience by increasing the predictability of I/O responsiveness, by simplifying driver development, and by providing an updated 68K driver interface and an improved concurrent Device Manager.

You need to understand the framework that the I/O architecture provides for innovation and how it affects compatibility with both hardware and software products if you are one of the following types of developers:

- If you are a Mac OS licensee, you need to understand the I/O architecture to be certain that devices you incorporate into your hardware product will operate with Copland and to understand how software can be loaded into your product when it is turned on.
- If you are a hardware vendor who makes NuBus™ or PCI cards, ADB devices, GeoPort™ pods, or other hardware devices, you need to know how to create software that allows access to your product.
- If you are a system-extension author who produces software products such as network protocol implementations, file system implementations, and virtual device drivers to extend the capabilities of the system, or if you develop system utilities such as driver installers, hard disk formatting and partitioning packages, and emergency repair products, you need to understand the I/O architecture to determine if you need to modify your software product to run on Copland.
- If you are an application developer whose application writes to or otherwise manipulates devices, you need to understand how to take advantage of the new features in the Copland I/O architecture and how to enhance your application's compatibility with future versions of Mac OS.

This chapter briefly introduces the Copland I/O architecture. Then it discusses

- short- and long-term design goals of the I/O architecture
- architectural features, such as the Driver Loader Library, the Driver Services Library, booting services, power management, the user activity monitor, and support for hot swappable devices
- selected aspects of I/O families and plug-ins

- family activation models
- the Name Registry as it is used by the I/O system
- compatibility issues for device driver writers and application developers

You'll find this chapter easier to understand if you are familiar with certain features of Copland, such as its tasking mechanisms, the defined execution environments and execution modes, distinct address spaces, and microkernel messaging. You can find information about these topics in previous chapters in this document and in *Microkernel White Paper*.

Introduction

Copland changes how the lowest levels of the Mac OS work. It implements a tasking model of process management, with address space protection for tasks executing in supervisor mode. Drivers execute in supervisor mode. The transition to a microkernel-based, preemptive, multitasking operating system has significant implications for developers creating drivers and other I/O services for the Mac OS and for applications that use them:

- Applications running in user mode and driver software running in supervisor mode have no direct access to each other's data. Drivers are protected from applications and vice versa. Access to driver services is available only through an I/O family's programming interface.
- I/O devices are not directly accessible to application software, nor is it vulnerable to application error. Applications have access to hardware services only through an I/O family's programming interface.
- The context within which a driver runs and the method by which it interacts with the system are defined by the I/O family to which it belongs.

You can find more information on these topics in the section "Compatibility—Backward and Forward," beginning on page 5-34.

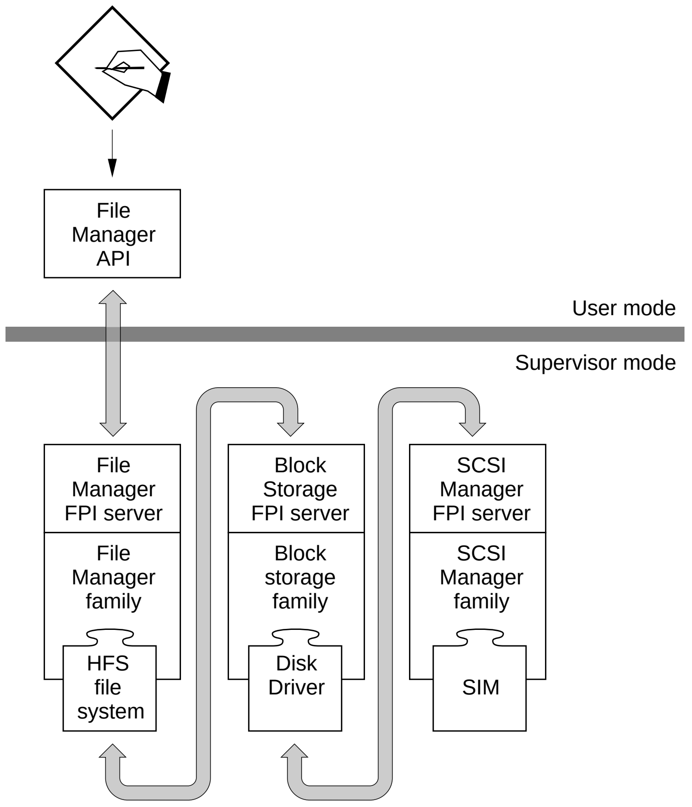
The Copland I/O architecture introduces new terminology. An I/O **family** is a collection of software pieces that provide a single set of services to the system, such as the SCSI family and its SCSI interface modules (SIMs) or the file systems family and its installable file systems. Each family defines a family programming interface (FPI) designed to meet the particular needs of that family. An FPI provides access to a given family's plug-ins.

A **plug-in** is a dynamically loaded piece of software that provides an instance of the service provided by a family. For example, within the file systems family (File Manager), a plug-in implements file-system-specific services. Plug-ins are a superset of device drivers—all drivers are plug-ins, but not all plug-ins are drivers.

Figure 5-1 illustrates an example of the relationship between an application, several I/O families, and their plug-ins. An application requests services through an FPI, shown in the figure as the File Manager API. Typically, the service requests flow as microkernel messages to FPI servers, shown in the figure as gray arrows.

In this architecture, code that executes in supervisor mode, such as plug-ins, family implementations, and the FPI servers, is *trusted.* A failure in one of these software subsystems can cause complete system failure. However, failure of any particular application does not affect the ability of the I/O system and other microkernel-level services to continue serving other clients. The I/O system is insulated from application error.

Figure 5-1 High-level view of an application, I/O families, and plug-ins



Note that Figure 5-1 shows three I/O families that work together to complete a service request. The application makes the service request which then moves through the file system family, the block storage family, and the SCSI family. However, this does not imply any hierarchical relationship among families. In fact, all families are peers of each other.

In introducing the concepts of family and plug-in, the Copland I/O architecture formalizes existing programming practices. For example, when an application accesses the services of a video device through the Display Manager, it is calling the display family. The Display Manager API is tailored to the needs of video devices. Likewise, when an application calls the Sound

Manager, it is calling the sound family. The family concept in the Copland I/O architecture explicitly acknowledges that devices of similar sorts share many characteristics and needs. Therefore, it provides family programming interfaces tailored to the needs of specific device families. These specially tuned sets of services allow drivers for a given family to be as simple as possible.

Families and plug-ins are described in more detail in the next two sections.

Families

The notion of family is fundamental to the Copland I/O architecture. A family provides a distinct set of services to the system. For example, the Open Transport family and its Data Link Provider Interface (DLPI) device drivers provide network services; the block storage family and its block storage drivers provide access to a variety of block storage mediums. Often, a family is associated with a set of devices that have similar characteristics, such as display devices or ADB devices.

Apple will provide the following families in its first release of Copland:

Device Manager family	Open Transport family
ADB family	Keyboard family
Pointing family	Display family
SCSI family	Sound family
PRAM family	IDE family
Real time clock family	PCI family
File systems family	PCMCIA family
Block storage family	NuBus family

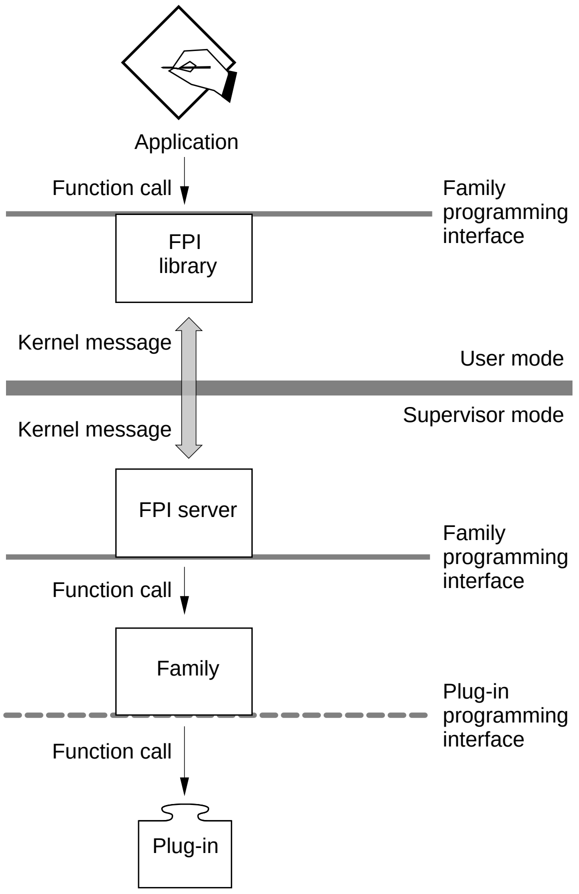
You can create additional I/O families, extending the base system features and APIs. Each family provides the following software pieces:

- a family programming interface and its associated FPI library or libraries for its clients
- an FPI server
- an activation model
- a family expert

- a plug-in programming interface for its plug-ins
- a family services library for its plug-ins

Figure 5-2 provides a high-level view of how selected family software pieces are related.

Figure 5-2 Family software diagram



The **family programming interface** (**FPI**) provides access to the family's services to applications, to plug-ins from other families, and to system software. The term *family programming interface* distinguishes an I/O family's API from other APIs provided by Copland, such as microkernel APIs or high-level Toolbox APIs. Each FPI is designed to provide callers with services appropriate to a particular family.

The FPI library contains the code that passes requests for service to the family FPI server. Typically, an FPI library maps FPI function calls into microkernel messages and sends them to the family's FPI server for servicing. To make certain optimizations possible, a family may provide two versions of its FPI library, one for user-mode clients and one for supervisor-mode clients.

An **FPI server** runs in supervisor mode and responds to service requests from family clients. How it responds to a request depends on the family's activation model. For instance, it may put a request in a queue or it may call a plug-in directly to service the request. If the FPI library and the FPI server use microkernel messaging to communicate, the FPI server supports a message port. The choice of microkernel messages as a communication mechanism is not visible to family clients. Clients use only the FPI to make requests of the family and its plug-ins. This is a change from the existing Mac OS in which both high-level and low-level interfaces to components of the operating system are available.

An **activation model** provides the runtime environment of the family and its plug-ins. For information about activation models, see the section "Activation Models," beginning on page 5-24.

A **family expert** (also referred to as a *high-level expert*) is the code within a family that maintains knowledge of the set of family plug-ins within the system. At system startup, and each time it's notified of a change in the Name Registry, the family expert scans the system's Name Registry for plug-ins that belong to its family. For example, a display family expert looks for display device entries. When a family expert finds an entry for a family plug-in, it instantiates the plug-in, making it available to clients of the family. The system notifies the family expert on an ongoing basis about new and deleted nodes in the Name Registry. As a result, the set of plug-ins known to and available through the family remains current with changes in system configuration.

Family experts do not add or alter information in the Name Registry, nor do they scan hardware. Families don't care about how devices are connected to the system—they are insulated from knowledge of physical connectivity. To learn how device information gets into the Name Registry, see the section "Name Registry," beginning on page 5-33.

The **plug-in programming interface** (**PPI**) provides a family-to-plug-in interface that defines the entry points a plug-in must support so that it can be called and a plug-in-to-family interface that defines the routines plug-ins must call when certain events, such as an I/O completion, occur. In addition, a PPI defines the path through which the family and its plug-ins exchange data.

A **family services library** is a collection of routines that provide services to the family's plug-ins. The services are specific to a given family and may be layered on top of services provided by the microkernel. Within a family, the family services library implements the methods by which data is communicated, memory is allocated, interrupts are registered and serviced, and timing services are provided. Family services libraries also maintain state information needed by a family to dispatch and manage requests.

For example, the services library for the display family provides routines that deal with vertical blanking because display devices care need them. Likewise, because SCSI device drivers must manipulate command blocks, the SCSI family services library contains routines to do that easily. A family services library that provides commonly needed routines simplifies the development of that family's plug-ins.

Plug-ins

A plug-in is a dynamically loaded piece of software that provides an instance of the service provided by a family. For example, within the file systems family, a plug-in implements file-system-specific services. The plug-ins understand how data is formatted in a particular volume format such as HFS or DOS FAT. But file systems family plug-ins don't understand how to get data from a physical device. To do that, a file system family plug-in talks to the block storage family. Block storage plug-ins provide both media-specific drivers—such as a tape driver, a CD-ROM driver, or a hard disk driver—and volume plug-ins that represent partitions on a given physical disk.

With the first release of Copland, Apple will provide plug-ins for the families listed on page 5-7. Third-party hardware developers are encouraged to develop new plug-ins.

