I N S I D E   M A C I N T O S H

Macintosh Toolbox Essentials

Addison-Wesley Publishing Company

Reading, Massachusetts   Menlo Park, California   New York
Don Mills, Ontario   Wokingham, England   Amsterdam   Bonn
Sydney   Singapore   Tokyo   Madrid   San Juan
Paris   Seoul   Milan   Mexico City   Taipei

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

ISBN 0-201-63243-8
1 2 3 4 5 6 7 8 9-MU-9695949392
First Printing, October 1992

The paper used in this book meets the EPA standards for recycled fiber.

Contents

Figures, Tables, and Listings

P R E F A C E

About This Book

This book, *Inside Macintosh: Macintosh Toolbox Essentials,* describes the essential elements of a Macintosh application and the system software routines that you can use to implement them.

If you are new to programming on the Macintosh computer, you should also read *Inside Macintosh: Overview* for an introduction to general concepts of Macintosh programming and *Macintosh Human Interface Guidelines* for a complete discussion of user interface guidelines and principles that every Macintosh application should follow.

This book describes events, windows, menus, controls, alert boxes, and dialog boxes. It also discusses how your application interacts with the Finder.

Macintosh applications respond to user actions and to other hardware- and software-related events. To design your application so that it can respond to events (such as keyboard input, mouse input, changes in the appearance of windows on the screen, and changes in your application's processing status), see the chapter "Event Manager" in this book.

To create menus and set up your application's menu bar, see the chapter "Menu Manager." This chapter describes how to define the items in your menus, how to enable and disable menus, how to allow the user to choose a menu item, and how to respond once the user chooses a menu item.

To create windows in which the user can view or edit information, see the chapter "Window Manager." This chapter describes the basic types of windows and discusses how your application can work together with the Window Manager to support the standard user interface conventions associated with manipulating a window, such as moving a window, zooming a window, and resizing a window.

To create controls in your application's windows—such as scroll bars—or to create controls in dialog boxes—such as buttons or checkboxes—see the chapter "Control Manager."

To create dialog boxes or alert boxes—windows that your application uses to communicate with or solicit information from the user—see the chapter "Dialog Manager."

To create icons for your applications and the documents it creates, see the chapter "Finder Interface." This chapter also introduces file types and creators and describes the various kinds of resources (icons, file references, and bundles) that the Finder needs to display your application and the documents it creates.

After implementing the basic elements of a Macintosh application as described in this book, you can add additional features, such as help balloons and support for copy and paste, as described in *Inside Macintosh: More Macintosh Toolbox*. You can also find detailed information about the Resource Manager in *Inside Macintosh: More Macintosh Toolbox*.

Once you understand how to create menus, windows, and dialog boxes, you can save information that the user enters in a window by writing the data to a file. You can also open a previously saved file and read the information from the file into a window. You use the File Manager to open, read, write, and close files. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for information on how to read and write files.

For information about drawing into a window or other graphics port, see *Inside Macintosh: Imaging*.

For information on handling text in your application, see *Inside Macintosh: Text*.

For information on communicating with other applications, see *Inside Macintosh: Interapplication Communication*.

Format of a Typical Chapter

Almost all chapters in this book follow a standard structure. For example, the Event Manager chapter contains these sections:

- "Introduction to Events." This section presents a general introduction to the types of events that your application can receive.
- "About the Event Manager." This section provides an overview of the features provided by the Event Manager.
- "Using the Event Manager." This section describes the tasks you can accomplish using the Event Manager. It describes how to use the most common routines, gives related user interface information, provides code samples, and supplies additional information.
- "Event Manager Reference." This section provides a complete reference to the Event Manager by describing the data structures, routines, and resources it uses. Each routine description also follows a standard format, which presents the routine declaration followed by a description of every parameter of the routine. Some routine descriptions also give additional descriptive information, such as assembly-language information or result codes.
- "Summary of the Event Manager." This section provides the Pascal and C interfaces for the constants, data structures, routines, and result codes associated with the Event Manager. It also includes relevant assembly-language interface information.

Conventions Used in This Book

Inside Macintosh uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as the contents of registers, use special formats so that you can scan them quickly.

Special Fonts

All code listings, reserved words, and names of actual data structures, fields, constants, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the Glossary.

Types of Notes

There are several types of notes used in this book.

Note

A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 2-7.) u

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 5-27.) s

s **W A R N I N G**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page page 2-105.) s

Empty Strings

This book occasionally instructs you to provide an empty string in routine parameters and resources. How you specify an empty string depends on what language and development environment you are using. In Rez input files and in C code, for example, you specify an empty string by using two double quotation marks (""), and in Pascal you specify an empty string by using two single quotation marks ('').

Assembly-Language Information

Inside Macintosh provides information about the registers for specific routines like this:

Registers on entry

A0	Contents of register A0 on entry

Registers on exit

D0	Contents of register D0 on exit

In the “Assembly-Language Summary” section at the end of each chapter, *Inside Macintosh* presents information about the fields of data structures in this format:

0	`what`	word	event code
2	`message`	long	event message
6	`when`	long	ticks since startup

The left column indicates the byte offset of the field from the beginning of the data structure. The second column shows the field name as defined in the MPW Pascal interface files; the third column indicates the size of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see the discussion of the data structure in the reference section of the chapter.

The Development Environment

The system software routines described in this book are available using Pascal, C, or assembly-language interfaces. How you access these routines depends on the development environment you are using. When showing system software routines, this book uses the Pascal interface available with the Macintosh Programmer’s Workshop (MPW).

All code listings in this book are shown in Pascal (except for listings that describe resources, which are shown in Rez-input format). They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in many cases, tested. However, Apple Computer, Inc., does not intend for you to use these code samples in your application. You can find the location of code listings in the list of figures, tables, and listings. If you know the name of a particular routine (such as `DoEvent` or `MyAdjustMenus`) shown in a code listing, you can find the page on which the routine occurs by looking under the entry “sample routines” in the index of this book.

In order to make the code listings in this book more readable, they show only limited error handling. You need to develop your own techniques for handling errors.

This book occasionally illustrates concepts by reference to a sample application called *SurfWriter;* this is not an actual product of Apple Computer, Inc.

APDA is Apple’s worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most products. APDA offers convenient payment and shipping options including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog,* contact:

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone:	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (elsewhere in the world)
Fax:	716-871-6511

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M / S 75-3T
Cupertino, CA 95014-6299

C H A P T E R 1

Introduction to the Macintosh Toolbox

Contents

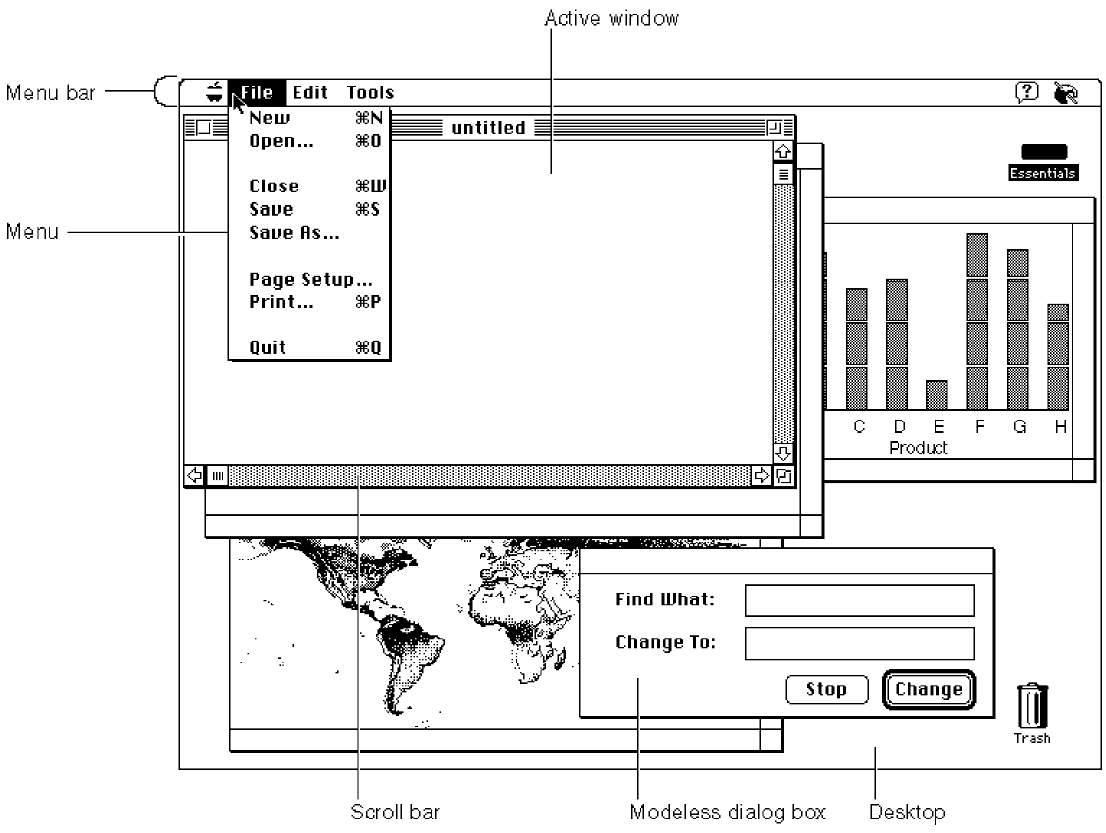
This chapter presents an introduction to the features provided by the Macintosh Toolbox. The Macintosh Toolbox is a collection of system software routines that your application can use to present a consistent and standard interface to the user; these routines also allow you to simplify other tasks your application might need to perform.

A typical Macintosh application presents a friendly, intuitive, easy-to-use, visual interface to the user. The careful design of a Macintosh application gives users the freedom to perform actions and accomplish tasks according to their needs. The idea behind this careful design is to put the user in control. In general, the user of a Macintosh application should always be free to choose the next action he or she will perform. (This is the basic tenet of the event loop and is explained in more detail in the chapter "Event Manager" in this book.)

Figure 1-1 shows the screen as it might appear when a user is interacting with a typical Macintosh application, such as SurfWriter. The SurfWriter application is an application that lets a user do simple text editing. Like most Macintosh applications, the SurfWriter application uses

- menus to let the user choose commands
- windows to allow the user to enter and edit information
- scroll bars to allow the user to view more information in a window

Figure 1-1 The SurfWriter application with multiple windows on the desktop



- other controls (such as the Change button) to let the user control various settings or options
- dialog boxes to solicit information from the user

You can create an application that incorporates these user-interface elements and that helps users accomplish specific tasks by taking advantage of the routines provided by the Macintosh Toolbox.

Overview of the Macintosh Toolbox

Macintosh system software contains a powerful set of routines that your application can use to create windows, manage menus, paint objects, display text, open files, share data between programs, and print files, as well as perform many other helpful tasks.

The Macintosh Toolbox encompasses a number of system software routines, most (but not all) of which help present your application's interface to the user. Some of these routines include those provided by the Event Manager, Menu Manager, Window Manager, Control Manager, Dialog Manager, Help Manager, Resource Manager, and Scrap Manager.

You can directly call these routines from within your application. By using system software routines, you can take advantage of the many tasks they can perform for you, and you can concentrate on the parts of your application that are specific to your particular product.

Using the Macintosh Toolbox, you can

- respond to user actions, such as mouse actions or keyboard input
- create and display menus
- create and display windows, alert boxes, and dialog boxes
- create and display controls in windows, alert boxes, and dialog boxes
- create icons for your application and its documents

This book, *Macintosh Toolbox Essentials,* describes these fundamental elements of a Macintosh application. *Inside Macintosh: More Macintosh Toolbox* describes additional features of a Macintosh application, including how you can

- create help balloons for your application's menus, windows, and dialog boxes
- support copy and paste
- specify characteristics of your application's menus, windows, controls, dialog boxes, and help balloons in resources so that you can more easily localize your application

The best Macintosh applications are designed according to the guidelines in *Macintosh Human Interface Guidelines*. You should always design your application so that it meets the needs of its users and responds in consistent and expected ways. *Macintosh Human Interface Guidelines* describes

- the philosophy and the design principles behind the Macintosh interface
- the parts of the Macintosh interface including the interface elements and behaviors
- ways to do human interface design for Macintosh products

You can often get valuable feedback on the design of your application by performing user testing. Do usability testing of your application early and often in the development phase of your product.

Events

At the core of every Macintosh application is the application's event loop. The event loop is that piece of code in an application that processes and responds to user actions and other events. You can use the Event Manager to retrieve information about these actions. For example, you can get information that tells your application whether the user pressed a key or the mouse button, whether one of your application's windows needs updating as a result of the user moving windows, or whether some other hardware or software action requires a response from your application.

You should structure your application so that it can respond to events and so that the user is able to perform tasks in any order. For example, a user should be able to type text in a window, select a graphic and copy it, open a new document, paste in the graphic, open another document, and then go back to the first window to select text and change its typeface, size, or style.

Your application should respond to events in a way that lets the user switch between your application and others whenever the user chooses to do so (for example, by clicking in a window belonging to another application). Your application should also yield time to other applications when it isn't busy. System software provides a cooperative multitasking environment that allows users to switch between many open applications and that allows applications to receive available processing time when other applications aren't using the processor. System software coordinates the scheduling of processing time between your application and other applications.

You can also let your application communicate with other applications in order to request services or information from another application or to provide services to other applications. You can use the Event Manager or Apple Event Manager to do this.

The chapter "Event Manager" in this book describes how to structure your event loop and event-handling code to receive notification of user actions and changes in the processing status of your application, how to communicate with other applications, and how to respond to these events.

Menus

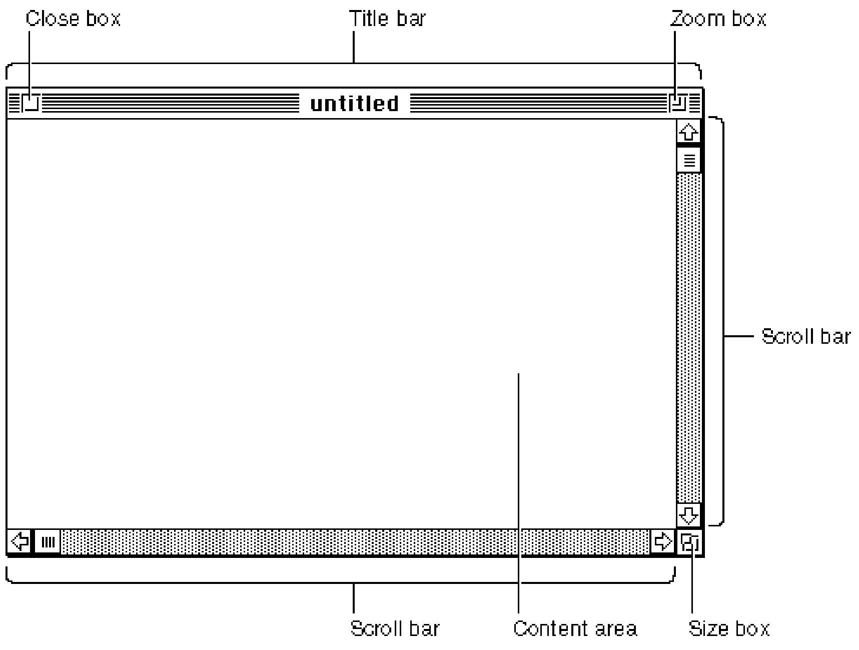
Menus are an important part of the design of a Macintosh application. Menus let users view or choose an item from a list of choices or commands that your application provides. You design your menus according to the tasks or actions your application performs. All applications should support the Apple, File, Edit, Help, Keyboard, and Application menus. Figure 1-1 on page 1-3 shows the File menu of the SurfWriter application.

System software makes it easy for you to create pull-down, hierarchical, and pop-up menus. The chapter "Menu Manager" in this book describes how to create your application's menus, set up your menu bar, display menus, and respond to the user's choice of an item from a menu.

Windows

Most applications interact with the user through windows. Figure 1-2 shows a common window and its elements. The chapter "Window Manager" in this book describes the types of windows your application can create and how to respond to user actions involving windows.

Figure 1-2 A typical window



The user typically has one or more windows on the desktop, often from a number of different applications. Although the user can have multiple windows on the desktop, only one window is the active window. The active window is the window that appears frontmost on the desktop and is identified by racing stripes in its title bar. Figure 1-2 shows an active window; in Figure 1-1 on page 1-3, the window titled SalesReport is an inactive window.

All keyboard activity is directed toward the active window. You should make sure that your application follows the human interface guidelines regarding active and inactive windows. For example, you should show the scroll bars and highlight any selection in an active window belonging to your application; you should hide the scroll bars and remove highlighting from any selection in an inactive window belonging to your application. The menu bar of your application also should always reflect the state of your application's active window—that is, your application should enable only those menu commands that pertain to the active window.

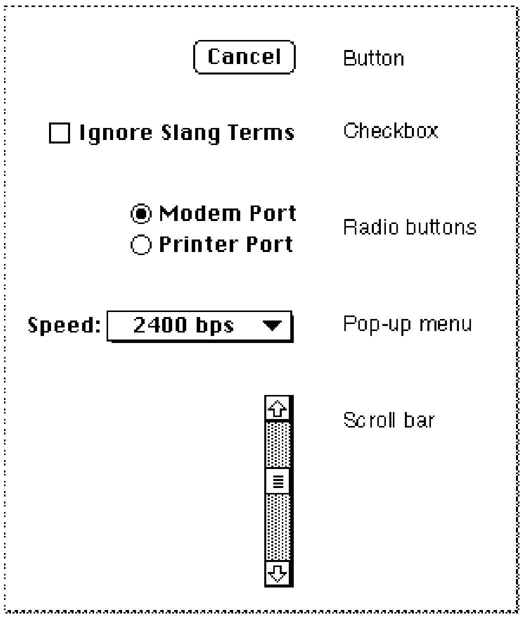
You can use system software routines to assist you when your application needs to create, move, size, zoom, or update the contents of your window. The chapter "Window Manager" in this book describes how you can accomplish these tasks.

Controls

Most windows and dialog boxes contain controls. Controls are onscreen objects that the user can manipulate with the mouse to cause an immediate action from your application or to change settings in order to modify a future action.

Buttons, checkboxes, radio buttons, pop-up menus, and scroll bars are examples of common controls used by most applications. Checkboxes, radio buttons, and pop-up menus are most often used in dialog boxes; buttons are most often used in alert boxes or dialog boxes; scroll bars are most often used in windows. Figure 1-3 illustrates these types of controls.

Figure 1-3 Common controls



A button appears as a rounded rectangle with a title centered inside. Use a button to perform an instantaneous action when the user clicks the button, such as completing operations defined by a dialog box or acknowledging an error message in an alert box.

A checkbox appears as a small square with a title beside it; the box contains an X when the setting associated with the box is on and is empty when the setting is off. Use a checkbox to indicate an option that must be either off or on.

A radio button appears as a circle with a title beside it; the circle contains a small black dot when the setting associated with the radio button is on and is empty when the setting is off. Radio buttons are similar to checkboxes in that they retain and display an on-or-off setting; however, only one radio button in a group of radio buttons should be on at any one time. You must decide how to group your radio buttons, and your application must ensure that only one radio button in a group is on.

A pop-up menu is a menu that appears in a dialog box or window. You can use pop-up menus as an alternative to radio buttons, to allow the user to select from a list of choices or settings.

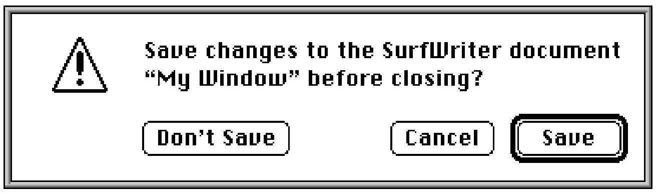
A scroll bar appears as a light gray rectangle that has scroll arrows at each end of the rectangle. A window can have a horizontal scroll bar, a vertical scroll bar, or both. You can use scroll bars to let the user change the portion of a document that the user can view within a window.

You can track and respond to user actions in controls, redraw controls, and manipulate controls using Control Manager routines. You usually use the Dialog Manager to handle most controls in dialog boxes or alert boxes for you. The chapter "Control Manager" in this book describes how to create controls (with special emphasis on creating and using scroll bars in windows), and the chapter "Dialog Manager" in this book provides additional information about how to create controls in dialog boxes and alert boxes.

Alert Boxes and Dialog Boxes

In addition to standard windows, your application typically also uses alert boxes and dialog boxes. An alert box is a window that your application displays on the screen to warn the user or to report an error to the user. An alert box typically consists of text describing the situation and buttons for the user to acknowledge or rectify the problem. Figure 1-4 shows an alert box that the SurfWriter application displays when the user attempts to close a window without saving the document. The alert box gives the user a chance to save the document before the SurfWriter application closes the window; this prevents the user from accidentally losing data.

Figure 1-4 An alert box

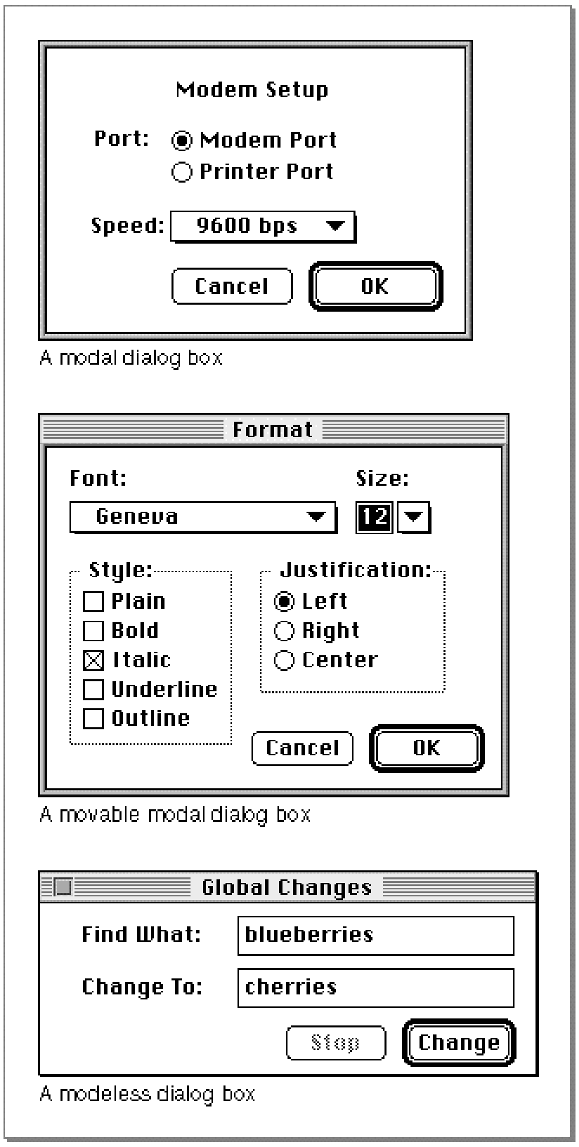


Save changes to the SurfWriter document "My Window" before closing?

A dialog box is a window that you can use for the specific purpose of soliciting additional information from the user. The Dialog Manager provides routines to help you display dialog boxes and provides standard and consistent methods of interacting with the user. Dialog boxes can contain editable text items, informative or instructional text, and controls such as buttons and checkboxes. You can create modal, movable modal, or modeless dialog boxes. Figure 1-5 shows an example of each type of dialog box.