All plug-ins share the following characteristics:

- They must conform to their family activation model.
- They cannot call Toolbox routines.
- They run in supervisor mode and have access to the microkernel's protected memory space.
- They are packaged as Code Fragment Manager fragments.
- They can be written in a high-level language.
- They must be written in native PowerPC code.
- They have a layered structure. Most of their work is done in a task. Some small amount of work may be done by interrupt handlers. The layered structure model for plug-in development allows code to be compartmentalized so that it works well within the Copland environment.

The typical parts of a plug-in include

- the main code section that runs as a supervisor-mode task. It is here that the plug-in does most of its work.
- a hardware interrupt handler that services hardware interrupts if the plug-in responds to a physical device. Only essential work that cannot be done in the task should be done by the hardware interrupt handler.

All plug-ins must have a main code section, but not all will have a hardware interrupt handler.

Plug-in code executes in supervisor mode and responds to client service requests made through the FPI. For example, Device Manager family plug-ins (device drivers of family type `'ndrv'`) respond to the functions `Open`, `Close`, `Control`, `Prime`, and so on.

Plug-in code should make no assumptions about particular hardware settings or configurations. The main code section should never attempt to obtain device configuration information directly from APIs such as the Resource Manager or the File Manager. A plug-in obtains configuration information in several ways. It can read the static configuration information stored in the Name Registry. Dynamically changing configuration information is communicated to a plug-in through the plug-in programming interface; when a family client uses the family's programming interface to notify the family of a configuration change, the family notifies the plug-in. In addition, a plug-in can call another family to obtain some types of configuration information. For instance, a video plug-in may call the PRAM family programming interface to obtain video mode information stored in PRAM prior to the last system reboot.

The hardware interrupt handler executes in supervisor mode and responds to interrupts from a physical device. It should perform only essential functions, deferring all other work to the plug-in task or a secondary interrupt handler. The plug-in programming interface specifies how interrupts are managed within a family.

Design Goals for the Copland I/O Architecture

The next two sections describe the short-term and long-term design goals of the Copland I/O architecture.

Short-Term Design Goals

In the first release of Copland, the I/O architecture is targeted to meet the following design goals:

- **End-user flexibility.** Mac OS provides end users with tremendous value that is directly attributed to the flexibility and adaptability of its I/O system. For example, its plug-and-play capability and dynamic monitor configuration are features that are simply not possible with many I/O architectures. The Copland I/O architecture is designed to provide these end-user features and to retain the flexibility of the Mac OS.
- **Performance.** The architecture favors lower-latency responses over higher bandwidths to provide greater responsiveness to users. To help achieve this goal, all drivers and all their support services are native. Additionally, very little code is permitted to run at the hardware-interrupt level. Although the architecture does not guarantee the best performance for burst and single-stream high-bandwidth clients, the Copland implementation will produce much better throughput results than that available in System 7. The I/O architecture provides support for the real-time needs of MIDI, Sound, GeoPort, and QuickTime and enables implementations that meet or exceed the performance of competing platforms.
- **PCI driver compatibility.** The Copland I/O architecture extends the architecture for the I/O system on PCI-based Mac-compatible computers.

Drivers compliant with the specification for driver development contained in the document *Designing PCI Cards and Drivers for Power Macintosh Computers* will continue to function well within the Copland I/O model. In addition, Copland seeks to provide binary compatibility with PCI ROM-based video and network drivers developed in accordance with the specification for native drivers described in *Designing PCI Cards and Drivers for Power Macintosh Computers*.

- **Reliability, availability, and serviceability.** In Copland, the I/O system works as expected and continues to work acceptably in the face of failures of particular subsystems. For instance, disk I/O continues to work if a failure in the serial hardware occurs. When failures do occur, the I/O system provides support for analysis and corrective measures by the user and by support organizations.
- **Resource allocation and control.** Having limited resources, the components of the Copland I/O system distribute those resources in a fair and meaningful fashion among themselves. In particular, the first driver loaded cannot consume resources such as memory, message ports, timers, interrupt latency, or bus bandwidth in a way that prevents subsequent drivers from loading or operating correctly. Configurations that cannot work because their needs are mutually exclusive are recognized and reported in a meaningful way.
- **Power management.** Obviously required for battery-powered systems such as PowerBook™ computers, the need for integrated power management is increasing for all systems. The I/O architecture provides an infrastructure to enable optimal power management in diverse systems.
- **Extensibility.** The Copland I/O architecture enhances the ability of OEMs to create Mac-compatible hardware and peripherals. It is intended that all hardware-dependent software fall into one of two categories:
  - software based on clearly defined hardware invariants such as big-endian addressing and the PowerPC 601, 603, and 604 processors
  - software that is dynamically loadable at system startup time, such as drivers, the SCSI Manager, and SCSI interface modules

Long-Term Design Goals

In subsequent releases of Mac OS, the I/O architecture is targeted to meet these additional design goals:

- **Multiprocessor support.** High-quality support for a limited number of tightly coupled, cache-coherent processors is a long-term goal of the architecture. While revisions to the architecture may be desirable for multiprocessor systems, conforming I/O components should be compatible within multiprocessor versions of the architecture.
- **Real-time I/O support.** The architecture specifies basic support for real-time I/O needs, largely as a subset of the resource allocation and control mechanisms provided by the architecture. Families and plug-ins are prioritized according to their needs to better support real-time clients.
- **Improved reliability, availability, and serviceability (RAS).** RAS is the natural successor to the Mac OS plug-and-play capability. The addition of RAS to Mac OS provides users, system administrators, and technicians with a broad set of tools for maintaining a Mac OS system, resulting in lower training and support costs. RAS is one of the mechanisms by which Mac OS will maintain its lead as the easiest and most configurable system available.
- **Visual system administration.** Enabling end users, system administrators, and support staff to examine and manipulate the configuration of a specific system is a natural extension to the benefits of RAS support.
- **Scalable to future technologies.** Copland provides sufficient architectural integrity to ensure that implementations of technologies that are not quite available today are obtainable on desktop platforms. ATM and infrared networking and Firewire bus connectivity are examples of such technologies.
- **Distributed computing.** As system performance increases, it is increasingly reasonable to provide access to devices that are not attached directly to the CPU on which an application is running. For example, with high-cost, high-speed networks, video capture via a frame-grabbing card plugged into a computer in another office is possible today. As networking costs decrease, distributed services become feasible on increasing numbers of desktop systems. Distribution of I/O subsystems across a suitable network is a long-term goal of this architecture.
- **Universal booting.** A single system image that boots on all hardware configurations that support Copland is a goal of the architecture. In addition, these systems will support both minimal and third-party customized installations of Mac OS.

Architectural Features

This section describes several fundamental I/O system services provided by the Copland I/O architecture. They are baseline services present in the system. They are not specific services for different classes of devices such as serial devices or video display monitors.

Driver Loader Library

The I/O architecture provides a Driver Loader Library. The **Driver Loader Library** is a set of routines that all I/O families can use to locate and instantiate their plug-ins. The routines work with all plug-ins regardless of whether the plug-in is a driver and regardless of whether the driver touches hardware. The services provided by the Driver Loader Library fall into three categories:

- routines that provide family experts with an easy way to instantiate plug-ins. All plug-ins are packaged as Code Fragment Manager fragments, frequently referred to as shared libraries. This set of utility routines serves as a wrapper around CFM functions. They hide CFM complexities, giving family experts a simple set of functions to access the shared libraries they need and load them into memory.
- driver matching routines that help family experts locate a device driver for a given piece of hardware. This makes driver replacement easy and provides support to families that manage drivers for hot swappable devices.
- routines that work with the Device Manager family. They install, remove, and replace driver entries in the unit table.

Driver Services Library

The **Driver Services Library** provides basic driver services to families. It contains all the base-level generic services needed by families and plug-ins, such as interrupt registration, timing facilities, allocation and deallocation of memory, and secondary interrupt-handling capabilities.

The Interrupt Manager is part of the Driver Services Library. It provides routines that allow drivers to install the interrupt handlers that are invoked when a device presents an interrupt to the system.

Families can extend the base system services in family-appropriate ways by adding a family services library to augment the services available from the Driver Services Library. In some cases, a family services library will replace the Driver Services Library. For example, plug-ins belonging to the Open Transport family don't link to the Driver Services Library, because the Open Transport family services library provides all the services they need.

Booting Services

The I/O architecture provides a method for loading and launching the system software. The Copland microkernel booting architecture maintains the Mac OS user experience at system startup. The user should not be required to build a system tailored for the hardware that the system will run on. Many users may choose to install hardware support for a large class of devices that might be connected to their computers. For those users, the system finds the right support software at startup time and configures that software into a runnable system without user intervention.

Power Management

The I/O architecture provides mechanisms for power state transitions within the system, such as bringing the system up the first time, shutting it down completely, moving from low to high power, and maintaining a sleep state. It provides APIs for power management at the application, plug-in, and system levels.

There are at least three systemwide power states:

- **Full power-on mode.** The core system is available for service requests. Within this mode, some devices, applications, and services may manage their power requirements independent of the system as a whole. Low-power mode is a substate of full-power mode, in that it affects only those devices that can continue to perform with less power.
- **Sleep mode.** The contents of memory are preserved, but active processing is halted.

- **Power-off mode.** The entire system is powered down and no processing of any sort is possible.

For the purposes of power management, there are three classes of devices and services:

- CPUs that have low-power modes in which some processing can still take place.
- Devices and services with a user interface that are therefore directly tied to user actions, such as keyboards, screens, modems, applications, and networks.
- Devices without a user interface, such as hard disks that may be controlled independently from user activity.

Given the fuzzy boundaries in the device and service categories and the varying nature of each device, the I/O architecture provides mechanisms for controlling power state transitions without setting policy for devices or services. A centralized power management service provides coordinated systemwide power state changes based on input from services and drivers.

The power state and power requirements of each device that is power managed is maintained in the centralized power management service. This power management service receives input from the User Activity Monitor service and individual applications and services. It provides notification to applications, drivers, and services, manages systemwide power state transitions, and provides centralized administration of device power behavior.

User Activity Monitor

Power management requires the ability to detect when the user is doing something with the computer. In Copland, the User Activity Monitor provides the power management service with information about user activity so that it can know when to put the system into sleep mode, turn a monitor down or off, and so forth.

Copland uses an activity timer to detect idle periods. Activity is defined as mouse motion or keyboard activity. Other events, such as the arrival of data on a serial interface, can also be considered activity.

The User Activity Monitor accepts requests for notification from I/O subsystems. Subsystems can request to be notified when a specified amount of time elapses during which there is no user activity. Any of the events defined as user activity cause the timer to be reset. Subsystems may also be notified that activity has occurred. This is useful when subsystems have already received notification of inactivity and powered down their hardware. Here are some examples of why a subsystem should use the User Activity Monitor:

- The screen backlight on a PowerBook computer needs to dim after a user-controllable amount of time elapses with no activity.
- The CPU should transition into low-power mode when no compute-bound process is running and a user-controllable amount of time elapses with no activity.
- The entire computer needs to transition into sleep mode after a user-controllable amount of time elapses with no activity.

The subsystems that can register activity must do so. They must tell the User Activity Monitor that activity has occurred, causing it to reset its inactivity timer and notify requesters (if any) of the event.

Support for Hot Swappable Devices

The Copland I/O architecture provides support for hot swappable devices such as PCMCIA cards—that is, it can support dynamic changes in connectivity to devices that may appear and disappear at any time. This feature allows a user to insert and remove devices such as disk driver card or modem card without powering down and restarting the computer. The family expert code that locates and instantiates the family plug-ins remains resident for families whose plug-ins exhibit dynamic plug-and-play characteristics.

A Closer Look

This section consists of selected topics concerning I/O families and plug-ins.

Families

The next sections discuss family programming interfaces and family communication models.

Family Programming Interfaces

A family provides either a user-mode or a supervisor-mode FPI library, or both, to support the family's FPI. Figure 5-3 illustrates an abstracted view of the Copland I/O architecture. Each of the large blocks in the area below the thick horizontal line represents an instance of a family. Boxes that share an edge represent directly callable interfaces.

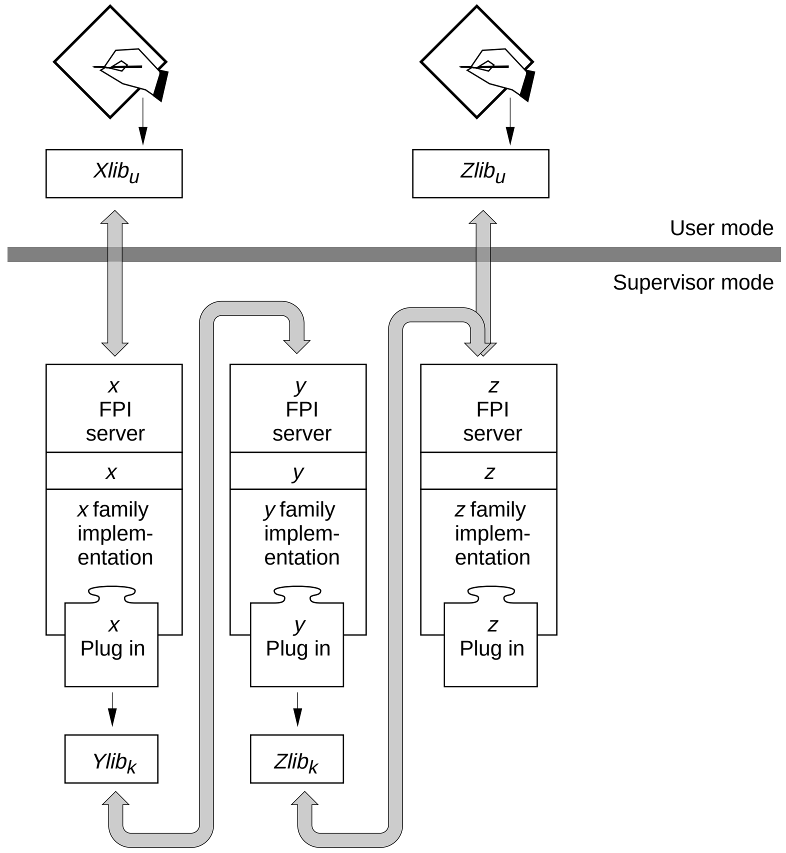
In the area above the thick horizontal line, the boxes labeled $xlib_u$ and $zlib_u$ represent the FPI libraries that support the programming interfaces for families *x* and *z*, and that are available to user-mode clients. In the area below the thick horizontal line, the boxes labeled $ylib_k$ and $zlib_k$ represent the FPI libraries for families *y* and *z* that are available to supervisor-mode clients. Typically, FPI libraries map FPI functions into microkernel messages.

Both the user-mode and the supervisor-mode versions of the FPI libraries present exactly the same interface to clients—a single FPI is the only way family services can be accessed. Copland distinguishes between the user-mode and supervisor-mode versions to permit optimization of the supervisor-mode FPI libraries in some instances. For example, operations that must be implemented in the user-mode library, such as copying data across address space boundaries, may be unnecessary in the supervisor-mode library. In some instances, the user-mode and supervisor-mode versions maybe the same.

An FPI server dispatches requests for services to the family. Typically, it does this by receiving a microkernel message, mapping the message back into the FPI function called by the client, and then calling the function. There is a one-to-one correspondence between the FPI functions called by clients and the functions called by FPI servers as a result. Take as an example the *x* family in Figure 5-3. The box labeled *x* represents the interface presented to the FPI server by the *x* family. It is exactly the same as the FPI available to applications or other system software.

The box labeled *x family implementation* represents the family activation model that defines how the request is actually serviced by family code and plug-in code.

Figure 5-3 A closer look at the Copland I/O architecture



Family Communications

Microkernel messaging is assumed to be the normal communication method for I/O families—between the FPI libraries and the FPI server for a given family, between different families, and between plug-in *x* and family *z*. That doesn't preclude the possibility of other communication mechanisms. The choice is up to the family. Whatever the communication method, it is completely opaque to a client requesting a family service.

The messaging model facilitates the development of families and plug-ins by providing a very easy programming model. It is a straightforward interfamily communication mechanism that fits well within Copland tasking mechanisms. The use of microkernel messaging permits greater independence of family activation models.

An added benefit to using microkernel messaging is that improvements in the messaging and tasking performance of the microkernel are reflected in corresponding performance improvements throughout the I/O system.

Plug-ins

Family plug-ins must operate within the activation model mandated by the family and provide the code and data exports described by family documentation. For example, *Designing PCI Cards and Drivers for Power Macintosh Computers* contains descriptions of the required interfaces and activation models for networking and video plug-ins. The required code and data exports and the activation model for each of these two families of drivers is family specific and different. The packaging for the two family driver types is the same.

The standard family and plug-in definitions cover most cases of I/O component development. However, there are exceptions to the model. The next sections describe two; there may be more.

Extending Family Programming Interfaces

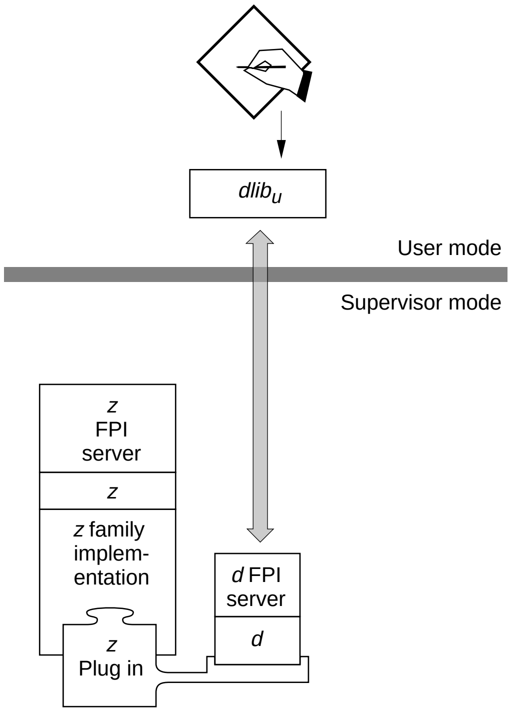
A plug-in may provide a plug-in-specific interface that extends its functionality beyond that provided by its family. This feature is useful in a number of situations. Take, for example, a block storage plug-in for a CD-ROM device. In addition to the block storage plug-in interface required of the CD-ROM device, many CD-ROM devices also present an interface that allows knowledgeable application software to control audio volume and to play, pause, stop, and so forth. Such added capabilities require a plug-in-specific API.

Most family interfaces provide some level of extensibility to the family's plug-ins. For example, the Device Manager allows extensible sets of control and status selectors that may be used to gain device-specific information and control. And Open Transport device drivers may receive special calls to extend the device information and control. This kind of device extension within the family framework is not changed with the Copland I/O architecture. If, however, a device wishes to export extended functionality outside the family framework, it needs to provide a separate message port and an interface library for that portion of the device driver, as shown in Figure 5-4.

Figure 5-4 illustrates a plug-in module labeled *z plug-in* that extends beyond the *z* family boundary. *z plug-in* is a plug-in with an extended API—it offers features in addition to those available to clients through it's family's programming interface. To make its extra services available, the plug-in must provide the additional software shown in Figure 5-4:

- *$dlib_u$*: the interface library
- *d FPI server*: the message port code
- *d*: the code that implements the extra features

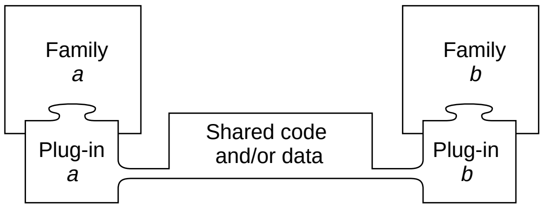
Figure 5-4 Extending a family programming interface



Sharing Code and Data Between Plug-ins

Two or more plug-ins can share data or code or both, regardless of whether the plug-ins belong to the same family or to different families. Sharing code or data is desirable when a single device driver wishes to subscribe to two or more families. Such a driver needs a plug-in for each family. These plug-ins can share libraries that contain information about the device state and common code. Figure 5-5 illustrates two plug-ins that belong to separate families and that share code and data.

Figure 5-5 Plug-ins that share code and data



Plug-ins can share code and data through Code Fragment Manager fragments, (shared libraries). The Code Fragment Manager allows you to instantiate independently plug-ins that share code or data without encountering problems related to simultaneous instantiation. The first plug-in to be opened and initialized gets access to the shared libraries, but it does not share access at that point. When the second plug-in is opened and initialized, it establishes a new connection to the shared libraries. From that point, the two plug-ins contend with each other for access to the shared libraries.

Sharing code or data is also desirable in certain special cases. Some of the special-case solutions provided on System 7 use two or more separate device drivers that use shared data as a communication mechanism. Typically, special case solutions install a set of devices and a set of special drivers. The closely coupled devices use a high-speed data path to move data between them. For example, a video input device puts video data in a shared buffer; subsequently, a video compression device reads and compresses the data it finds in the shared buffer. Access to the high-speed data path via the shared buffer is synchronized by solution specific mechanisms. In essence, this solution is a vendor-supplied family, and its plug-ins are the device drivers that come with the solution.

Activation Models

A family's activation model defines how the family software is implemented and the environment within which a family's plug-ins execute. It defines the relationship between family code and its plug-ins, including such things as

- the tasking model a family uses
- the opportunities the family plug-ins have to execute and the context of those opportunities (for instance, are the plug-ins called at task level? at secondary interrupt level? and so forth)
- the knowledge about states and processes that a family and its plug-ins are expected to have
- the portion of the service requested by the client that is performed by the family and the portion that is performed by the plug-ins
- the required characteristics of plug-ins, such as whether the plug-in blocks or returns an error when it encounters resource exhaustion

If you want to develop a new I/O family, you need to design and implement an activation model that best suits the needs of your I/O family. If you want to develop a new plug-in, you need to understand the activation model used by the family to which your plug-in belongs.

This section describes three family activation models used in the Copland I/O system. Each model provides a distinctly different environment for the plug-ins to the family, and different implementation options for the family software. The activation models discussed are

- the single-task model
- the task-per-plug-in model
- the task-per-request model

Many variations of (and hybrid approaches to) the activation models discussed here are possible and to be expected. The choice of activation model is left to the family designer. The selected models are simply examples of how you can implement a family.

To provide the asynchronous or synchronous behavior desired by the family client, the three activation models discussed here use microkernel messaging as the interface between the FPI libraries that family clients link to and the FPI servers. Within all activation models, asynchronous I/O requests are provided a task context. In all cases, the implementation of the FPI server depends on the family activation model.

The choice of activation model limits the plug-in implementation choices. For example, the activation model defines the interaction between a driver's hardware interrupt handler and the family environment in which the main driver code runs. A plug-in must conform to the activation model employed by its family.

You cannot understand the discussion of activation models without some understanding of Copland's messaging system and the tasking and interrupt mechanisms that define the environments in which software executes. You can find information about these topics in earlier chapters in this document and in *Microkernel White Paper*.

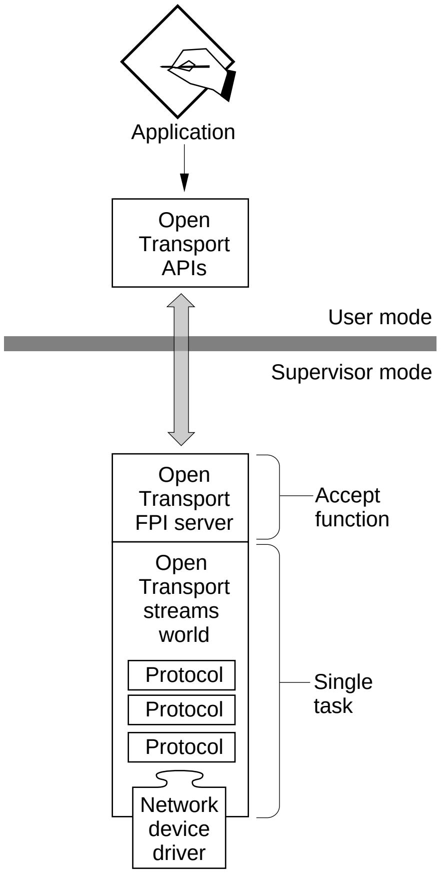
Single-Task Model

In the single-task activation model, the family runs as a single monolithic task that is fed from above by a request queue and from below by interrupts delivered by the plug-ins. Requests are delivered from the FPI library to an accept function that queues the request for processing by the family's processing task and wakes the task if it is sleeping. Queuing, synchronization, and communication mechanisms within the family follow a well-defined set of rules specified by the family.

The interface between an FPI server and a family implementation using the single-task model must be asynchronous. Regardless of whether the family client called a function synchronously or asynchronously, the FPI server always calls the family code asynchronously. The FPI server must maintain the set of microkernel message IDs that correspond to messages to which the FPI server has not yet replied.

Consider as an example the Open Transport family, which uses the single-task activation model, shown in Figure 5-6. The Open Transport FPI server is an accept function that executes on the thread of the calling client via the FPI library. An accept function, unlike message-receive-based microkernel tasks, is able to access data within the user and microkernel bands directly. The accept function messaging model requires that the Open Transport FPI server be reentrant because the calling client task may be preempted by another Open Transport client task making service requests.

Figure 5-6 Single-task activation model



When an I/O request completes within the Open Transport environment, the Open Transport stream's completion notification trickles upstream until it reaches the stream head and from there the Open Transport family's FPI server converts the completion into the appropriate microkernel message ID reply. The Open Transport family implementation is insulated from the microkernel; it has no microkernel structures, IDs, or tasking knowledge. On the other hand, the relationship between the FPI server and the Open Transport family code is rich, asynchronous, and has internal knowledge of Open Transport data structures and communication mechanisms.