A modal dialog box is a dialog box that puts the user in the state or “mode” of being able to work only inside the dialog box. A modal dialog box is similar in appearance to an alert box, except that a modal dialog box can contain editable text items and additional

Figure 1-5 Modal, movable modal, and modeless dialog boxes



Modem Setup

Port: [✓ Modem Port] Printer Port

Speed: 9600 bps

Cancel OK

A modal dialog box

Format

Font: Geneva

Size: 12

Style: Plain, Bold, [✓ Italic], Underline, Outline

Justification: [✓ Left] Right Center

Cancel OK

A movable modal dialog box

Global Changes

Find What: blueberries

Change To: cherries

Stop Change

A modeless dialog box

controls, such as radio buttons and pop-up menus. The user cannot move a modal dialog box, and the user can dismiss a modal dialog box only by clicking its buttons. You should use a modal dialog box only when it's essential for the user to complete an operation before performing any other work.

A movable modal dialog box is a modal dialog box with a title bar (but no close box) that allows the user to move the dialog box. The user can dismiss the dialog box only by clicking its buttons; however, when you use movable modal dialog boxes, you should allow the user to switch to another application if the user clicks in the window of another application or chooses another application from the Apple or Application menu. Use a movable modal dialog box when the user might need to move the dialog box to view other areas of the screen or when the user can switch to another application without affecting the state of your application.

A modeless dialog box is a dialog box that looks like a document window without a size box or scroll bars. A modeless dialog box does not require the user to respond before doing anything else. The user can move a modeless dialog box, move between a modeless dialog box and other windows, and close a modeless dialog box just like a document window. Whenever possible, use a modeless dialog box instead of a movable modal or modal dialog box. Use a modeless dialog box when the user can perform other operations—such as working in document windows—without dismissing the modeless dialog box.

The chapter "Dialog Manager" in this book describes in detail how you can create alert boxes and dialog boxes for your application.

Icons and Other Interactions With the Finder

Once you've designed your application, you need to create icons to represent the application and the documents it creates. The Finder displays these icons to the user. If your application appears as an item in the Apple or Application menu, the Menu Manager displays your application's icon next to its name, and the Menu Manager displays your application's icon as the title of the Application menu when your application is the active application.

The chapter "Finder Interface" in this book describes how to define and create the icons for your application and its documents. The chapter also describes how your application interacts with the Finder.

When a user opens your application or opens or prints one of its documents, the Finder uses the Process Manager to schedule your application for execution and then sets up the information your application needs to determine which, if any, files to open or print. In System 7, your application can choose to receive this information through Apple events. By supporting these and other Apple events, your application can efficiently respond to requests from the user as well as requests from other applications. See *Inside Macintosh: Interapplication Communication* for information about supporting Apple events.

Resources

Resources are basic elements of every Macintosh application. By defining descriptions of menus, windows, controls, dialog boxes, sounds, fonts, and icons in resources, you can make these and other elements easier to create and manage. Using resources also eases translation of user interface elements into other languages.

A **resource** is any data stored according to a defined structure in the resource fork of a file; the data in a resource is interpreted according to its resource type. You usually create resources using a resource compiler or resource editor. This book shows resources in Rez format; Rez is a resource compiler provided with the Macintosh Programmer's Workshop (available from APDA). You can also use other resource tools, such as ResEdit (also available from APDA), to create the resources for your application.

Most of the managers described in this book use the Resource Manager to read resources for you. For example, you can use the Menu Manager, Window Manager, Dialog Manager, and Control Manager to read descriptions of your application's menus, windows, dialog boxes, and controls from resources. These managers all interpret a resource's data accordingly once it is read into memory. While you'll typically use these managers to access resources for you, you can also directly use the Resource Manager to read and write resources.

The chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* describes the Resource Manager in detail. However, to help you understand how the Menu Manager, Window Manager, Dialog Manager, and Control Manager use resources, this section gives a brief overview of resources and provides a general introduction to the Resource Manager.

Macintosh system software treats a **file** as a named, ordered sequence of bytes stored on a Macintosh volume and divided into two forks, the data fork and the resource fork. The **data fork** contains data that usually corresponds to data created by the user; the application creating the file can store and interpret the data in the data fork in whatever manner is appropriate. The **resource fork** of a file consists of a resource map and the resources themselves.

When you write data to a file, you write to either the file's resource fork or its data fork. You typically read from and write to a file's data fork using File Manager routines and read from and write to a file's resource fork using Resource Manager routines.

You typically store as resources data that has a defined structure—such as icons and sounds—and descriptions of menus, controls, dialog boxes, and windows. When you create a resource, you assign it a resource type and resource ID. A **resource type** is a sequence of four characters that uniquely identifies a specific type of resource, and a **resource ID** identifies by number a specific resource within that type. (You can also use a resource name in place of a resource ID to identify a particular resource within a resource type.) For example, to create a description of a menu in a resource, you create a resource of type `'MENU'` and give it a resource ID or resource name that is unique from any other `'MENU'` resources that you have defined. Some resources have restrictions on the numbers you can use for resource IDs; in general, numbers 128 through 32767 are available for your use.

System software defines a number of standard resource types, such as 'ALRT', 'CNTL', 'CODE', 'DITL', 'DLOG', 'FONT', 'ICN#', 'ICON', 'MBAR', 'MENU', 'STR ', 'STR#', and 'WIND'. You can use these resource types to define their corresponding elements (for example, use a 'WIND' resource to define a window). You can also create your own resource types if your application needs resources other than the standard resource types defined by the system software.

The Resource Manager does not interpret the format of an individual resource type. When you request a resource of a particular type with a given resource ID, the Resource Manager looks for the specified resource and, if it finds it, reads the resource into memory and returns a handle to it. Your application or other system software routines can use the Resource Manager to read resources into memory. For example, when you use the Window Manager to read a description of a window from a 'WIND' resource, the Window Manager uses the Resource Manager to read the resource into memory. Once the resource is in memory, the Window Manager interprets the resource's data and creates a window with the characteristics described by the resource.

System software stores certain resources used by the system software in the System file. Although many of these resources are used only by the system software, your application can access some of these resources if needed. For example, the standard images for the I-beam and wristwatch cursors are stored as resources of type 'CURS' in the System file. You can use these resources to change the appearance of the cursor used by your application.

Occasionally you may need to write resources to the resource fork of a file. For example, if your application saves the last position and size of a window (as determined by the user), you can store this information in the resource fork of the document in a resource defined by your application. The next time the user opens the document, your application can read the location saved in this resource and position the document accordingly.

You typically store the resources specific to your application, such as descriptions of its menus, windows, controls, and dialog boxes, in the resource fork of your application. You can store resources specific to a document created by your application in the resource fork of the document file.

The resource map in the resource fork of a file contains entries that provide the location of each resource in the resource fork. When the Resource Manager opens the resource fork of a file, it reads the resource map into memory. As the Resource Manager reads resources into memory, it replaces their entries in the resource map with handles to their data in memory. The Resource Manager always searches the resource map in memory, not the resource map of the resource fork on disk, when it searches for a resource. If a requested resource is in memory, the Resource Manager uses the resource in memory; otherwise it reads the resource from the resource fork on disk into memory.

Once the Resource Manager has opened a resource fork and read its resource map into memory, it keeps the map in memory until the file is closed. You can specify that a resource be read into memory immediately when the Resource Manager opens a file's resource fork, or you can specify that the Resource Manager read it into memory only when needed. The Resource Manager stores resources from resource forks opened by your application in relocatable blocks in your application's heap. You can also specify whether the Resource Manager should purge a resource from memory in order to make room in memory for other data. If you specify that a resource is purgeable, you need to use the Resource Manager to make sure the resource is in memory before accessing it through its resource handle.

When a user opens your application, system software opens your application's resource fork. When your application opens a file, your application typically opens both the file's data fork and the file's resource fork. When your application requests a resource from the Resource Manager, the Resource Manager follows a specific search order. (If necessary, your application can change the search order using Resource Manager routines.) The Resource Manager normally looks first for the resource in the resource fork of the last file that your application opened. So, if your application has a single file open, the Resource Manager looks first in that file's resource fork. If the Resource Manager doesn't find the resource there, it continues to search each resource fork open to your application in the reverse order that the files were opened. After looking in the resource forks of files your application has opened, the Resource Manager searches your application's resource fork. If it doesn't find the resource there, it searches the resource fork of the System file.

This search path allows your application to use resources defined in the System file, to override resources defined in the System file, to share resources between files by using resources stored in your application's resource fork, and to override your application-defined resources and use resources specific to a document.

A Macintosh file always contains both a resource fork and a data fork, although one or both of those forks can be empty. Document files typically contain the document's data in the data fork and any document-specific resources—such as preference settings, window location, and the document icon—in the resource fork. The resource fork of an application typically includes resources that describe the application's menus, windows, controls, dialog boxes, and icons, as well as the code itself, which is also stored as a resource.

Whether you store data in the data fork or the resource fork of a document file depends largely on whether you can structure that data in a useful manner as a resource. For example, it's often convenient to store document-specific settings, such as the document's previous window size and location, as a resource in the document's resource fork. Data that is likely to be edited by the user is usually stored in the data fork of a document.

A resource fork can contain at most 2700 resources. The Resource Manager uses a linear search when searching a resource fork's resource types and resource IDs. In general, you should not create more than 500 resources of the same type in any one resource fork.

Inside Macintosh: More Macintosh Toolbox describes resources and the use of the Resource Manager in more detail. For information on writing data to a file's data fork, see *Inside Macintosh: Files.*

Help Balloons

Your application can provide help balloons for elements such as menus, dialog boxes, or the content area of a window. The chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox* describes how your application can provide help balloons. You can also create help balloons for some elements of your application’s interface—such as its menus—using the application BalloonWriter, which is available from APDA.

Copy and Paste

All Macintosh applications should support the copying of data from and pasting of data to the Clipboard. The chapter “Scrap Manager” in *Inside Macintosh: More Macintosh Toolbox* describes how to copy data from the Clipboard and paste data to the Clipboard by using the Scrap Manager.

Related System Software Features

In addition to the managers provided by the Macintosh Toolbox, you can also use other managers and system software routines. For example, you can use QuickDraw routines to draw the content of your application’s windows, TextEdit (in conjunction with the Dialog Manager) to handle editable text items in dialog boxes, the File Manager to read and write files, the Process Manager and Memory Manager to control various aspects of your application’s execution, and the Edition Manager and Apple Event Manager to support interapplication communication. The rest of this chapter describes some of these managers and where you can find more information about them.

Drawing on the Screen

System software routines, such as the routines provided by QuickDraw, perform all drawing on the screen. For example, your application tells QuickDraw what and where to draw, and QuickDraw does the actual drawing to the screen. The graphics routines provided by system software support quick drawing of objects such as circles, ovals, rectangles, lines, text, and pictures. See *Inside Macintosh: Imaging* for specific graphics-related information.

Handling Text

You can use the system software routines provided by TextEdit to greatly simplify basic text editing and formatting that your application would otherwise have to implement. For example, most applications use editable text items in dialog boxes; your application can use the Dialog Manager (which calls TextEdit) to automatically handle user interaction in editable text items. The Dialog Manager and your application can use

TextEdit to insert new text, delete characters that the user backspaces over, scroll text within a window, cut text, copy text, paste text, select text, and handle word wrapping. Most applications use TextEdit only for simple text editing in a dialog box and use their own techniques for handling editable text in document windows.

You should design your application so that it can handle text in more than one language or script. System software provides many routines to help you accomplish this. For example, if your application automatically displays the date in the footer of your document, you can use Text Utility routines to automatically display the date in the format common to the current script. Similarly, if your application provides a Find command, it can use Text Utility routines to search according to the word-break tables and according to the current script.

See *Inside Macintosh: Text* for information on how you can provide support for text editing in documents created by your application and for information on designing your application so that it can support text editing in more than one language or script.