The single-task model is best for families of devices that have either of two characteristics:

- Each I/O request requires little CPU effort. This characteristic applies not only to keyboard and mouse devices but also to DMA devices to the extent that the CPU need only set up the transfer.
- No more than one I/O request is ever handled at once. This characteristic might apply to sound, for example, or to any device for which exclusive access is required. It also applies to families that monitor their own scheduling for the interleaving of family I/O processing, such as Open Transport.

Here are the key questions to ask before deciding whether to choose this model:

- Can the CPU initiate an I/O request rapidly and then not be involved until the request completes?
- Do supported devices implicitly allow only one I/O request to be completed at a time or does the family provide for its own I/O scheduling?

If the answer to either question is yes, the single-task model is the right choice.

Task-per-Plug-in Model

In the task-per-plug-in activation model, for each plug-in instantiated by the family, the family creates a task that provides the context within which the plug-in operates. In Copland, the Device Manager family uses the task-per-plug-in activation model. Figure 5-7 illustrates the task-per-plug-in model using the Device Manager family as the representative family,

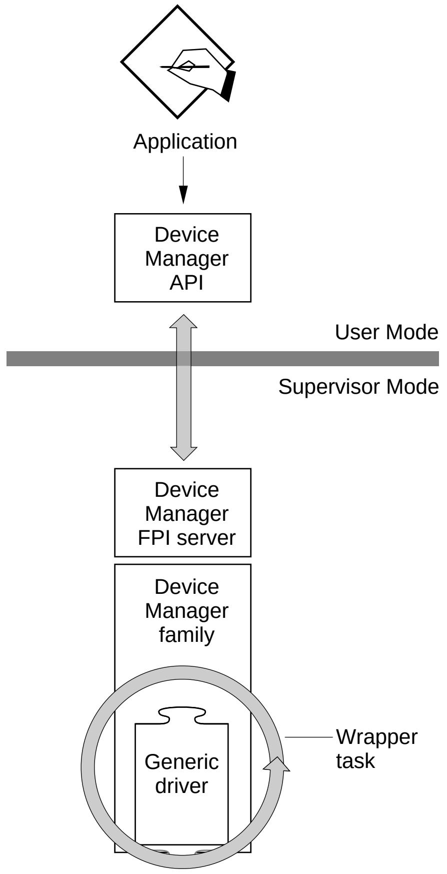
Typically with this model, the FPI server is a simple task-based message-receive loop or an accept function that presents data to an event-based task loop. The FPI server receives requests from calling clients and passes those requests to the family plug-ins. The FPI server is responsible for making the data associated with a request available to the family, which in turn makes it available to the plug-in that services the request. In some instances, buffers associated with the original request message may need to be copied or mapped once.

The family code consists in part of one or more tasks, one for each family plug-in. The tasks act as wrappers for the family plug-ins—all tasking knowledge is located in the family code.

When a plug-in's task receives a service request (by whatever mechanisms the family implementation uses), the task calls its plug-in's entry points, waits for the plug-in's response, and then responds to the service request.

The plug-in performs the work to actually service the request. It doesn't need to know about the tasking model used by the family or how to respond to event queues and other family mechanisms. It just needs to know how to perform its particular function.

Figure 5-7 Task-per-plug-in model



For concurrent drivers, all queuing and state information describing an I/O request is contained within the plug-in code and data and within any queued requests. The FPI library forwards all requests regardless of the status of outstanding I/O requests to the FPI server. When the client makes a synchronous service request, the FPI library sends a synchronous microkernel message. This message blocks the requesting client, but the plug-in's task continues to run within its own task context, permitting clients to make requests of this plug-in even while another client's synchronous request is being processed.

For the Device Manager family, generic drivers can be either concurrent or nonconcurrent; clients of the Device Manager family can make both synchronous and asynchronous requests. The Device Manager FPI server knows that nonconcurrent drivers cannot handle multiple requests concurrently. Therefore, it provides a mechanism to queue client requests. It makes no subsequent requests to a plug-in’s task until the task signals completion of an earlier I/O request.

The FPI library makes sure both synchronous and asynchronous clients see appropriate behavior. When a client calls a family function asynchronously, the FPI library sends an asynchronous microkernel message to the FPI server and returns to the caller. When a client calls a family function synchronously, the FPI library sends a synchronous microkernel message to the FPI server and does not return to the caller until the FPI server replies to the message, thus blocking the caller’s execution until the I/O request is complete.

In either case, the behavior of the Device Manager FPI server is exactly the same: for all incoming requests, it either queues the request or passes it to a family task, depending on whether the target plug-in is busy. When the plug-in signals that the I/O operation is complete, the FPI server replies to the original microkernel message. When the FPI library receives the reply, it either returns to the synchronous client, unblocking its execution, or it calls the asynchronous client’s I/O completion routine.

The task-per-plug-in model is intermediate between the single-task and task-per-request models in terms of the number of tasks it typically uses. It is best used where the processing of I/O requests varies widely among the plug-ins. In this model, the plug-in is insulated from microkernel tasking mechanisms and from synchronization issues that result from system resource contention and multiple client requests to a single plug-in.

Task-per-Request Model

The task-per-request model shares the following characteristics with the two activation models already discussed:

- The FPI library to FPI server communication provides the synchronous or asynchronous calling behavior requested by family clients.

- The FPI library and FPI server use microkernel messages to communicate I/O requests between themselves.

In the task-per-request model, the FPI server's interface to the family implementation is completely synchronous.

In this model, one or more internal family request server tasks, and, optionally, an accept function, wait for messages on the family message port. An arriving message containing information describing an I/O request awakens one of the request server tasks, which calls a family function to service the request. All state information necessary to handle the request is maintained in local variables on the thread of execution of the request server task. The request server task is blocked until the I/O request completes, at which time it replies to the microkernel message from the FPI library to indicate the result of the operation. After replying, the request server task waits for more messages from the FPI library.

As a consequence of the synchronous nature of the interface between the FPI server and the family implementation, code calling through this interface must be running as a blockable task. This calling code is either the request server task provided by the family to service the I/O (for asynchronous I/O requests) or the task of the requester of the I/O (for certain optimized synchronous requests).

The task-per-request model is best for a family where an I/O request can require continuous attention from the CPU and multiple I/O requests can be in progress simultaneously. A family that supports dumb, high-bandwidth devices is a good candidate for this model. The Copland File Manager uses the task-per-request model.

One problem associated with this activation model is tuning the number of request server tasks to permit the desired level of concurrence. Tuning can be done dynamically: When the family detects that performance could benefit from more request server tasks to process more requests concurrently and there are resources to permit it, new tasks can be created as needed. Similarly, when resources become scarce or the number of concurrent requests is much smaller than the number of request server tasks available to handle them, some tasks can be destroyed, freeing their resources for other uses. This programming model requires the family plug-in code to have microkernel tasking knowledge and to use microkernel facilities to synchronize multiple threads of execution contending for family and system resources.

Family Programming Issues

The choice of activation model is the biggest family programming issue. Each of the models discussed previously has merit. Within each model, there are issues to be addressed. The single-task and task-per-plug-in models require state information to be stored either within the FPI libraries, the plug-ins, or the family activation code, or within some combination of those. The task-per-request model is the simplest model, but it will probably be the most expensive model in terms of system overhead. It makes heavy use of microkernel messaging and tasking resources.

Unless there are multiple task switches within a family, the tasking overhead is identical within all of the activation models. The shortest task path from application to I/O is completely synchronous because all code runs on the caller's task thread. For a long I/O path, through multiple families, the greater the use of synchronous calls, the smaller the number of task switches. However, using only synchronous calls decreases the responsiveness of the application making the request— its activity stops pending the completion of an outstanding I/O request. Providing at least one level of asynchronous call between an application and an I/O request results in the best latency results from the user perspective. Within the file system, the application task is not used as the thread of completion for I/O. A task switch at the File Manager API level allows a user-visible application, such as the Finder, to continue. The File Manager creates an I/O task thread to handle the I/O request, and that task might be used via synchronous calls by the block storage and SCSI families to complete their part in I/O transaction processing.

This kind of short-cut communication between families requires a very clear understanding of the relationships between the families, including the stack needs of the called family, the activation model of the called family, and the asynchronous and synchronous paradigms used by the called family. This is part of the decision-making process in developing each family activation model.

Name Registry

The Name Registry is a high-level Mac OS naming service that stores system information. It is key to implementing several important features in the Copland I/O architecture:

- **Effective driver replacement and overloading capability.** This capability allows you to release updates to drivers.
- **Dynamic driver loading and unloading.** The Name Registry provides a dynamic and flexible environment for identifying devices. This type of capability is necessary for supporting devices such as hot swappable PCMCIA cards.
- **Simplification of driver writing.** You won't need to follow different rules for writing device drivers located on the main logic board, NuBus, the PCI bus, or the PCMCIA bus.
- **Hardware-independent device drivers.** The Name Registry provides the layer of abstraction necessary for driver writers to remove conflicting device identification and device information callouts (as occurred previously with the Slot Manager) that prevented drivers from being portable to new versions of Macintosh hardware.

The Name Registry is a tree-structured collection of entries, each of which can contain an arbitrary number of name-value pairs called properties. Family experts peruse the Name Registry to locate devices or plug-ins available to the family. Low-level experts, described later in this section, describe platform hardware by populating the Name Registry with device nodes.

The Name Registry contains a subtree pertinent to the I/O architecture: the device portion of the Name Registry describes the configuration and connectivity of the hardware in the system. Each entry in the device subtree has properties that describe the hardware represented by the entry and may contain a reference to the driver in control of the device.

A **low-level expert**, sometimes referred to as a *bus expert* or *motherboard expert*, has specific knowledge of a piece of hardware such as a bus or a main logic board. It knows how physical devices are connected to the system and it installs and removes that information in the device portion of the Name Registry.

For example, a SCSI bus expert scans a SCSI bus for devices and installs an entry into the device portion of the Name Registry for each device that it finds. The SCSI bus expert knows nothing about a particular device for which it installs an entry. As part of the installation, the SCSI bus expert invokes the driver matching routines in the Driver Loader Library to associate a driver with the entry. The driver knows the capabilities of the device and specifies that the device belongs to a given family.

Low-level experts and family experts use the Name Registry notification mechanism to recognize changes in the system configuration and to take family-specific action in response to those changes.

Here's an example of how family experts, low-level experts, and the Name Registry service work together to stay aware of dynamic changes in system configuration. Suppose that a Macintosh Duo is docked. The Duo motherboard expert notices that a new bus, a new network interface, and a new video device have appeared within the system. The Duo motherboard expert adds a bus node, a network node, and a video node to the device portion of the Name Registry. The Name Registry service notifies all software that registered to receive notifications of these events.

Once notified that changes have occurred in the Name Registry, the networking and video family experts scan the Name Registry and notice the new entry belonging to their family type. Each instantiates the new entry within the family.

The SCSI bus expert notices an additional bus, and probes for SCSI devices. It adds a node to the Name Registry for each SCSI device that it finds. New SCSI devices in the Name Registry result in perusal of the Registry by the block storage family expert. The block storage expert notices the new SCSI devices and loads the appropriate drivers, and then creates the appropriate volume Registry entries to make these volumes available to the File Manager. The File Manager receives notification of changes to the block storage family portion of the Registry, and notifies the Finder that volumes are available. Those volumes then appear on the user's desktop.

Compatibility—Backward and Forward

The following sections discuss Copland compatibility issues for developers of device drivers and applications.

If You Develop Device Drivers

Copland and its I/O architecture introduce a new environment for device drivers—one that is fundamentally different from that familiar to current Macintosh driver developers. Although Copland places some restrictions on drivers, it greatly increases system stability and protects drivers from application error.

The System 7 I/O architecture is based on resources of type `'DRVR'` and on the Device Manager API. Many different types of software use these mechanisms. Some types are affected by the changes introduced by Copland I/O and some are not.

Copland employs a more restricted concept of driver software. In the Copland I/O architecture, a driver is the native code that controls a physical device or that manages a system service. (Code that controls a virtual device such as a RAM disk may also be considered a driver in Copland.) This type of software (that controls a physical device or manages a system service) is affected by the new I/O architecture in Copland. Example of this type of software include

- serial drivers (.AIn, .BOut)
- protocol stacks (.MPP, .IPP)
- network drivers (.ENET, ADEVs, MDEVs)
- video drivers (.Display)
- SCSI interface modules (SIMs)

Software that uses the 'DRVR' resource type and the Device Manager API to provide application-level functionality is not directly affected by Copland I/O changes. Examples of this type of software include:

- desk accessories
- print drivers

For backward compatibility, Copland supports, through the Device Manager, emulated drivers of type `'DRVR'` that do not touch hardware. Such software is not a plug-in. It runs in user mode outside the I/O system and can exist only in the traditional application environment that uses the `WaitNextEvent` function and that has full access to the Toolbox.

The Copland I/O system is the first complete implementation of the I/O architecture described in this chapter. A subset of the I/O architecture is implemented to support PCI devices on upcoming Power Macintosh™ models. The document *Designing PCI Cards and Drivers for Power Macintosh Computers* describes the capabilities provided to driver writers for the first PCI-based Power Macintosh computers. If you write a PCI driver according to the specifications there, PCI cards with ROM-based drivers will work unchanged between the version of Mac OS delivered on upcoming PCI-based Power Macintosh models and subsequent PCI-based hardware platforms running Copland.

The Copland driver environment differs from the System 7 driver environment in several ways:

- The system distinguishes between software that runs in user mode or in supervisor mode. In System 7, drivers run in the same environment as applications in a single address space. In Copland, drivers run in supervisor mode and have access to the microkernel's protected memory space. Applications can't touch the hardware or the driver code or data directly.
- Drivers are packaged as Code Fragment Manager fragments (shared libraries).
- Distinct execution environments are defined in which different sets of services are available. Because drivers execute in supervisor mode, they cannot call Mac OS Toolbox routines. On the other hand, by executing in supervisor mode, drivers gain a fine granularity of control over devices and overall system responsiveness. Drivers use microkernel, driver, and family service libraries as appropriate. Families and their plug-ins are expected to adhere to the rules appropriate to their execution environment.
- The system employs new tasking and messaging mechanisms that allow prioritizing of I/O processing and that make I/O latency predictable. These mechanisms are the foundation for preemptive multitasking and memory protection.
- Drivers exist as plug-ins to a particular I/O family and must conform to the activation model employed by that family. Therefore, when writing your driver, you need to adhere to the plug-in programming interface and the family's implementation guidelines. I/O family provide libraries of commonly needed routines, thus simplifying your development effort.
- Drivers that touch hardware must be written in native PowerPC code. As a result, Copland will deliver superior I/O performance. Emulated 68K drivers that directly access hardware are not supported.

As a result of these changes, you need to change the way you write a device driver. With the exception of drivers written according to specifications for PCI-based Macintosh computers, System 7 drivers that access hardware will *not* run under Copland.

The next two sections give more information on the separation of application and device driver interfaces and the packaging of driver software and they describe benefits that result from these changes.

Separation of Application and Device Driver Interfaces

In System 7 there is only one kind of programming interface: the application programming interface (API). This makes all Mac OS services available to all varieties of software. Copland distinguishes between programming interfaces available to applications and those available to device drivers. Programming contexts become increasingly specialized in Copland.

In Copland, drivers have available to them plug-in programming interfaces specifically tuned to the needs of different types of devices, such as display devices or SCSI devices. The plug-in programming interfaces provide a fine level of control over core operating system facilities such as paging and interrupts. Use of plug-in programming interfaces is essential to your driver's portability in future Mac OS releases. These interfaces are guaranteed to be common across OS releases.

Because drivers operate outside the application software context in Copland, they do not have access to the rich set of APIs available to applications. If you find that a service you depend on has been removed from the plug-in programming interface for your driver, you should contact Apple at the AppleLink address NEW.IO or new.io@applelink.apple.com.

Common Packaging of Loadable Software

In Copland, all drivers are created as Code Fragment Manager (CFM) fragments (shared libraries). Each CFM fragment must export a driver description structure that the system uses to locate, load, and initialize the driver.

Copland drivers, therefore, are packaged differently from previous Macintosh device drivers. Because they are CFM fragments, they are allowed to have specific static data storage, and they can be written in a high-level language without assembly-language headers. Each instance of a single driver has private static data and shares code with every other instance of that driver. A device driver no longer locates its private data by means of a field in its Device Unit Table entry.

One consequence of drivers as CFM fragments is that a single device driver no longer controls multiple devices. Normally there is a driver instance for each device, although only one copy of the driver's code is loaded into memory.

If You Develop Applications

Adjusting to the architectural shift in the I/O system should be relatively easy for the application developer. For compatibility with System 7 applications, the Copland Device Manager supports all of the functions described in the chapter "Device Manager" of *Inside Macintosh: Devices*. However, a smaller set of devices will be available through the Device Manager; for them, the system supports a compatibility layer that converts old function calls to new ones. Thus, if your application calls the Device Manager, it will continue to run on Copland, but it will incur a performance penalty going through the compatibility layer.

For better performance and for access to services well suited to a given class of device, you should update your application to use the FPI for that device rather than the Device Manager. For example, if your application uses the Display Manager, you benefit from a set of routines tuned to work with display devices.

In most cases, Copland FPIs will be the same as or very similar to existing APIs, such as those provided by the File Manager, the Display Manager, and Open Transport. If your application uses these higher-level APIs, it is insulated from underlying changes in the Copland I/O architecture and Copland device drivers and you shouldn't have to change it to work with Copland.

In addition to benefiting from the more effective services available through Copland FPIs, adopting the new FPIs now facilitates subsequent development for versions of the Mac OS beyond Copland. APIs that Copland maintains for compatibility may not be available with versions of the Mac OS beyond Copland. For example, the networking paradigm for the Mac OS is changing, moving in the direction of Open Transport. Although Copland will support System 7 AppleTalk interfaces, later versions of the Mac OS will not. Versions of the Mac OS beyond Copland will require you to use the Open Transport FPI.

If your application ignores public APIs and instead uses nonstandard methods to access a device, you'll need to change your application. In Copland, hardware is not mapped into application address space and attempts to touch hardware will result in access violations. Devices and drivers are not directly accessible to an application. The only access to their services is through a family programming interface or an API maintained for compatibility.

Device Manager Compatibility

In Copland, the Device Manager functions described in the chapter "Device Manager" of *Inside Macintosh: Devices* are supported. Drivers that provide their services through the Device Manager API belong to the Device Manager family and are called **generic drivers**. The Device Manager functions constitute the FPI for the Device Manager family. The family has its own activation model and set of services, but it is not tuned to the needs of a given type of device.

Although the Device Manager API is more limiting than that provided by family FPIs, the Device Manager family offers a migration path to driver developers who implement the basic changes required by Copland without totally converting to the Copland I/O architecture.

If no family for a device exists, the Device Manager offers a way to use it in Copland. Consider, for example, a PCI card that receives data, encrypts it, and sends it back. An encryption family doesn't currently exist. By writing the driver according to the rules for drivers of family type `'ndrv'` described in *Designing PCI Cards and Drivers for Power Macintosh Computers*, the card is supported in Copland as a plug-in to the Device Manager family.

To summarize, the Copland Device Manager supports drivers that have been revised to run in Copland but that have not taken advantage of the enhanced driver services available through Copland I/O families, or for which no family exists. As a result, the Device Manager family's plug-ins are likely to differ quite a bit among themselves, rather than belonging to a general class of devices such as video monitors. For example, Device Manager family plug-ins may include drivers for instrumentation bus adapters, graphics devices, encryption hardware, and so forth. Typically, plug-ins in the Device Manager family are drivers that talk to hardware, but they can also talk to virtual devices such as a RAM disk or loopback software.

C H A P T E R 6

About the Copland File Manager

Contents

This chapter introduces the features of the new File Manager available with Copland. The **File Manager** defines the APIs that your application can use to manage the organization, reading, and writing of data located on persistent media. You should read this chapter if your product manages or manipulates files. If your software product creates, opens, closes, saves, or renames files, it can take advantage of Copland's new File Manager features for improved performance. If your product provides access to volume formats other than HFS (for example, a standard format such as DOS FAT or a custom format optimized for your customers' particular needs), the Copland File Manager simplifies your product development.

Compatibility—Backward and Forward

The Copland File Manager provides complete support for all System 7 File Manager functions used strictly in the manner documented in *Inside Macintosh: Files*.

If your application uses System 7 File Manager routines (and their associated data structures) to manage files and volumes, and does not directly manipulate the data structures or low-memory global variables used by the System 7 File Manager, it is compatible with the Copland File Manager. (Specific instances of when your application might not be compatible with the Copland File Manager—as well as specific recommendations about what you can do now to prepare your application to support the Copland File Manager—are described at the end of this chapter.)

The Copland File Manager is based on function calls through a shared library. In addition to supporting the System 7 File Manager APIs, Copland provides new APIs that are both simpler and more functional than the older ones. The new APIs are simpler because they do not use monolithic parameter blocks, but rather use smaller, logical data types to carry shared data between calls. As a result, the Copland File Manager has fewer than 80 entry points, while the System 7 File Manager has more than 170. The Copland File Manager APIs also offer support (for example, for large files and volumes) that the System 7 APIs cannot provide.

So that Apple Computer can respond more quickly and easily to your future needs, it has designed flexibility and extensibility into the Copland File

Manager. By adopting the Copland File Manager APIs instead of relying on the parameter block-based APIs of System 7, your application can take advantage of future improvements that Apple makes to the File Manager.

Design Goals for the Copland File Manager

The Copland File Manager is designed to

- increase speed
- support larger volumes and files
- support a variety of volume formats in addition to an improved version of the HFS volume format
- simplify product development by providing a richer and simpler programming interface
- support international file names by using TextObjects

Features of the Copland File Manager

This section describes the features of the Copland File Manager and its benefits to users and developers.

Increased Speed

The Copland File Manager provides significant performance improvements over previous versions of the File Manager, even when applications use the System 7 File Manager APIs.

- **Increased efficiency.** Better algorithms improve the performance of the Copland File Manager. For example, with the Copland APIs, your application can specifically define the information it needs; the Copland File Manager no longer creates—or inundates your application with—the extraneous information produced by such parameter block-based functions as `PBGetCatInfo`.

- **High-performance paging and memory cache.** The Copland File Manager is integrated with Copland's virtual memory system to provide high-performance paging and a new cache architecture, both of which take advantage of the Copland I/O system. This new cache architecture also uses Copland's paging mechanism to take advantage of all available memory.
- **Multitasking.** By taking advantage of Copland's multitasking capabilities, your application can create secondary tasks to efficiently perform file I/O processing in the background. Because the Copland File Manager is fully reentrant and takes advantage of the Copland microkernel to process multiple requests concurrently, a user's computer no longer wastes valuable cycles doing nothing while waiting for a file system request to be executed.

An Improved HFS Volume Format

To take advantage of contemporary storage technology, the Copland File Manager supports volumes and forks up to 8 exabytes (2^{63} bytes). Up to this limit, the sizes of the volumes and forks that the File Manager can support are limited only by the capabilities of a specific volume format.

A newly implemented HFS volume format, for example, supports 256-terabyte (2^{48} byte) volumes and 2-gigabyte (2^{31} byte) forks.Table 6-1 compares the limits of the HFS volume formats for System 7 and Copland.

Table 6-1 HFS volume and fork sizes

System version	Maximum volume size	Maximum fork size	Corresponding minimum allocation size
7.1	2 GB	2 GB	32 KB
7.5	4 GB	2 GB	64 KB
Copland	256 TB	256 TB	4 TB

(Because HFS allocates storage in units called allocation blocks and accounts for those with 16-bit values, the smallest unit of allocation in a volume is 1/65,536 the size of the volume. The size of an allocation block can be stored in a 32-bit value, and the allocation block size field can go up to a volume size of 2 terabytes.)

Support for Third-Party Volume Formats

The Copland File Manager is built around a message protocol designed to support a wide variety of volume formats. The new message protocol is easily extensible; this extensibility simplifies product development for you if you provide support for your own volume formats.

In addition to HFS, the Copland File Manager initially supports

- the AppleTalk Filing Protocol (AFP), which allows access to AppleShare and Personal FileShare volumes
- DOS FAT
- These common CD-ROM formats:
  - High Sierra
  - ISO 9960
  - Photo CD
  - Audio CD

The Copland File Manager also provides a generalized iteration mechanism for the searching and enumeration of files across different volumes and different volume formats. It replaces the current functionality of both the `PBCatSearch` and `PBGetCatInfo` functions with a mechanism that is both easier to implement now and easier to evolve into the future.

Ease of Development

The Copland File Manager simplifies your product development in several important ways.

Clearer and More Streamlined APIs

As previously mentioned, the Copland File Manager contains less than half as many APIs as the System 7 File Manager, but those that are there provide greater utility (such as access to large volumes). Instead of using the monolithic parameter blocks of System 7, the Copland File Manager APIs accept and return smaller, shared structures as parameters. Your application can reuse the values in these parameters in its API calls. Moreover, you will find these APIs easier to use because the distinction between input and output values is much more obvious than that for the System 7 APIs.

Event Notification

The Copland File Manager also provides notification for file system events, eliminating the need for your application to poll or patch system software to find out when there is a file system change. On Copland, your application can accept and process file system notification events.