Managing Files

When the user chooses the Save or Save As menu command, you usually write to a file the data that the user has entered in the active window. When the user selects a file using the Open command, you read information from a file. You can use the File Manager to read and write files. You can use the system software routines provided by the Standard File Package to present a standard and consistent interface to the user when saving and opening files. See the chapters “Introduction to File Management” and “Standard File Package” in *Inside Macintosh: Files* for information about these topics.

Allocating Memory and Launching Processes

For information about how the Process Manager launches your application, see the chapter “Process Manager” in *Inside Macintosh: Processes*. See the chapter “Introduction to Memory Management” in *Inside Macintosh: Memory* for information about how system software manages memory; how you can manage the memory in your application’s partition effectively; and how your application can allocate, release, or manipulate memory.

Creating Publishers and Subscribers

Your application should support Edition Manager features so that users can share and automatically update data between documents. See the chapter “Edition Manager” in *Inside Macintosh: Interapplication Communication* for information about supporting publish and subscribe features.

Communicating With Other Applications

System software provides various means of communication between applications. You can use Event Manager routines to communicate, in the form of high-level events, with other applications. High-level events are not required to adhere to any specific protocol, so their interpretation is defined by each application that sends or receives them. Apple events are high-level events that follow a standard defined protocol (the Apple Event Interprocess Messaging Protocol). In most cases, you should use Apple events for communication between applications. Because Apple has defined a standard set of Apple events, all applications can interpret specific Apple events in the same way and respond in an expected manner.

Both the Event Manager and Apple Event Manager rely on the services of the Program-to-Program Communications (PPC) Toolbox to actually send and receive events between applications. Your application can also directly access the PPC Toolbox if you need to get additional control or services not provided by the Event Manager or Apple Event Manager.

If your application supports publish and subscribe features, the Edition Manager sends your application Apple events to notify it when new data is available for a subscriber or to request that it create a new publisher.

For information on Apple events, publish and subscribe features, or direct access to the PPC Toolbox, see *Inside Macintosh: Interapplication Communication*.

Designing Your Application

As previously described, you'll need to make extensive use of this book and *Macintosh Human Interface Guidelines* as you begin to design your application. Once you implement the basic elements of a Macintosh application, you can begin to add features unique to your application. Once again, you'll find *Macintosh Human Interface Guidelines* and other books in the *Inside Macintosh* library valuable tools as you create applications.

C H A P T E R 2

Event Manager

Contents

This chapter describes how your application can use the Toolbox Event Manager to receive information about actions performed by the user, to receive notice of changes in the processing status of your application, and to communicate with other applications.

For example, you can retrieve information from the Toolbox Event Manager that gives your application details about whether the user has pressed a key or the mouse button, whether one of your application's windows needs updating, or whether some other hardware-related or software-related action requires a response from your application.

Your application also uses the Event Manager to support the cooperative, multitasking environment available on Macintosh computers. This environment allows users to switch between many open applications and allows other applications to receive background processing time. By using Event Manager routines, you allow the system software to coordinate the scheduling of processing time between your application and other applications.

The Event Manager and Process Manager maintain the cooperative, multitasking environment. The Process Manager coordinates the scheduling of applications, and the Event Manager communicates information about changes in your application's processing status to your application.

See the chapter "Process Manager" in *Inside Macintosh: Processes* for complete information on how the Process Manager schedules applications for execution.

You can use the Event Manager to communicate with other applications. Your application can also communicate with other applications using the services of the Apple Event Manager.

The Event Manager and Apple Event Manager routines that let your application communicate with other applications depend on the services of the Program-to-Program Communications (PPC) Toolbox. The services performed by the Event Manager and Apple Event Manager meet the needs of most applications for interapplication communication. However, to get additional control or capabilities not provided by the Event Manager or Apple Event Manager, you can choose to access the PPC Toolbox directly. The chapter "Program-to-Program Communications Toolbox" in *Inside Macintosh: Interapplication Communication* describes the PPC Toolbox routines that are available to your application.

For a comparison of the services provided by the Event Manager, Apple Event Manager, and PPC Toolbox, see *Inside Macintosh: Interapplication Communication*. For additional information about Apple events, including descriptions of how to process the required Apple events, see *Inside Macintosh: Interapplication Communication*.

This chapter describes both the Toolbox Event Manager and the Operating System Event Manager. The Operating System Event Manager maintains a queue in which it stores hardware-related occurrences that you may want your application to respond to. The Toolbox Event Manager communicates the information maintained by the Operating System Event Manager to your application. In most cases, your application needs to interact only with the Toolbox Event Manager. In this chapter, the name *Event Manager* refers to the Toolbox Event Manager.

This chapter provides a general introduction to events and then explains how you can use the Event Manager to

- receive keypresses and mouse clicks as input for your application
- receive indication that your application's windows need to be activated or updated
- allow other applications to use the available system resources when your application isn't using them
- communicate with other applications

Introduction to Events

Most Macintosh applications receive information about hardware and software occurrences that require a response from the application, through events. An **event** is the means by which the Event Manager communicates information about user actions, changes in the processing status of the application, and other occurrences that require a response from the application.

The Event Manager communicates information about events that occur through the event record. The `EventRecord` data type defines the event record. The **event record** contains information about what type of event occurred (a mouse click or keypress, for example) and contains additional information associated with the event (for example, for a keypress the Event Manager also reports which key was pressed).

Most Macintosh applications are event-driven—that is, they respond to various changes or occurrences and take action based on the nature of the event. Typically, a Macintosh application repeatedly checks to see if an event has occurred and, if so, responds to the event. If no events are pending, the application can choose to relinquish the processor for a specified amount of time or can perform other tasks before checking again to see whether an event has occurred.

Your application typically retrieves events from the Event Manager and also relinquishes processor time by using the `WaitNextEvent` function. If any events are pending for your application, the `WaitNextEvent` function returns the event to your application. If no events are pending for your application, the `WaitNextEvent` function may allocate processing time to other applications.

When multiple applications are open, the user chooses one to interact with at any given time. The active application (or **foreground process**) is the one currently interacting with the user. The foreground process displays its menu bar, and its windows are in front of the windows of all other applications. (The term **process** refers to an open application or, in some cases, an open desk accessory.)

There can be only one foreground process at any one time; however, multiple processes can exist in the background. A **background process** is a process that is not currently interacting with the user. The foreground process has first priority for accessing the CPU. Other processes can access the CPU only when the foreground process yields time to them.

By using `WaitNextEvent` to retrieve events, you allow other applications to make use of processing time that your application would otherwise not use. When your application is in the background, it in turn can receive processing time when other applications relinquish the CPU. Using `WaitNextEvent` also allows users to switch between multiple open applications.

An application that is in the background can get CPU time but can't interact with the user while it is in the background. (However, the user can choose to bring the application to the foreground—for example, by clicking in one of the application's windows.) An application can also post a notification request using the Notification Manager if the application is in the background and requires the user's attention. Any application that has the `canBackground` flag set in its size (`'SIZE'`) resource is eligible to obtain access to the CPU when it is in the background.

At any given time a process is either in the foreground or the background; a process can switch between the two states at well-defined times.

The Event Manager ensures that switching between applications occurs in a smooth fashion—by sending your application an event when it is about to be suspended and sending it an event when it has processing time again and can resume executing. The Event Manager and Process Manager coordinate this switching and scheduling of processor time among many applications.

Your application can receive various types of events: low-level events, operating-system events, and high-level events.

The Event Manager returns low-level events to your application for occurrences such as the user pressing the mouse button, releasing the mouse button, pressing a key on the keyboard, or inserting a disk. The Event Manager also returns low-level events to your application if your application needs to activate (make changes to a window based on whether it is in front or not) or update (redraw the contents of) one of its windows. When your application requests an event and there are no other events to report, the Event Manager returns a null event.

The Event Manager returns operating-system events to your application when the processing status of your application is about to change or has changed. For example, if a user brings your application to the foreground, the Process Manager sends an event through the Event Manager to your application. Some of the work of reactivating your application is done automatically, both by the Process Manager and by the Window Manager; your application must take care of any further processing needed as a result of your application being reactivated.

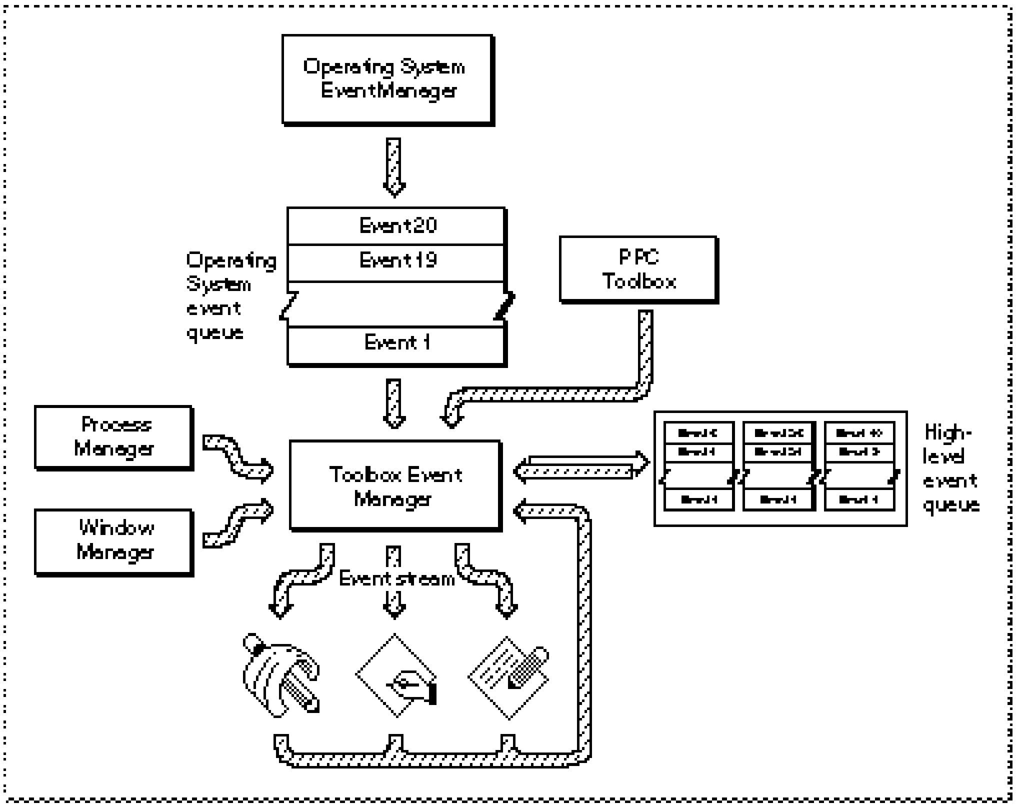
The Event Manager returns high-level events to your application as a result of communication directed to your application from another application or process.

Low-level events, except for update events and null events, are always directed to the foreground process. Operating-system events are also always directed to the foreground process. High-level events, update events, and null events can be directed to the foreground process or background processes.

You can specify which types of events you want your application to receive. You do this by specifying an event mask as a parameter to various Event Manager routines. An **event mask** allows you to mask out the events you are not interested in receiving. For example, you can accept all events except high-level events.

Events can originate from a number of different sources: the Operating System Event Manager, Window Manager, Process Manager, and PPC Toolbox. Figure 2-1 shows the relationships between the Toolbox Event Manager, other parts of the system software, and your application.

Figure 2-1 Sources of events sent to your application



The Operating System Event Manager creates and maintains a queue referred to as the **Operating System event queue.** The Operating System Event Manager detects and reports low-level hardware-related events such as mouse clicks, keypresses, and disk insertions. The Operating System Event Manager places these events in the Operating System event queue. The Toolbox Event Manager retrieves events from this event queue and returns events, one at a time at your application's request, to your application.