New Concepts Behind the Copland File Manager

Three basic concepts form the foundation of the Copland File Manager: FSProperties, FSObjects, and FSLinks. These basic concepts are described in this section.

FSProperties

An FSProperty is a value, such as the name of a file, that is associated with a persistent structure in the file system. Each persistent structure in the file system can have a variety of FSProperties associated with it. FSProperties provide access to all of the system information that the Copland File Manager maintains.

All FSProperties are owned by a property service that defines the name space, allocation, and format standards for a set of FSProperties. Within each property service, a property selector identifies individual FSProperties. An OSType identifies each property service and each property selector.

The Copland File Manager is designed with the following expectations:

- FSProperties will be numerous.
- FSProperties will have small values that can be read and written as a unit.
- FSProperties will be accessed frequently by software other than the software that created them.

This pattern of access applies to most uses of FSProperties. Some units of data, however, do not fit this model. Many of the exceptions to this model involve FSProperties that are few, large, accessed by partial reads and writes, and tend to be accessed only by the software that created them. To support these exceptions, the Copland File Manager defines the Fork property service.

The Fork property service defines the Fork FSProperty and allows the use of an access method to open, read, and write forks. The Copland File Manager supports two types of forks: one whose property selector is `'data'` and one whose property selector is `'rsrc'`. Other restrictions apply to forks. For details, see the next section, "FSObjects."

FSObjects

An FSObject is a group of FSProperties that can be manipulated as a unit. The Copland File Manager supports four subclasses of FSObject:

- **File.** A file is a collection of FSProperties (including forks) that can be copied, moved, deleted, and renamed atomically in a file system. Files represent the largest collection of information that can easily be dealt with as a unit. While a program can use the Copland File Manager API to access and manipulate a finer granularity of information, the user views a file as a unit that is manipulated as a whole. For the Copland File Manager, the file is the only FSObject to which a fork can be added.
- **Directory.** A directory is a collection of files and directories. Directories allow users to organize files so that files can be found and used. Users often consider a file to be part of some group and want to name and access those files in the context of that group.
- **Prototype.** A prototype is a repository of shared FSProperties. Consider a group of related FSObjects that share a common set of values for certain FSProperties. Because storing and updating the shared values individually would be inefficient, the Copland File Manager provides Prototype objects for holding shared values. The Copland File Manager supports prototypes in a limited way: prototypes are based strictly a file's type and creator code.
- **Volume.** A volume is a unit of "availability" that contains directories and files. For every computer, there is a large amount of available information. Some of that information may be stored on hard disks attached to the computer, some may be stored on removable disks and floppies, and some may be stored on media that is only accessible across a network connection.

Of that information, only a subset is available at any given time. Volumes represent the smallest collections of files and directories that are available a unit. (Note that availability and access are different concepts: A volume may be available to the computer, but the user may not have the privileges required to access the data.)

Every FSObject can deliver a number, unique within a volume, by which it can be identified for the duration of the current mount. Every FSObject also has a name that is unique within certain contexts.

FSLinks

FSLinks are directional connections between FSObjects that associate related objects and that provide a way to navigate the collection of FSObjects in a file system. Every FSLink is owned by some FSRelationship that defines the interpretation and standards for a set of FSLinks.

Two FSObjects named A and B can be linked in four ways:

- A can be a “parent” of B.
- A can be a “child” of B.
- A can be a “sibling” of B if A and B are children of some other object named C.
- A can be a “spouse” of B if there is some object named C that is a child of both A and B.

The topology of a collection of links is constrained by the relationships to which the links belong. For instance, links belonging to some relationships are guaranteed to be acyclic (that is, a link that begins with A cannot return to A), while no such guarantee is made for links that belong to other relationships.

The Copland File Manager defines the following FSRelationships:

- **Hierarchy.** The Hierarchy FSRelationship groups FSObjects on a Volume. Hierarchical FSLinks are created from a directory to each of its content objects. Thus, the contents of the “parent directory” are “children” of that directory (with respect to the Hierarchy FSRelationship). The Hierarchy FSRelationship is defined to be acyclic, and every FSObject in the volume is defined to have a single hierarchical “parent.” For every volume, there is one distinguished “root” directory from which all files and directories (but not necessarily all prototypes) descend. All of the FSObjects that share a parent must have mutually unique names. Files cannot have hierarchical children.
- **Prototype.** The Prototype FSRelationship links one FSObject to a “parent” prototypical FSObject from which certain shared FSProperties can be obtained. The Prototype FSRelationship is defined to be acyclic, and every FSObject (including Prototypes) can have multiple parents. Note, the “inheritance” implied by a prototype link must always be invoked explicitly. The Copland File Manager never follows such links automatically. The Prototype relationship defines one distinguished “root” FSObject from which all Prototypes are Hierarchical (not Prototype) descendants.
- **Metaroot.** The Metaroot FSRelationship identifies the one universal object within the file system. No actual links can be created for this relationship, but the global and each per-volume distinguished objects for this relationship all refer to the same metaroot object (which does not itself exist on any volume). No named distinguished objects exist for the metaroot relationship.

- **Volume.** The Volume FSRelationship associates volume objects to the metaroot object. The only links that belong to this relationship are links created by the Copland File Manager that go from the metaroot object to each volume object. The distinguished object for the volume relationship on each volume is the volume object itself. There are no global or named distinguished objects for the volume relationship.

The Copland File Manager does not support the association of an FSProperty with an FSLink, nor does it support iteration over FSLinks.

Future Plans

The Copland File Manager is designed to be flexible and extensible for the future. This release will support a limited set of predefined of FSProperties and FSRelationships that will be supplemented in subsequent releases. Future releases will expand the functionality of prototypes, will support additional types of forks, and will support some kind of object that has both forks and children.

Preparing Your Product for the Copland File Manager

The following recommendations are offered to assist you in preparing your software product now to take advantage of the Copland File Manager.

- Make your application native for PowerPC-based computers.
- Don't assume 31-character filenames. Many volume formats have shorter or longer names.
- Do not use newline mode, because in the System 7.5 File Manager, newline mode is a mixed-mode switch. Instead, you should write your own function to put data into a buffer, and then write another to get each line of data out.
- Depending on the amount of memory you have and the speed of the device you are using, consider reading and writing file data in multiples of 16 KB. You have less overhead if you read an entire file into a buffer with one call instead of using multiple calls.

- Use the `noCacheBit` flag in the `ioPosMode` field of the parameter block. Don't cause other blocks of the cache to be flushed out to make room for data if you're not going to reread it.
- Isolate data storage methods.
- Isolate information calls. That is, write your own functions to get the data you need. For example, put wrappers around calls to `PBGetCatInfo` to define the information you require (for example, to get a file's type or creator).
- Keep reads and writes block aligned.
- Set the following when compiling your code:
  ```
  OLDROUTINENAME=0
  OLDROUTINELOCATION=0
  ```
- Preallocate forks before writing file data.
- Use 64-bit math for operations on all file system size data.
- Use the `FSSpec` structure to specify files and directories, because use of partial pathnames is limited on the Copland File Manager.

To improve I/O performance, the Copland File Manager provides concurrent processing of file processing requests. This has two important implications that you must consider when preparing your software product for Copland.

- Your software cannot touch any data after passing it to the Copland File Manager, because your application cannot depend on the state of that data.
- If your code extends the File Manager, your code must also be reentrant and able to handle concurrent requests.

As mentioned at the beginning of this chapter, your application will be compatible with the Copland File Manager if your application uses System 7 File Manager routines (and their associated data structures) to manage files and volumes, and if it does not directly manipulate the data structures or low-memory global variables the System 7 File Manager uses. Even if you do not take advantage of the new Copland File Manager APIs, you should make sure that your System 7 application runs on Copland. You will most likely have compatibility problems with the Copland File Manager if your System 7 software product does any of the following:

- creates or modifies data stored in file control blocks (FCBs) or modifies the file control block list

- relies on fields in an FCB other than `fcbFlNum`, `fcbFType`, `fcbCrPs`, `fcbDirID`, `fcbVPtr`, and `fcbCName`
- creates or modifies data stored in volume control blocks (VCBs) or modifies the volume control block queue
- relies on fields in a VCB other than `vcbFlags`, `vcbSigWord`, `vcbCrDate`, `vcbLsMod`, `vcbAtrb`, `vcbNmFls`, `vcbNmAlBlks`, `vcbAlBlkSiz`, `vcbFreeBks`, `vcbVN`, `vcbDrvNum`, `vcbDRefNum`, `vcbFSID`, `vcbVRefNum`, `vcbVolBkUp`, `vcbFilCnt`, `vcbDirCnt`, and `vcbFndrInfo`
- relies on or modifies any low-memory global variables maintained by the System 7 File Manager, such as `FSQHdr`, `FSBusy`, `FSQueueHook`, `ExtFSHook`, `ToExtFS`, `CkdDB`, `CurDB`, `NxtDB`, `FSCallAsync`, `MaxDB`, `FlushOnly`, `RegRsrc`, `FLckUnlck`, `FrcSync`, `NewMount`, `NoEject`, `DrMstrBlk`, `HFSStkTop`, `RgSvArea`, `hfsVars`, `HFSStkPtr`, `XRgSvArea1`, `HFSFlags`, `CacheFlag`, `SysCRefCnt`, `XRgSvArea2`, `SysBMCPtr`, `SysVolCPtr`, `SysCtlCPtr`, `XRgSvArea3`, `PMSPPtr`, `HFSTagData`, `XRgSvArea5`, `HFSDSErr`, `CacheVars`, `HFSVarEnd`, `cacheCom`, `XRgSvArea6`, `HFSDefaults`, `FmtDefaults`, `ErCode`, `Params`, `FSTemp8`, `FSTemp4`, `FSIOErr`, and `FSVarsPtr`
- relies on data returned by `GetFCBInfo` other than that returned in the `FCBPBRec` fields `ioNamePtr`, `ioVRefNum`, `ioRefNum`, `ioFCBIndx`, `ioFCBFlags` (only bits 9 and 14), `ioFCBEOF`, `ioFCBPLen`, `ioFCBCrPs`, `ioFCBVRefNum`, and `ioFCBParID`

A number of System 7 File Manager APIs are superfluous in Copland. Therefore, these calls perform no action, and instead return the `noErr` result code: `PBDTCloseDown`, `PBDTDeleteAsync`, `PBDTDeleteSync`, `PBDTRestAsync`, `PBDTResetSync`, `DILoad`, `DIUnload`, and `FInitQueue`. The functions `PBDTGetPath` and `PBDTOpenInform` return reference numbers usable only by other PBDT calls.

The Copland File Manager provides only limited support for the System 7 patching mechanism. To give your application greater control over the APIs that it absolutely needs to modify, Copland provides new mechanisms for patching the File Manager. See the chapter “About the Copland Runtime Environment” for information about these new mechanisms.

C H A P T E R 7

About Copland Networking

Contents

This chapter describes Open Transport, which is the networking component of the Copland operating system. **Open Transport** is a communications architecture that can be used to implement any number of networking and other communications systems. It replaces the AppleTalk, MacTCP, and Serial Driver interfaces in use today.

This chapter introduces Open Transport to developers interested in writing network applications for Copland. You should plan to use Open Transport rather than the current AppleTalk or MacTCP APIs for any new networking applications that require you to deal directly with the networking protocols. Similarly, you should use Open Transport rather than the current Serial Driver APIs for serial communication applications that call the serial drivers.

You should not use Open Transport if you can use one of the higher-level interapplication communication or collaboration technologies—such as PPC, Apple events, or PowerTalk—to achieve your goals.

Open Transport allows you to use a single set of APIs to implement any networking protocol available on Mac-compatible computers. By supporting industry standards at both the hardware and the software levels, Open Transport offers you an attractive cross-development platform for your products. Furthermore, it takes advantage of Copland technologies, including protected memory, multitasking, and multithreading, to provide higher performance for your network and communications products.

For more detailed information about Open Transport, see the Open Transport folder on the WWDC CD.

Compatibility—Backward and Forward

Open Transport includes a full set of backward-compatible programming interfaces for the AppleTalk and MacTCP protocol families. Therefore, any network application that uses public APIs and that works today with AppleTalk or MacTCP will continue to work.

However, unless you revise your application for Open Transport, your users won’t get all of the advantages of Open Transport. For example, your users won’t get the highest network performance on a Power Macintosh running Open Transport until your application uses native PowerPC code as well as the Open Transport interfaces.

Applications written using Open Transport will continue to work on the versions of the Mac OS beyond Copland. Later versions will not support the current AppleTalk and MacTCP APIs.