A maximum of 20 events can be pending in the Operating System event queue. If the Operating System event queue becomes full, the Operating System Event Manager begins to discard old events to make room for new ones as events are posted. The Operating System Event Manager always discards the oldest event in the queue when it must discard an event. However, this is not a common occurrence; your application typically processes events much faster than the user can generate them. The actual capacity of the event queue is determined by system startup information stored on the startup volume; see the chapter “File Manager” in *Inside Macintosh: Files* for system startup information.

The Event Manager can also report events from the Window Manager and Process Manager. If a window needs to be updated, activated, or deactivated, the Window Manager directs an event to the Toolbox Event Manager. Similarly, the Process Manager directs an event to the Toolbox Event Manager if the processing status of your application changes. The Toolbox Event Manager reports these events to your application.

Note

On computers running System 6, MultiFinder provides some of the capabilities supplied by the Process Manager in System 7. On computers running System 6 without MultiFinder, only a single-application environment is supported. u

Your application can use the Event Manager to send and receive high-level events. To transmit high-level events between applications, the Event Manager uses the PPC Toolbox on behalf of your application. For each open application capable of receiving high-level events, the Event Manager maintains a separate queue, referred to as the application’s **high-level event queue,** to store high-level events. The size of an application’s high-level event queue is limited only by the amount of available memory.

Your application’s event stream consists of those events that are available to your application for retrieval when it makes a request for an event. For example, when your application is in the background, its event stream can contain only update events, null events, and high-level events.

When your application asks the Event Manager for the next event, the Event Manager returns the next available event according to its priority. In general, the Event Manager returns events in this order of priority:

1. low-level events
2. operating-system events
3. high-level events

The next sections describe low-level events, operating-system events, and high-level events in greater detail.

Low-Level Events

The Event Manager uses **low-level events** to report very low-level hardware and software occurrences. Low-level events report

- actions by the user (such as pressing the mouse button, typing on the keyboard, or inserting a disk)
- changes in windows on the screen
- that the Event Manager has no other events to report

Low-level events that report actions by the user include mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events. The Event Manager reports any of these events when the user performs the action associated with each event.

Mouse-down and **mouse-up events** report that the user pressed or released the mouse button. For these events the Event Manager returns the location of the cursor at the time the mouse button was pressed or released. **Key-down** and **key-up events** report that the user pressed or released a key. **Auto-key events** report that the user has held a key down for a certain amount of time. For keyboard-related events, the Event Manager reports which key was pressed. For mouse-related and keyboard-related events, the Event Manager also reports the state of the **modifier keys** (the Option, Command, Caps Lock, Control, and Shift keys) at the time of the event.

When the user inserts a disk, the Operating System attempts to mount the volume on the disk by calling the File Manager function `PBMountVol`. The Operating System Event Manager then generates a **disk-inserted event.** If the user is interacting with a standard file dialog box, the Standard File Package intercepts the disk-inserted event and handles it. Otherwise, the event is left in the event queue for your application to retrieve. In most cases your application can handle unexpected disk-inserted events by simply checking to see if the volume was successfully mounted.

Update events and activate events are two types of low-level events that the Event Manager can report as a result of changes in the appearance of windows on the screen. For example, if a user is working with several open documents belonging to your application, you can allow the user to switch from one document to another when the user clicks in the appropriate window. You can determine whether the user clicked in another window by using the Window Manager function `FindWindow`; if the user clicked in another window, you can then use the Window Manager procedure `SelectWindow` to generate the necessary activate events. Before the Event Manager sends your application any activate events relating to this occurrence, the Window Manager does some work for you, such as unhighlighting the deactivated window and highlighting the newly activated window. At your application's next request for an event, the Event Manager returns an activate event.

An **activate event** indicates the window involved and whether the window is becoming activated or deactivated. Your application should perform any other necessary actions to complete the transformation of the window from active to inactive or vice versa. For example, when a window becomes active, your application should show any scroll bars and restore any selections.

Your application typically receives an activate event for the window being deactivated, followed by an activate event for the window becoming active at your application's next request for an event.

Note

If the user switches between your application and another application (by clicking in the window of another application, for example), your application is responsible for activating or deactivating any windows as appropriate. Your application is notified of this occurrence through operating-system events. If your application has the `acceptSuspendResumeEvents` and `doesActivateOnFGSwitch` flags set in its `'SIZE'` resource, your application is notified of the switch through an operating-system event and does not receive a separate activate event when the user switches between applications. u

The Window Manager generates **update events** to control the appearance of windows on the screen. The Window Manager keeps track of the front-to-back ordering of windows and allows windows to overlap other windows. The Window Manager coordinates the display of windows. When one window covers another window and then the user moves the first window, the Window Manager generates an update event so that the contents of the newly exposed area can be updated. The Event Manager reports update events as needed to the applications whose windows need updating. Unlike other low-level events, update events can be directed to the foreground process or background processes.

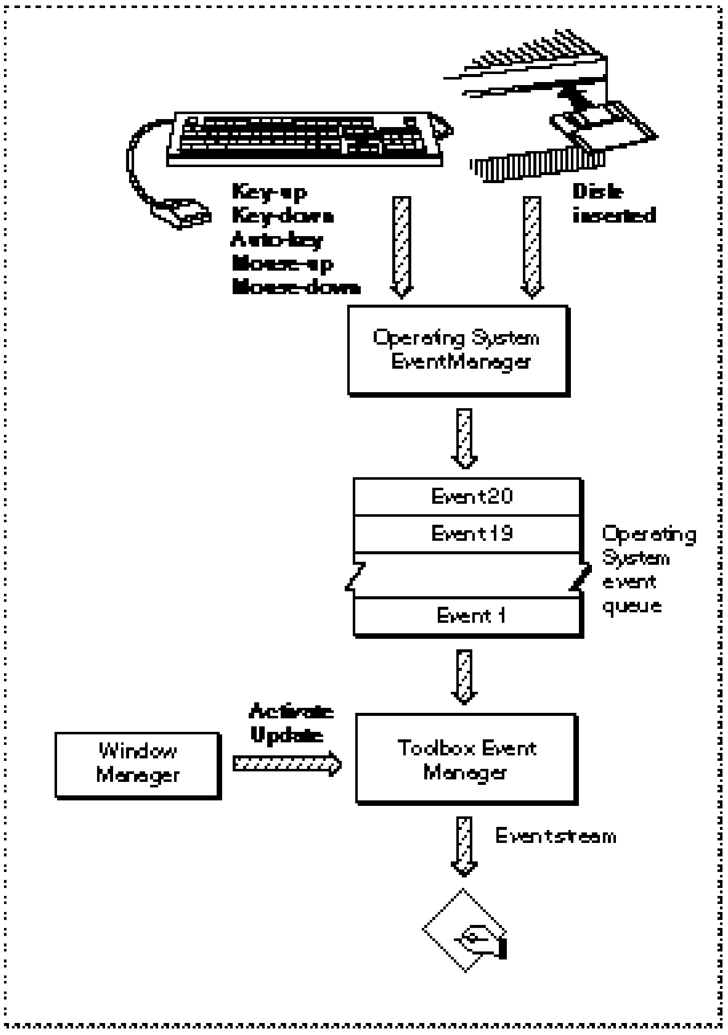
Activate and update events generated by the Window Manager are not placed into the Operating System event queue but are sent directly to the Event Manager.

The Event Manager reports a **null event** when your application requests an event and your application's event stream does not contain any of the requested event types. By using the `WaitNextEvent` function, you can yield time to other processes when null events are the only pending events for your application.

When your application receives a null event, your application can do idle processing (such as blinking the caret) if it is in the foreground or do other tasks if it is in the background. If you want your application to receive null events when it is in the background, you must set the `canBackground` flag in your application's `'SIZE'` resource. If your application does not perform any processing in response to null events when it is in the background, then set the `cannotBackground` flag. If you set the `cannotBackground` flag, the Event Manager does not report null events to your application when it is in the background. However, the Event Manager still reports update events (and high-level events if the `isHighLevelEventAware` flag is set in the `'SIZE'` resource) to your application when it is in the background regardless of how the background flag is set.

Figure 2-2 shows the various kinds of low-level events your application can receive. See "Handling Low-Level Events" beginning on page 2-32 for complete details of how your application should respond to low-level events.

Figure 2-2 Low-level events



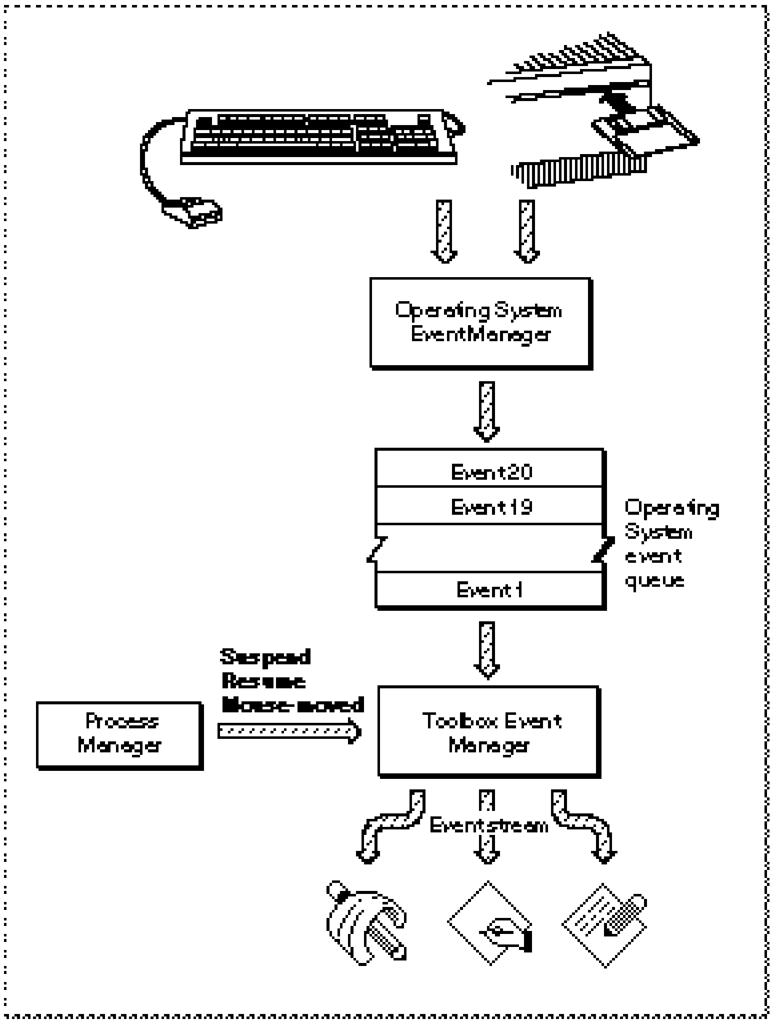
Operating-System Events

The cooperative, multitasking environment allows the user to interact with your application and with other applications. The Process Manager coordinates the scheduling of applications, and the Event Manager communicates information about changes in the operating status of applications to the applications involved.

For example, when your application is about to be switched into the background, the Event Manager sends it a **suspend event.** Then, when your application is switched back into the foreground, it receives a **resume event.** These types of events, as well as a special type of mouse event, the **mouse-moved event,** are known as **operating-system events.**

Figure 2-3 illustrates how the Event Manager helps provide this cooperative, multitasking environment. The Process Manager generates suspend, resume, and mouse-moved events, and the Event Manager reports these events to applications.

Figure 2-3 Operating-system events



Note
If your application sets the `acceptSuspendResumeEvents` and `doesActivateOnFGSwitch` flags in its `'SIZE'` resource, your application is also responsible for activating or deactivating any windows as appropriate in response to operating-system events. For maximum compatibility, your application should set these flags and handle suspend and resume events. See "The Size Resource" beginning on page 2-115 for more information on these and other flags in the `'SIZE'` resource. u

When your application receives a suspend event, it does not actually switch to the background until it makes its next request to receive events from the Event Manager. At the time that it receives the suspend event, your application should convert any private scrap into the global scrap if necessary. Your application should hide scroll bars, remove the highlighting from any selections, and hide any floating windows. If your application shows a window that displays the Clipboard contents, you should hide this window also. Then you should call `WaitNextEvent` to relinquish the CPU and allow the Operating System to schedule other processes for execution. It is important to minimize the processing you do in response to a suspend event so that the computer appears responsive to the user.