The Open Transport serial interface is built on top of the existing serial drivers. Therefore, device-driver (`PBControl`) calls to the serial drivers continue to work in Copland. The Open Transport serial interface also provides backward compatibility for the Communications Toolbox Connection Manager APIs. However, if you switch to the Open Transport APIs, the interface to the serial drivers is identical to that for any connection-oriented, transactionless protocol. Therefore, you can integrate serial communications easily into a communications application that also provides communications over a network. In addition, Open Transport provides TCP/IP communcations over a serial connection through implementation of the Point-to-Point Protocol (PPP).

The Communications Toolbox File Transfer Manager and Terminal Manager APIs are supported by Copland without modification. On a Power Macintosh computer, they run in native PowerPC code.

Open Transport does not support routers and bridges written using the older AppleTalk or MacTCP protocol implementations. Users can continue to run their old routers on Mac-compatible computers running the older protocol implementations, or they can replace them with new routers written to use Open Transport. Unless you are writing router or bridge software, this change does not affect your development effort.

Design Goals for Open Transport

The design goals for Open Transport include many important features, among them

- transport independence
- multihoming
- multinodes
- IP multicasting
- DHCP configuration and address leases
- improved human interface for configuration

- address resolution

The Open Transport APIs are independent of the underlying network or transport technology, a feature called **transport independence**. The set of functions you call and the sequence of calls depend solely on the nature of the communication; for example, you use one set of functions for any connection-oriented, transactionless protocol, such as ADSP or TCP, and a different set of functions for any connectionless, transactionless protocol, such as UDP or DDP. Once you have determined the *type* of protocol that is appropriate for your application, you can write your networking code without worrying about which protocol family will be used.

Multihoming is a key feature of Open Transport. **Multihoming** allows multiple Ethernet, token ring, FDDI, and other network interface controller (NIC) cards to be active on a single node at the same time. (A **node** is a device addressable at the network-layer protocol level. Examples of network-layer protocols are IP and DDP.) In addition to selecting the type of network connection, the user can select a particular device to be used for the network connection. For example, suppose a user has a LocalTalk LaserWriter in the office but uses it as a personal printer. The user also connects to the Ethernet backbone for file access and e-mail. Today, the user has to change the AppleTalk connection back and forth. With multihoming, Open Transport maintains two separate but equally useful connections. Yet there is no connection created between the networks.

Multinode architecture is an AppleTalk feature that allows an application to acquire node IDs that are additional to the standard node ID assigned to the system when the node joins an AppleTalk network. Multinode architecture is provided to meet the needs of special-purpose applications that receive and process AppleTalk packets in a custom manner instead of passing them directly on to a higher-level AppleTalk protocol for processing. A multinode ID allows the system running your application to appear as multiple nodes on the network. Whereas the older implementation of AppleTalk supports multinode architecture for use by Apple Remote Access (ARA) only, under Open Transport any developer can use this feature.

Video, audio, and other real-time multimedia data on the Internet are based on Internet Protocol (IP) multicast technology. An IP **multicast** is a message that is received by any number of TCP/IP hosts that are registered members of a multicast group. Open Transport provides full support for IP multicasts and multicast groups.

Dynamic Host Configuration Protocol (DHCP) is the emerging Internet standard for managing end-node IP configuration. DHCP automates the configuration process by returning information about the TCP/IP network to the user's configuration control panel. Open Transport also supports DHCP address leases, which allow a network administrator to configure a host's IP address to last for only a limited period of time. If the DHCP cannot renew the address lease when it is about to expire, Open Transport closes down the TCP/IP interface.

Open Transport provides a new, easier-to-use human interface. Apple Computer led the way to plug-and-play networking with AppleTalk, but TCP/IP has, for the most part, remained complex and hard to configure. One of the requirements for Open Transport has been to make it easier for people to use networking. This starts with making networking easier to install and configure. The difficulty with installation is compounded by the increasing mobility of computers. With today's portable computers, users may in a single day connect to several different networks, each with different configuration requirements.

To make networking easier for users, Open Transport uses a consistent human interface for configuring AppleTalk and TCP/IP networks (as well as for networks using other Open Transport–based networking protocols). Open Transport also clearly distinguishes between the protocol software and the configuration data. Network configuration information is stored in a preferences file. Auser can have multiple saved configurations, any of which can be selected through the control panel. A user can even use the control panel to export a configuration file that can be sent to a co-worker. In addition, Open Transport gives users the ability to reconfigure and restart network services without restarting their computers.

In developing your network and communication products, you can move away from dealing with protocol addresses. Certain Open Transport functions can accept host names, including AppleTalk names (based on NBP and ZIP) and TCP/IP names (based on DNS and BIND) and resolve them to addresses without further action by your application.

Cross-Platform Standards

Open Transport is based on three key standards: two from the X/Open Group—the X/Open Transport Interface (XTI) and the Data Link Provider Interface (DLPI)—and one, STREAMS, from UNIX® System V.

The Open Transport API is a superset of the XTI standard. Whereas XTI specifies functions only for transactionless protocols, Open Transport also includes functions for transaction-based protocols, such as ATP.

XTI complies with POSIX and XPG3, and it is at the "top" of the protocol stack. Protocol developers work in the STREAMS environment. Because Open Transport is a direct port of UNIX STREAMS, moving a protocol from UNIX to the Mac OS is a very easy process involving a few lines of code and compiler exports. (Within the STREAMS environment, Apple Computer is developing new versions of AppleTalk, TCP/IP, and serial communications protocols, and it is working closely with Novell to implement IPX protocols.) Hardware developers use the DLPI to integrate their networking cards into the Copland I/O architecture.

Preparing Your Application for Copland

Although Copland supports the older AppleTalk and MacTCP APIs, versions of the Mac OS beyond Copland will not. What's more, the Open Transport APIs are easier to use than the older APIs and make your application more transport independent. Therefore, you should use Open Transport for any new networking development you do for Mac-compatible computers.

Open Transport's support of standards—standard protocols, standard APIs, and standard development tools and environments—is a key element of its design. Because Open Transport takes a fully standards-based approach, you will discover that it is much easier to find experienced programmers to assist you in developing products. Moreover, because Open Transport uses a single set of APIs for all protocols, you no longer have to choose protocols when choosing an API. Instead, you can use the same set of APIs to run your application over multiple protocols. This ability increases your potential return on investment by both lowering development costs and offering access to broader markets.

C H A P T E R 8

About the Copland PowerTalk Environment

Contents

This chapter describes the PowerTalk system software, which provides a full set of electronic mail and collaborative services for the Mac OS. Using PowerTalk, you can build high-performance, multithreaded applications that access a wide range of collaboration and communication services. You can use these services, for example, to add PowerTalk mailers to your documents, store information in PowerTalk catalogs, send interapplication messages, and add digital signatures to documents and parts of documents.

The **PowerTalk** system software provides you with a consistent set of APIs to a wide range of collaboration services. The PowerTalk APIs for Copland provide you with three levels of services:

- The Collaboration Package provides a high-level API that lets you add mail and catalog services to your applications, and gives you access to the universal mailbox and its contents.
- Service access APIs allow for the development of gateways, catalog providers, and other service access modules.
- Low-level APIs provide you with detailed access to all of the collaboration services, including messaging, catalog, and authentication services, and digital signatures.

For a complete description of the PowerTalk system software that has already been released, see *Inside Macintosh: AOCE Application Interfaces* and *Inside Macintosh: AOCE Service Access Modules*. The new PowerTalk mailbox API that will be part of Copland is described in greater detail in other documentation on one of the WWDC CDs. The Key Chain API will be documented in future Copland developer releases.

Compatibility—Backward and Forward

Applications written with the current PowerTalk APIs will continue to work without modification on Copland, as these APIs are not changing. The collaboration system software takes advantage of improvements introduced with Copland; therefore your application's performance should improve considerably on Copland. What's more, if you have used the AOCE Collaboration Package to add PowerTalk mailers to your application, the mailers will automatically provide several new and improved features, such as return receipts and mail status information.

Design Goals for PowerTalk on Copland

The PowerTalk system software provided as part of Copland offers several improvements over the original PowerTalk software. These improvements can be grouped into two categories: new features and operating-system integration.

New Features for PowerTalk

Copland adds a number of new PowerTalk features and enhancements to meet the demands of users and developers.

- PowerTalk system software provides more e-mail features, such as return receipt, "unsend," "who's read," and folders in the mailbox.
- The PowerTalk catalog, mailbox, and mailer are fully scriptable using AppleScript.
- PowerTalk system software extends connectivity to more mail systems and to other collaborative technologies, such as pagers and faxes. Collaboration services support enterprise networks, Macintosh work groups, mobile users, and home offices.
- The PowerTalk mailbox API (introduced as optional software before Copland but integrated into Copland) provides access to all of the services provided by the mailbox but formerly unavailable to developers. An application can open a user's mailbox, search or filter for a specific type of message, open the message, act on the message's content, open and search the PowerTalk catalog, send a message or a fax, or even page the user.
- Users can check their mail on any Mac-compatible computer running Copland regardless of whether they are the principal user of the computer.
- The PowerTalk Key Chain API allows developers to add services to the Key Chain and to find out whether the Key Chain is locked.

Operating System Integration

Copland integrates the PowerTalk collaboration services into the operating system in the following ways:

- The new Copland human interface incorporates features first introduced with PowerTalk to provide a consistent look and feel throughout the Finder. For example, any folder in the Copland Finder can contain PowerTalk mail and display mail properties, such as Sender and Priority. Users look through folders, use active assistants, or use the Find function in exactly the same way regardless of whether they are dealing with files or mail.
- PowerTalk system software uses native PowerPC code on the Power Macintosh platform, greatly improving performance.
- PowerTalk uses the Copland file system for increased performance.
- PowerTalk uses Open Transport for increased performance.
- The PowerTalk Key Chain is further integrated to provide a single point of authentication to system services.

This level of integration ensures improved performance and an easy migration path for your existing collaboration applications to the Copland platform.

Preparing Your Application for Copland

Because PowerTalk will be part of every user’s HI once Copland is released, and because PowerTalk mailers are becoming even more powerful and easier to use, you should start adding PowerTalk mailers to your application documents today. You should also consider incorporating PowerTalk catalogs, store-and-forward messaging, and other services into active assistants to help users accomplish any number of cooperative or collaborative tasks.

Glossary

The terms defined in this glossary are preliminary and subject to change.

action The objects and activities that make a human interface task unique, such as the names of folders to back up or a list of people to whom a message should be sent. See also **task**.

activation model The tasking and synchronization model mandated by an I/O family. It consists of the code that provides the runtime environment to the family and its plug-ins. See **family, plug-in.**

active assistance General term for Copland features that allow the computer to assist the user actively. See also **automation, delegation, task, task definition, assistant.**

address space The set of addresses that a process can reference.

area A logical extent of memory with common attributes. Areas never overlap; a particular address in an address space is included in at most one area. See also **global area.**

assistant An entity that provides a specific kind of active assistance in a given context by asking the user questions and then taking actions or creating human interface tasks based on the answers.

automation Automatically controlled operations. In Copland, features that allow the computer to create a series of actions, or a human interface task, in a form that can be repeated. See also **task.**

backing provider A server process responsible for transferring memory between backing storage (for example, a hard disk) and physical memory in response to page faults.

bus expert See **low-level expert**.

CFM-based software Software compiled for execution in a runtime environment that uses Code Fragment Manager (CFM) fragments to organize executable code and data in memory.

code fragment See **fragment.**

Code Fragment Manager The part of the Mac OS that loads fragments into memory and prepares them for execution.

condition The set of events or states that trigger a human interface task. See also **task.**

cooperative process A process that has a primary task created by the Process Manager. The act of relinquishing control of the processor at a well-defined time, such as at a call to the `WaitNextEvent` function, is a defining characteristic of a cooperative process. See **process, server process.**

cooperative thread See **thread.**

delegation General term for Copland features that allow the computer to trigger human interface tasks when a specified condition occurs. See also **condition, task.**

desktop animation module A module controlled by the Desktop Animation Manager that can maintain the appearance of the desktop background or act as a screen saver when no keyboard or mouse events occur within a specified period of time.

Device Manager family A Copland I/O family to which two types of device drivers belong: those that have not been revised to run as a plug-in to another I/O family and those for which no family exists. See **family, plug-in.**

DLL See **dynamically linked library.**

document information panel A composite panel, available via the Document Info command in an application's File menu, that presents the same information presented by the Get Info command in the Finder's File menu. The document information panel also appears in an application's Save and Open dialog boxes. See also **panel.**

Driver Loader Library A set of routines that all I/O families can use to locate and instantiate their plug-ins.

Driver Services Library A set of routines that provide basic driver services to families, such as interrupt registration, timing facilities, allocation and deallocation of memory, and secondary-interrupt-handling capabilities.

Dynamic Host Configuration Protocol (DHCP) An Internet standard for managing end-node Internet Protocol (IP) configuration. DHCP automates the configuration process by returning information about the TCP/IP network to the user's configuration control panel.

dynamically linked library (DLL) A shared library that is automatically loaded by the Code Fragment Manager at runtime in order to export code or data referenced by another fragment.

exception An error or other special condition detected by the microprocessor in the course of program execution.

execution environment A set of conventions regarding how code gets activated and what services and memory are available to it. See also **hardware interrupt level, secondary interrupt level, software interrupt level, task level.**

expert See **family expert, low-level expert.**

extent Continuous memory space reserved on backing storage or in physical RAM for data or code.

family (1) A collection of software pieces that provide a single set of I/O services to the system, such as the SCSI family and its SCSI interface modules (SIMs) or the file systems family and its installable file systems. Often, a family is associated with a set of devices that have similar characteristics, such as display devices or ADB devices. (2) A collection of devices that provide the same kind of I/O services, for example, the family of display devices.

family expert Code within a family that maintains knowledge of the set of family plug-ins within the system. It locates family plug-ins in the Name Registry and instantiates them. Sometimes referred to as a *high-level expert*.

family programming interface (FPI) An I/O family's API that provides applications, other families, and system software with access to the family's services.

family server See **FPI server.**

family services library A set of services to which a family's plug-ins can subscribe. Such services can include communicating data, allocating memory, and registering and servicing interrupts. It provides services that supplement those available from the microkernel.

FPI See **family programming interface.**

FPI server Family software that runs in supervisor mode and responds to service requests from family clients.

fragment In Copland, the basic unit of executable code and its data.

generic driver A driver whose services are available through the Device Manager. Generic drivers are plug-ins to the Device Manager family. See **plug-in.**

global area An area mapped to all address spaces. Specifying this area attribute allows software residing in different address spaces to share data and code.

global family constants A set of shared family values.

guard pages Inaccessible pages of memory placed immediately before and after the range of addresses specified by an area. Copland does not actually allocate backing storage or RAM for guard pages; it merely marks the addresses in those pages as inaccessible to user-level or supervisor-level software.

hardware interrupt An exception signaled by a physical device to the processor, notifying it of a change of condition of the device, such as the reception of incoming data.

hardware interrupt handler Code that is invoked as a direct result of a hardware interrupt. Only essential work that cannot be done elsewhere should be done in a hardware interrupt handler. A hardware interrupt handler always runs in supervisor mode.

hardware interrupt level The execution environment in which a hardware interrupt handler runs. Only a subset of microkernel and OS services are available. No Toolbox services are available. Only memory that is physically resident is accessible; page faults at hardware interrupt level are illegal and system fatal. See **page fault.**

high-level expert See **family expert.**

interface definition objects (IDOs) Copland's SOM-based replacements for the definition procedures (defprocs) used in System 7.5.

interrupt See **hardware interrupt, secondary interrupt, software interrupt.**

interrupt handler A routine that services interrupts. See also **hardware interrupt handler, secondary interrupt handler, software interrupt handler.**

interrupt latency The time between the generation of an interrupt and the execution of its associated interrupt handler.

kernel See **microkernel**.

low-level expert Code that has specific knowledge of a piece of hardware, such as a bus or a main logic board. It knows how physical devices are connected to the system, and it installs and removes that information in the device portion of the Name Registry. Sometimes referred to as a *motherboard expert*.

memory area A range of addresses, within an address space, sharing common attributes.

microkernel The set of lowest-level operating system services, including memory management, task management, synchronization primitives, interprocess communication mechanisms, interrupt handling, and basic timing services.

motherboard expert See **low-level expert.**

multicast A message that is received by any number of hosts that are registered members of a group.

multihoming A networking feature that allows multiple network interface controller (NIC) cards to be active on a single node at the same time.

multinode architecture An AppleTalk feature that allows an application to acquire node IDs that are additional to the standard node ID assigned to the system when the node joins an AppleTalk network. Multinode architecture is provided to meet the needs of special-purpose applications that receive and process AppleTalk packets in a custom manner, instead of passing them directly on to a higher-level AppleTalk protocol for processing. A multinode ID allows the system running your application to appear as multiple nodes on the network.

Name Registry A high-level Mac OS naming service that stores system information.

network-layer protocol The protocol level directly above the data-link layer. The network layer is responsible for routing data between systems on the network.

node A device addressable at the network-layer protocol level.

non-reentrant code Code that should be executed by only one piece of software at a time.

notification The way in which a human interface task notifies the user that a task has been completed; for example, a notification might write details to a log and page the user via a commercial paging system when a task is completed. See also **task**.

Open Transport A communications architecture that can be used to implement any number of networking and other communications systems. It replaces the AppleTalk, MacTCP, and Serial Driver interfaces with a single, transport-independent interface.

page fault An exception that causes data to be transferred between backing storage (for example, a hard disk) and physical memory.

panel (1) A lightweight object that provides a SOM-based interface for human interface elements. (2) A hierarchy of embedded panels that make up a larger control panel, document information panel, or other integrated set of human interface elements. See also **document information panel.**

plug-in A dynamically loaded piece of software that provides an instance of the service provided by a family. Within the file systems family, for example, a plug-in implements file-system-specific services.

plug-in programming interface (PPI) A family–to–plug-in interface that defines the entry points a plug-in must support so that it can be called and a plug-in–to–family interface that defines the routines plug-ins must call when certain events occur. In addition, a PPI defines the data request path through which the family and its plug-ins exchange data.

pool A portion of logical memory from which software can dynamically allocate memory. A pool is contained in one or more areas.

Pool Manager In Copland, a reentrant, pointer-based heap allocator that allows software to dynamically allocate memory without impacting other areas of memory.

PowerTalk system software Apple Computer's implementation of the AOCE system software for use on Mac-compatible computers. The PowerTalk system software includes desktop services, as well as Mac OS managers and utility functions that provide APIs for catalog, messaging, and security services.

primary human interface The menus, windows, and tools that allow a user to control an application directly. See also **secondary human interface.**

primary interrupt level See **hardware interrupt level.**

primary task A task created by the Process Manager for an application when the application is launched. Cooperative scheduling of primary tasks is layered on top of preemptive scheduling of all tasks. The microkernel sees all primary and secondary tasks as peers and schedules them preemptively. The Process Manager ensures that for all cooperative processes, only one primary task is eligible to run at any one time. See also **secondary task, task.**

priority The ranking of a task used by the Copland microkernel for execution scheduling. The microkernel provides various symbolic priorities for applications, drivers, servers, and real-time operations.

privileged software See **supervisor-mode software.**

privileged task A task that runs in supervisor mode.

privilege level An access state to processor resources corresponding to the mode of the processor. Code that has the privilege to execute while the processor is in supervisor mode is called *supervisor-mode* *software*, and code that has the privilege to execute only while the processor is in user mode is called *user-mode software*.

process A set of one or more tasks and the memory and other operating system resources allocated to those tasks. For example, when Copland launches an application, the system creates a process that identifies the application's primary task, any tasks created by the application, and the memory areas allocated to those tasks. A process is specific to one address space, but multiple processes can share the same address space. Copland uses processes to track resource allocation and to reclaim resources. See also **cooperative process, server process.**

Process Manager In Copland, the part of Mac OS that manages the scheduling of cooperative processes and that controls access to resources shared by those processes.

reentrant services Code written so that the data it manipulates is kept logically separate from the code itself, allowing it to be safely called by several pieces of software at the same time.

routine descriptor A data structure that indicates the instruction set architecture of a particular routine by describing the routine's address, its parameters, and its calling conventions.

runtime environment The set of conventions that determine how code is loaded into memory, where data is stored and how it is addressed, and how functions call other functions and system software routines.

scaling The ability in Copland to control the extent of a user's access to system and application features.

secondary human interface A high-level interface that frees users from detailed decisions about how to make the computer perform specific actions. For example, assistants make decisions on the user's behalf and use the primary human interface to carry them out. See also **assistant**, **primary human interface.**

secondary interrupt A signal sent to the microkernel by a primary interrupt handler or a privileged task requesting that a secondary interrupt handler be queued for execution.

secondary interrupt handler A routine that runs as a result of a secondary interrupt. It always runs in supervisor mode. A secondary interrupt handler can be preempted only by hardware interrupt handlers.

secondary interrupt level The execution environment in which a secondary interrupt handler runs. Only a subset of microkernel and OS services are available. No Toolbox services are available. Only memory that is physically resident is accessible; page faults at secondary interrupt level are illegal and system fatal.

secondary task A task that uses only reentrant operating-system services. See also **primary task, task.**

semaphores Synchronization primitives used to block (schedule or switch out) tasks until a required resource becomes available.

For example, a global semaphore could be used to restrict access to the Toolbox to one application at a time.

server (1) A computer and associated software that provide a service to users and that control access to that service, such as a file server or a database server. (2) In Copland, a process that provides a service to other processes on the same or a connected computer. See also **server process.** (3) In the context of an I/O family, software built on the client/server model, but that has a single client. See also **FPI server.**

server process A process that has its own address space and whose task (or tasks) uses only reentrant system services. See also **cooperative process.**

shared library A fragment that exports functions and variables to other fragments. A shared library is used to resolve imported symbols during linking and also during the loading and preparation of some other fragment. Some shared libraries are dynamically linked; others must be explicitly instantiated during execution. (2) Any fragment.

software interrupt A signal sent to the microkernel requesting that it invoke a particular routine that, upon completion, returns execution at the point where the interrupt was sent.

software interrupt handler A routine that runs in a particular task as a result of a software interrupt. A software interrupt handler can be preempted by other tasks, secondary interrupt handlers, and hardware interrupt handlers. Software interrupt handlers can access virtual memory.

software interrupt level The execution environment in which a software interrupt handler runs. All microkernel, OS, and Toolbox services available at task level are available at software interrupt level. See also **task level.**

SOM See **System Object Model.**

supervisor mode A state of operation for the PowerPC processor that allows software to gain access to all of memory, all processor registers, and other critical resources. Only software with supervisor-mode privilege can switch the processor between supervisor mode and user mode. Compare **user mode.**

supervisor-mode software Code that executes while the processor is in supervisor mode. Typically, only the operating system and portions of device drivers should run in supervisor mode in Copland.

System Object Model (SOM) A technology from International Business Machines, Inc., that provides language- and platform-independent means of defining programmatic objects and handling method dispatching dynamically at runtime.

task (1) In the Copland human interface, a persistent representation of a sequence of actions that can be triggered programmatically. A task is created from a task definition in a manner analogous to the way a document is created from an application. See also **task definition.** (2) In

Copland system software, the basic unit of program execution that is preemptively scheduled by the microkernel. A task has its own stack and set of registers and it may share the same address space with other tasks. A task executes in either user mode or supervisor mode. In Copland, System 7 tasks such as Time Manager tasks and deferred tasks are invisible to the microkernel. See also **primary task, privileged task, secondary task.**

task definition A definition of how a particular kind of human interface task is to be performed. Combined with information about the parameters for a specific task, such as filenames or other details and a condition, a task definition can be used by an assistant or directly by a user to create one or more tasks. See also **task.**

task level The execution environment in which all tasks run. Most microkernel, OS, and Toolbox services are available at task level. Task level software can access virtual memory.

task switch The act of suspending one task's execution and resuming a different task's execution. In a task switch, the microkernel saves the processor state for the suspended task and restores the processor state of the task resuming execution. The microkernel performs a task switch based on the priority of tasks that are eligible to execute and its time-slicing mechanism. See also **priority, time slice.**

text object A private data structure that contains information about both text content and text encoding and that takes the place of both Pascal and C strings.

theme A coordinated set of human interface designs that determine the appearance and behavior of human interface elements on a systemwide basis.

thread Within a task, a sequence of instructions and the processor context to execute it, including a register set, a program counter, and a stack. In Copland, all threads are cooperatively scheduled by the Thread Manager.

time slice A defined interval of time during which a task is allowed to execute.

transport independence A property of a communications architecture that makes the set and sequence of functions called by an application independent of the underlying network protocols used to transmit or receive data. The set of functions called and the sequence of calls depends solely on the nature of the communication, not on the protocol or protocol family used.

trusted software Code that executes in supervisor mode and has access to the microkernel's protected memory space. See **supervisor-mode software.**

user mode A state of operation for the PowerPC processor that allows software, typically application software, to execute in an environment that protects certain critical resources, such as portions of memory and certain processor registers. Compare **supervisor mode.**

user-mode software Code that executes while the processor is in user mode. To protect the state of the user's system, applications should typically execute as user-level software.

workspace One of several separate custom user environments for a single computer.

VBL See **vertical retrace interrupt**.

vertical blanking interrupt (VBL) See **vertical retrace interrupt.**

vertical retrace interrupt An interrupt generated by the video circuitry each time the electron beam of a monitor's display tube returns from the lower-right corner of the screen to the upper-left corner.