When control returns to your application, the first event it receives is a resume event. Your application should convert the global scrap back to its private scrap, if necessary. Your application should also restore any windows to the state the user left them in at the time of the previous suspend event. For example, your application should show any scroll bars, highlight any selections, and show any floating windows. See “Responding to Suspend and Resume Events” beginning on page 2-60 for complete details of how your application should respond to these events.

The events that your application can receive in the background are update, null, and high-level events. When your application is in the background, it should not perform any processing that would make the foreground process appear unresponsive to the user. When receiving events in the background, your application should perform any needed action in response to an event and then quickly return.

Your application should never interact with the user when it is in the background. If you need to notify the user of some special occurrence while your application is executing in the background, you should use the Notification Manager to queue a notification request. You should not attempt to display an alert box while your application is in the background. Instead, your application can specify that the Notification Manager play a sound, display an alert box, cause a small icon representing your application to blink in alternation with the Application menu icon, display a diamond next to your application’s name in the Application menu, or put a combination of these actions into effect.

These actions are designed to alert the user that another application needs the user’s attention. By using the Notification Manager you help maintain the user interface principle of giving the user control, as the user can choose to bring the application requesting attention to the foreground at the user’s convenience. See the chapter “Notification Manager” in *Inside Macintosh: Processes* for examples of how to post notification requests.

Another kind of operating-system event is the mouse-moved event. You can request that the Event Manager send your application a mouse-moved event whenever the cursor is outside of a region that you specify to the `WaitNextEvent` function. For example, you can use mouse-moved events as a convenient way for your application to change the appearance of the cursor as the user moves the cursor from the text area of a document to the scroll bar. See “Responding to Mouse-Moved Events” beginning on page 2-62 for detailed examples.

High-Level Events

The Event Manager provides routines that let applications communicate with each other by exchanging high-level events. A **high-level event** is an event that your application can send to another application to give it some information, to receive some information from it, or to have it perform some action.

For example, your application can send a high-level event to another application instructing that application to perform a specific action, such as adding a row to a spreadsheet or changing the font size of a paragraph. Your application can also send a high-level event to another application requesting information from that application—for example, requesting a dictionary application to return the definition of a particular word. When you send a high-level event to another application, you can also include additional information or commands in an optional data buffer. For example, your application can use a high-level event to send a list of new words and definitions to a dictionary application.

Note

High-level events are available only in system software version 7.0 or later. u

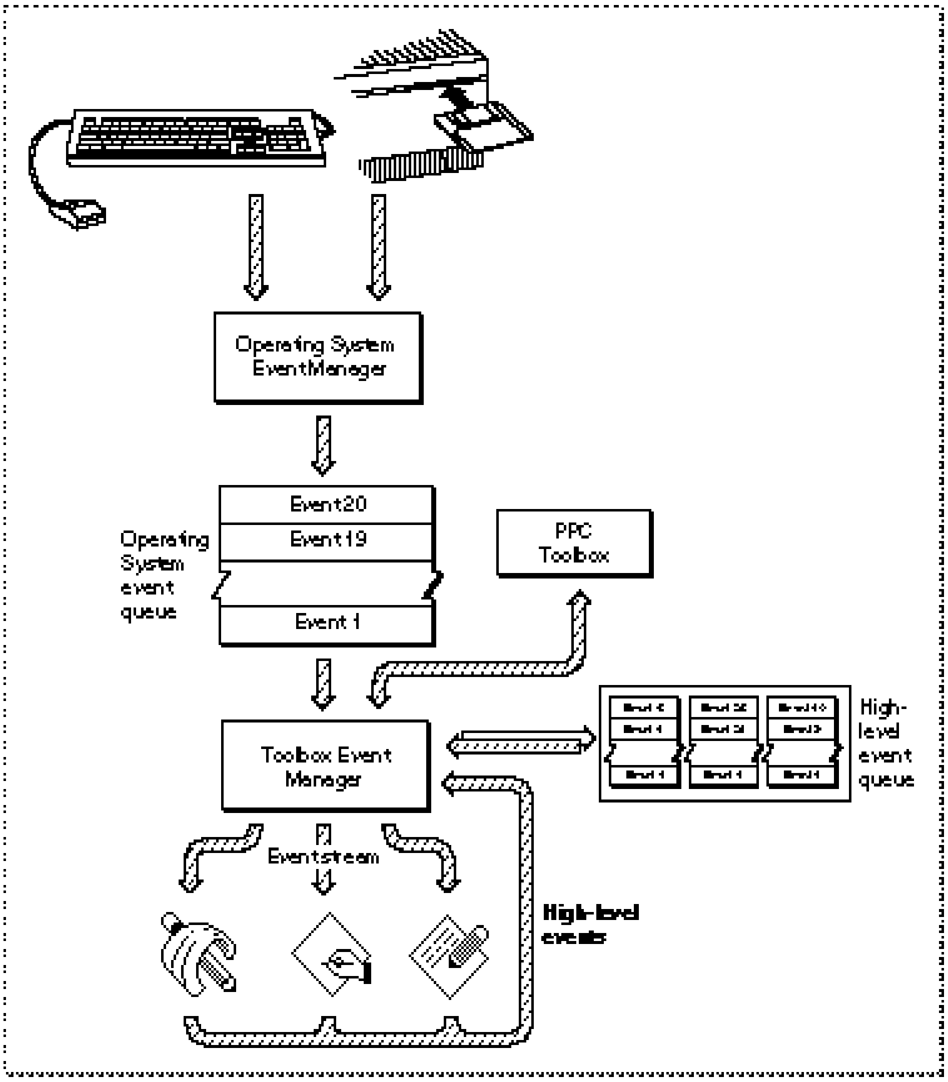
Figure 2-4 on the next page shows three different applications communicating with each other by sending and receiving high-level events. The Event Manager uses the PPC Toolbox to transmit high-level events. The Event Manager maintains a high-level event queue for each application that has identified itself as capable of receiving high-level events. The high-level event queues are limited in size only by available memory.

For effective communication between applications, your application must define the set of high-level events it responds to and let other applications know the events it accepts. By implementing the capabilities to send events to and receive events from other applications, you allow other applications to interact with your application and provide enhanced capabilities to your users.

Generally, there is no restriction on the type of processing that one application can request from another by sending it a high-level event. For a high-level event sent by one application to be understood by another application, however, the sender and receiver must agree on a protocol, that is, on the way the event is to be interpreted. **Apple events** are high-level events whose structure and interpretation are determined by the Apple Event Interprocess Messaging Protocol (AEIMP).

Your application should support the required Apple events, as described in *Inside Macintosh: Interapplication Communication*. The Finder uses the required Apple events to provide your application with information when it is opened and to give it the names of documents to open or print when the user opens or prints documents from the Finder.

Figure 2-4 High-level events



In addition, you may want your application to support other common Apple events. For example, the Edition Manager uses Apple events to communicate information about document sections among the various applications that may publish sections or subscribe to them. The Edition Manager sends the appropriate Apple events to applications that want to maintain up-to-date subscriber sections within their documents. If a user alters a section of a document that has previously been published and updates the edition, the Edition Manager might post an Apple event to the application indicating that a new edition is available. The application receiving the Apple event can then update the subscriber or ignore the information, as the user dictates. For complete information on responding to Apple events sent by the Edition Manager, see the chapter "Edition Manager" in *Inside Macintosh: Interapplication Communication*.

To ensure compatibility and smooth interaction with other Macintosh applications, you should use the Apple event protocol for high-level events whenever possible. You should define new protocols only if your application must communicate with applications on other computers that use different protocols or if your application has other special needs. For complete information about Apple events and about implementing the required set of Apple events, see *Inside Macintosh: Interapplication Communication*.

Note
All Macintosh system software that sends or receives high-level events uses the Apple events protocol. u

Priority of Events

Each type of event has a certain priority. The Event Manager returns events in this order of priority:

1. activate events
2. mouse-down, mouse-up, key-down, key-up, and disk-inserted events in FIFO (first-in, first-out) order
3. auto-key events
4. update events (in front-to-back order of windows)
5. operating-system events (suspend, resume, mouse-moved)
6. high-level events
7. null events

Several of the Event Manager routines can be restricted to operate on one or more specific types of events. You do this by disabling (or "masking out") the events you are not interested in receiving. See "Setting the Event Mask" beginning on page 2-26 for details about specifying the types of events you wish to receive.

Switching Contexts

Processes running in the background receive processing time when the foreground process makes an event call (that is, calls `WaitNextEvent` or `EventAvail`) and there are no events pending for that foreground process. A process running in the background should relinquish the CPU regularly to ensure a timely return to the foreground process when necessary.

In System 7 (or with MultiFinder in earlier versions), the available processing time is distributed among multiple processes through a procedure known as *context switching* (or just *switching*). All switching occurs at a well-defined time, namely, when an application calls `WaitNextEvent`. When a context switch occurs, the Process Manager allocates processing time to a process other than the one that had been receiving processing time. Two types of context switching may occur: major and minor.

A **major switch** is a complete context switch: an application's windows are moved from the back to the front, or vice versa. In a major switch, two applications are involved, the one being switched to the foreground and the one being switched to the background. The Process Manager switches the A5 worlds of both applications, as well as the relevant low-memory environments. If those applications receive suspend and resume events, they are so notified at the time that a major switch occurs.

A **minor switch** occurs when the Process Manager gives time to a background process without bringing the background process to the front. The two processes involved in a minor switch can be two background processes or a foreground process and a background process. As in a major switch, the Process Manager switches the A5 worlds and the low-memory environments of the two processes. However, the order of windows is not switched, and neither process receives either suspend or resume events.

When the frontmost window is an alert box or a modal dialog box, major switching does not occur, although minor switching can. To determine whether major switching can occur, the Operating System checks (among other things) to see if the window definition procedure of the frontmost window is `dBoxProc`, because the type `dBoxProc` is specifically reserved for alert boxes and modal dialog boxes. (If the frontmost window is a movable modal dialog box, major switching can still occur.)

Note

Your application can also get switched out if it calls a system software routine that makes an event call. For example, when your application calls `ModalDialog`, a minor switch can occur. u

Your application can receive processing time and perform tasks in the background, but your application should not interact with the user or perform tasks that would slow down the responsiveness of the foreground process.

Your application indicates scheduling options to the Operating System, such as whether the application can use null-event processing time when in the background, whether it can accept suspend and resume events, and so forth, by setting flags in its **size** (`'SIZE'`) **resource.** Every application executing in System 7, as well as every application executing in System 6 with MultiFinder, should contain a `'SIZE'` resource. See "Creating a Size Resource" beginning on page 2-30 for details on how to specify this information.

About the Event Manager

The Toolbox Event Manager provides routines that communicate information about actions performed by the user and give notice of changes in the processing status of your application. The Event Manager also provides routines that your application can use to communicate with other applications. You can control the scheduling of your application for execution by using the Event Manager.

The rest of this chapter explains

- how to structure your main event loop to receive and process events
- how to create a 'SIZE' resource to specify your application's memory requirements and scheduling options
- how to respond to most types of events
- how to receive and process high-level events
- how to send high-level events to other applications

Using the Event Manager

You can use the Event Manager to receive information about hardware-related events, about changes in the appearance of your application's windows, or about changes in the operating status of your application. You can also use the Event Manager to communicate directly with other applications. This communication can include sending events to other applications, receiving events from other applications, and searching for specific events from other applications.

Your application can both send and receive high-level events, but it generally only receives low-level events and should not send them. Your application receives low-level events, operating-system events, and high-level events in the same way, which is by asking the Event Manager for the next available event. If the event your application receives is a high-level event, your application might need to use another Event Manager or Apple Event Manager routine to retrieve an optional data buffer and additional information accompanying that event.

Before using the Event Manager, you can use the `Gestalt` function to determine if certain features of the Event Manager are available. See the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities* for information on the `Gestalt` function.

If your application sends or receives high-level events, you should use the `Gestalt` function with the `gestaltPPCToolboxAttr` selector to determine whether the PPC Toolbox is present. Use the `Gestalt` function with the `gestaltOSAttr` selector to see if the Process Manager is available. If the PPC Toolbox and the Process Manager are present, then the system software provides support for high-level events.

If your application sends or receives Apple events, use the `Gestalt` function with the `gestaltAppleEventsAttr` selector to determine whether the Apple Event Manager is available.

Your application needs to initialize QuickDraw, the Font Manager, and the Window Manager before using the Event Manager. Your application can accomplish this initialization by using the `InitGraf`, `InitFonts`, and `InitWindows` procedures.

When your application starts, you can call the `FlushEvents` procedure to empty the Operating System event queue of any low-level events left unprocessed by another application. For example, you might want to remove any mouse-down events or keyboard events that the user might have entered while the Finder launched your application.

This section shows how to retrieve events from the Event Manager, how to mask out unwanted events, how to specify memory and scheduling options for your application, and how to handle each type of event received from the Event Manager.

Obtaining Information About Events

You get information about events through the event record. The `EventRecord` data type defines the event record and has this structure:

```
TYPE  EventRecord =
      RECORD
         what:       Integer;       {event code}
         message:    LongInt;       {event message}
         when:       LongInt;       {ticks since startup}
         where:      Point;         {mouse location}
         modifiers:  Integer;       {modifier flags}
      END;
```

Field descriptions

`what` The `what` field indicates the type of event received. The type of event can be identified by these constants:

```
CONST
nullEvent        = 0; {no other pending events}
mouseDown        = 1; {mouse button pressed}
mouseUp          = 2; {mouse button released}
keyDown          = 3; {key pressed}
keyUp            = 4; {key released}
autoKey          = 5; {key repeatedly held down}
updateEvt        = 6; {window needs updating}
diskEvt          = 7; {disk inserted}
activateEvt      = 8; {activate/deactivate window}
osEvt            = 15;{operating-system event-- }
                        { resume, suspend, or }
                        { mouse-moved}
kHighLevelEvent  = 23;{high-level event}
```

`message`	The `message` field contains additional information associated with the event. The interpretation of this information depends on the event type. The contents of the `message` field for each event type are summarized here:

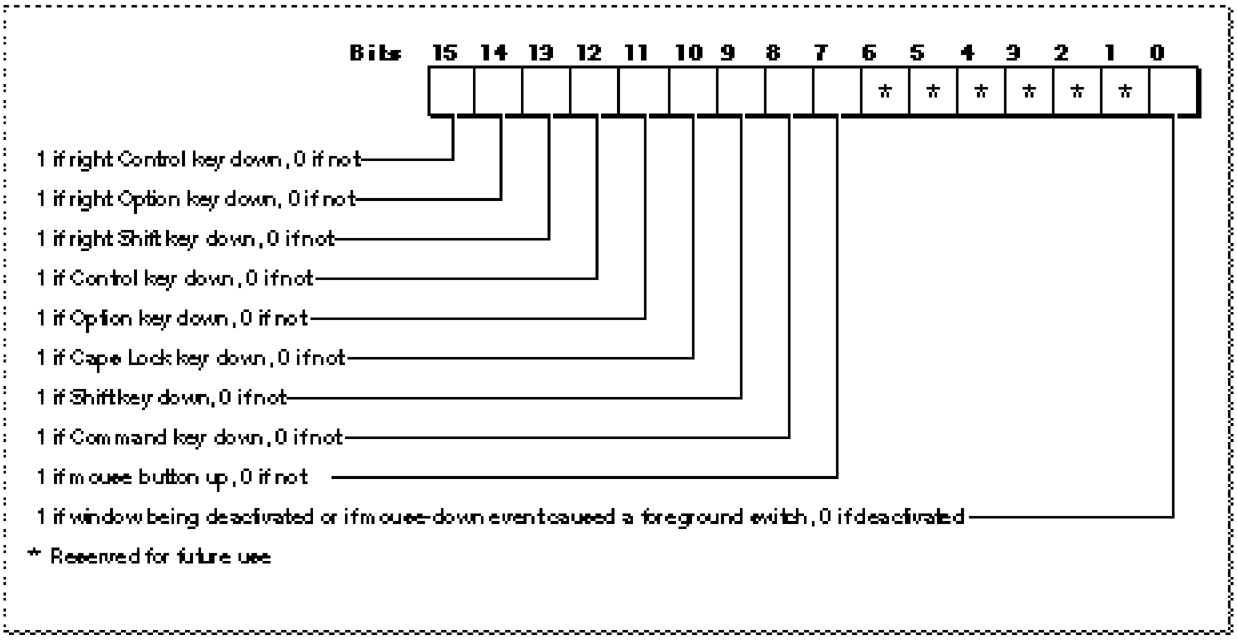
Event type	Event message
null, mouse-up, mouse-down	Undefined.
key-up, key-down, auto-key	Character code and virtual key code in low-order word. For Apple Desktop Bus (ADB) keyboards, the low byte of the high-order word contains the ADB address of the keyboard where the keyboard event occurred. The high byte of the high-order word is reserved.
update, activate	Pointer to the window to update, activate, or deactivate.
disk-inserted	Drive number in low-order word, File Manager result code in high-order word.
resume	The `suspendResumeMessage` constant in bits 24–31 and a 1 in bit 0 to indicate the event is a resume event. Bit 1 contains either a 1 or a 0 to indicate if Clipboard conversion is required, and bits 2–23 are reserved.
suspend	The `suspendResumeMessage` constant in bits 24–31 and a 0 in bit 0 to indicate the event is a suspend event. Bit 1 is undefined, and bits 2–23 are reserved.
mouse-moved	The `mouseMovedMessage` constant in bits 24–31. Bits 2–23 are reserved, and bit 0 and bit 1 are undefined.
high-level	Class of events to which the high-level event belongs. The `message` and `where` fields of a high-level event define the specific type of high-level event received.

`when`	The `when` field indicates the time when the event was posted (in ticks since system startup). When needed, you can use the `when` field to compare how much time has elapsed between successive mouse events.
`where`	For low-level events and operating-system events, the `where` field contains the location of the cursor at the time the event was posted (in global coordinates). For high-level events, the `where` field contains a second event specifier, the event ID. The event ID defines the particular type of event within the class of events defined by the `message` field of the high-level event. For high-level events, you should interpret the `where` field as having the data type `OSType`, not `Point`.

`modifiers` The `modifiers` field contains information about the state of the modifier keys and the mouse button at the time the event was posted. For activate events, this field also indicates whether the window should be activated or deactivated. In System 7 it also indicates whether a mouse-down event caused your application to switch to the foreground.

Each of the modifier keys is represented by a specific bit in the `modifiers` field of the event record. Figure 2-5 shows how to interpret the `modifiers` field. You can examine the `modifiers` field of the event record to determine which, if any, of the modifier keys were pressed at the time of the event. The modifier keys include the Option, Command, Caps Lock, Control, and Shift keys. If your application attaches special meaning to any of these keys in combination with other keys or when the mouse button is down, you can test the state of the `modifiers` field to determine the action your application should take. For example, you can use this information to determine whether the user pressed the Command key and another key at the same time to make a menu selection.

Figure 2-5 The `modifiers` field of the event record


Bits 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
* * * * * *
1 if right Control key down, 0 if not
1 if right Option key down, 0 if not
1 if right Shift key down, 0 if not
1 if Control key down, 0 if not
1 if Option key down, 0 if not
1 if Caps Lock key down, 0 if not
1 if Shift key down, 0 if not
1 if Command key down, 0 if not
1 if mouse button up, 0 if not
1 if window being deactivated or if mouse-down event caused a foreground switch, 0 if deactivated
* Reserved for future use

Bit 0 in the `modifiers` field gives additional information that is valid only if the event is an activate event or a mouse-down event.

For activate events, the value of bit 0 is 1 if the window pointed to by the event message should be activated, and the value is 0 if the window should be deactivated.

For mouse-down events in System 7, bit 0 indicates whether a mouse-down event caused your application to switch to the foreground. If so, bit 0 contains 1; otherwise, it contains 0.

You can also use these constants as masks to test the setting of various bits in the modifiers field:

```
CONST activeFlag  = 1;     {set if window being activated or if }
                           { mouse-down event caused fgnd switch}
      btnState    = 128;   {set if mouse button up}
      cmdKey      = 256;   {set if Command key down}
      shiftKey    = 512;   {set if Shift key down}
      alphaLock   = 1024;  {set if Caps Lock key down}
      optionKey   = 2048;  {set if Option key down}
      controlKey  = 4096;  {set if Control key down}
```

Note that the bit giving information about the mouse button is set if the mouse button is up. The bits for the modifier keys are set if the corresponding key is down.

Some keyboards do not distinguish between the right or left Control, Shift, and Option keys; for example, the virtual key code for the right Shift key and left Shift key might be the same. For these keyboards, if the user presses the Control, Shift, or Option key, the Event Manager sets only the bits corresponding to the shiftKey, optionKey, and controlKey constants. For keyboards that do distinguish between these keys, the Event Manager sets the bits in the modifiers field to indicate whether the right or left Control, Shift, or Option keys were pressed. For example, the Event Manager sets bit 13 in the modifiers field if the user presses the right Shift key and sets bit 9 if the user presses the left Shift key. In most cases your application should not need to distinguish between the left and right Control, Shift, and Option keys.

Processing Events

Applications receive events one at a time by asking the Event Manager for the next available event. You use Event Manager routines to receive (or in the case of EventAvail, simply to look at) the next available event that is pending for your application. You supply an event record as a parameter to the Event Manager routines that retrieve events. The Event Manager fills out the event record with the relevant information about that event and returns it to your application.

Your application can use the WaitNextEvent function to retrieve events from the Event Manager. If no events are pending for your application, the WaitNextEvent function may allocate processing time to other applications. If an event is pending for your application, the WaitNextEvent function returns the next available event of a specified type and removes the returned event from your application's event stream.

The EventAvail function gets the next available event of a specified type and returns it to your application, but does not remove the event from your application's event stream. EventAvail thus allows your application to look at an event in the event stream without actually processing the event.

Note
You can also use the `GetNextEvent` function to retrieve and remove an event; however, you should use `WaitNextEvent` to provide greater support for multitasking. u

Using the WaitNextEvent Function

Your application typically calls `WaitNextEvent` repeatedly. The next section, "Writing an Event Loop," shows how to use `WaitNextEvent` with other routines to process events. This discussion focuses on the `WaitNextEvent` function itself.

The `WaitNextEvent` function requires four parameters:

- an event mask (`eventMask`)
- an event record (`theEvent`)
- a sleep value (`sleep`)
- a mouse region (`mouseRgn`)

When `WaitNextEvent` returns, the event record contains information about the retrieved event, if any.

The `eventMask` parameter specifies the events you are interested in receiving. `WaitNextEvent` returns events one at a time, in order of priority and at your application's request, according to the value you specify in the `eventMask` parameter. If your application specifies that it doesn't want to receive particular types of events, those events are not returned to your application when it makes a request for an event. However, those events are not removed from the event stream. (To remove events from the Operating System event queue, you can use the `FlushEvents` procedure with a mask specifying only those events you wish to remove from the queue.) See "Setting the Event Mask" beginning on page 2-26 for examples of how to use constants to set the value of the `eventMask` parameter.

The `sleep` parameter specifies the amount of time (in ticks) for which your application agrees to relinquish the processor if no events are pending for it. When that time expires or when an event becomes available for your application, the Process Manager schedules your application for execution. In general, you should specify a value greater than 0 in the `sleep` parameter so that other applications can receive processing time if they need it. If the user is editing text and your application needs to blink the caret at periodic intervals or uses TextEdit to blink the caret, your application should not specify a value greater than the value returned by the `GetCaretTime` function.

In the `mouseRgn` parameter you specify a screen region inside of which the Event Manager does *not* generate mouse-moved events. You should specify the region in global coordinates. If the user moves the cursor outside of this region and your application is the foreground process, the Event Manager reports mouse-moved events. Your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event; otherwise it will continue to receive mouse-moved events as long as the cursor is outside of the original region. If you pass an empty region or a `NIL` region handle, the Event Manager does not return mouse-moved events. You can use the mouseRgn parameter as a convenient way to change the shape of the cursor—for example, when the user moves the cursor from the content area of a window to the scroll bar. See "Responding to Mouse-Moved Events" beginning on page 2-62 for information on how to set and change the mouseRgn parameter.

Listing 2-1 shows an example of using the WaitNextEvent function.

Listing 2-1 Using the WaitNextEvent function

```
VAR
   eventMask:     Integer;
   event:         EventRecord;
   cursorRgn:     RgnHandle;
   mySleep:       LongInt;
   gotEvent:      Boolean;

   eventMask := everyEvent;   {accept all events}
   mySleep := MyGetSleep;     {set an appropriate sleep value}
   cursorRgn := MyGetRgn;     {set the region as appropriate}
   gotEvent := WaitNextEvent(eventMask,event,mySleep,cursorRgn);
```

The code in Listing 2-1 specifies that WaitNextEvent should return the next pending event of any kind, give up the processor if no events are pending, and return a mouse-moved event if the user moves the cursor out of the specified region.

The WaitNextEvent function returns after retrieving an event or after the time specified in the sleep parameter has expired. If there are no events of the types specified by the eventMask parameter (other than null events) pending for your application, and the time specified in the sleep parameter has not expired, WaitNextEvent may allocate processing time to background processes. Once an event for your application occurs or the time specified in the sleep parameter expires, your application receives processing time again.

WaitNextEvent returns a function result of TRUE if it has retrieved any event other than a null event. If there are no events of the types specified by the eventMask parameter (other than null events) pending for the application, WaitNextEvent returns FALSE.

Before returning an event to your application, WaitNextEvent performs other processing and may intercept the event. The WaitNextEvent function:

- Calls the Operating System Event Manager function SystemEvent to determine whether the event should be handled by your application or the Operating System. For example, if the event is a Command–Shift–number key sequence, the Event Manager intercepts the event and calls the corresponding 'FKEY' resource to perform the associated action.
- Makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

- Calls the `SystemTask` procedure, which gives time to each open desk accessory or device driver to perform any periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

In System 7, the `WaitNextEvent` function reports a suspend event to your application when

- your application is in the foreground and the user opens a desk accessory or other item from the Apple menu,
- the user clicks in the window belonging to a desk accessory or another application, or
- the user chooses another process from the Application menu.

After your application is switched out, the Event Manager directs events (other than events your application can receive in the background) to the newly activated process until the user switches back to your application or another application.

Writing an Event Loop

In applications that are event-driven (that is, applications that decide what to do at any time by receiving and responding to events), you can obtain information about pending events by calling Event Manager routines. Since you call these routines repeatedly, the section of code in which you request events from the Event Manager usually takes the form of a loop; this section of code is called the *event loop*.

Listing 2-2 shows a simple event loop (an application-defined procedure called `MyEventLoop`) for an application running in System 7.

Listing 2-2 An event loop

```
PROCEDURE MyEventLoop;
VAR
   cursorRgn:     RgnHandle;
   gotEvent:      Boolean;
   event:         EventRecord;
BEGIN
   cursorRgn := NewRgn; {pass an empty region the first time thru}
   REPEAT
      gotEvent := WaitNextEvent(everyEvent, event, MyGetSleep,
                                 cursorRgn);
      IF (event.what <> kHighLevelEvent) AND (NOT gInBackground)
         THEN MyAdjustCursor(event.where, cursorRgn);
      IF gotEvent THEN  {the event isn't a null event, }
         DoEvent(event) { so handle it}
      ELSE              {no event (other than null) to handle }
         DoIdle(event); { right now, so do idle processing}
   UNTIL gDone;         {loop until user quits}
END;
```

The MyEventLoop procedure repeatedly uses WaitNextEvent to retrieve events. The WaitNextEvent function returns a Boolean value of FALSE if there are no events of the specified types other than null events pending for the application. WaitNextEvent returns TRUE if it has retrieved any event other than a null event.

After WaitNextEvent returns, the MyEventLoop procedure first calls an application-defined routine, MyAdjustCursor, to adjust the cursor as necessary. You usually adjust the cursor in response to mouse-moved events, and often in response to other events as well. This code adjusts the cursor once every time through the event loop, when the application receives any event other than a high-level event. The code does not adjust the cursor if the event is a high-level event, because the where field of a high-level event contains the event ID, not the location of the cursor. The code also does not adjust the cursor if this application is in the background, as the foreground process is responsible for setting the appearance of the cursor.

If WaitNextEvent retrieved any event other than a null event, the event loop calls DoEvent, an application-defined procedure, to process the event. Otherwise, the procedure calls an application-defined idling procedure, DoIdle.

Note

If your application uses modeless dialog boxes, you need to appropriately handle events in them. You can choose to handle events for modeless dialog boxes using the same routines that you use to handle events in other windows; this is the approach used throughout this chapter. Alternatively, you can choose to call the IsDialogEvent function in your event loop. See "Handling Events in a Dialog Box" on page 2-29 for information on handling events in alert boxes, modal dialog boxes, movable modal dialog boxes, and modeless dialog boxes. For additional information on dialog boxes, see the chapter "Dialog Manager" in this book. u

If you intend to design your application to run in either a single-application environment (such as System 6 without MultiFinder) or a multiple-application environment, the very beginning of your event loop should test to make sure the WaitNextEvent function is available. If WaitNextEvent is not available, your code should use GetNextEvent to retrieve events. If your code uses GetNextEvent, it should also call SystemTask to allow desk accessories to perform periodic actions. However, your code should always use WaitNextEvent if it is available, rather than GetNextEvent. If your application calls WaitNextEvent, it should not call the SystemTask procedure.

The event loop shown in Listing 2-2 calls an application-defined procedure, DoEvent, to determine what kind of event the call to WaitNextEvent retrieved. Listing 2-3 defines a simple DoEvent procedure. The DoEvent procedure examines the value of the what field of the event record to determine the type of event received and then calls an appropriate application-defined routine to further process the event.

Listing 2-3 Processing events

```
PROCEDURE DoEvent (event: EventRecord);
VAR
   window:     WindowPtr;
   activate:   Boolean;
BEGIN
   CASE event.what OF
      mouseDown:
         DoMouseDown(event);
      mouseUp:
         DoMouseUp(event);
      keyDown, autoKey:
         DoKeyDown(event);
      activateEvt:
         BEGIN
            window := WindowPtr(event.message);
            activate := BAnd(event.modifiers, activeFlag) <> 0;
            DoActivate(window, activate, event);
         END;
      updateEvt:
         DoUpdate(WindowPtr(event.message));
      diskEvt:
         DoDiskEvent(event);
      osEvt:
         DoOSEvent(event);
      kHighLevelEvent:
         DoHighLevelEvent(event);
   END; {of case}
END;
```

The next sections describe how to set the event mask, handle events in dialog boxes, and create your application's `'SIZE'` resource. Following sections show code that can handle each kind of event.

Setting the Event Mask

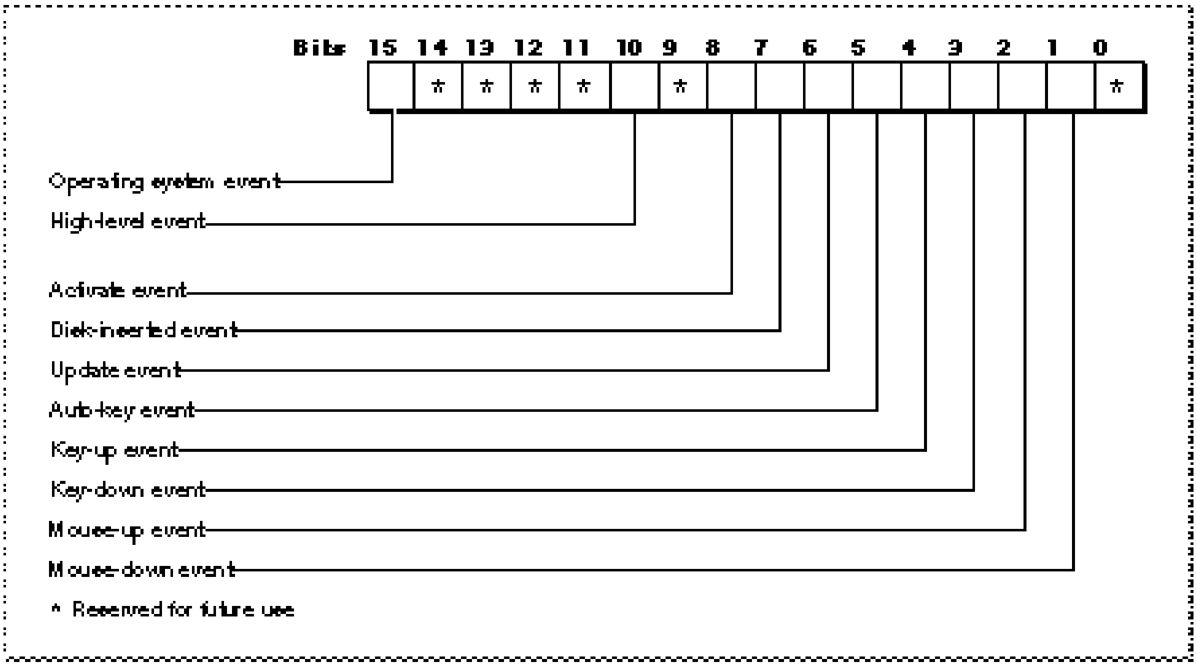
Several of the Event Manager routines can be restricted to operate on a specific event type or group of types. You do this by specifying the event types you want your application to receive, thereby disabling (or "masking out") the events you are not interested in receiving. To specify which event types an Event Manager routine governs, you supply a parameter known as an event mask.

The **event mask** is an integer with one bit position for each event type. If the bit representing a particular event type is set, then the Event Manager returns events of that type. If the bit is set to 0, the Event Manager does not return events of that type. To accept all types of events, set every bit of the event mask to 1. You can do this using the constant `everyEvent`.

```
CONST everyEvent           =  -1;      {every event}
```

Figure 2-6 shows the bits corresponding to each event type in the event mask.

Figure 2-6 The event mask



You can use these constants when referring to the bits in the event mask that correspond to each individual event type:

```
CONST mDownMask           =  2;      {mouse-down event       (bit 1)}
      mUpMask             =  4;      {mouse-up event         (bit 2)}
      keyDownMask         =  8;      {key-down event         (bit 3)}
      keyUpMask           =  16;     {key-up event           (bit 4)}
      autoKeyMask         =  32;     {auto-key event         (bit 5)}
      updateMask          =  64;     {update event           (bit 6)}
      diskMask            =  128;    {disk-inserted event    (bit 7)}
      activMask           =  256;    {activate event         (bit 8)}
      highLevelEventMask  =  1024;   {high-level event       (bit 10)}
      osMask              =  -32768; {operating-system event (bit 15)}
```

You can select any subset of events by adding or subtracting these constants. For example, you can use this code to accept only high-level events and mouse-down events and mask out all other events:

```
myErr := WaitNextEvent(highLevelEventMask + mDownMask, myEvent,
                         mySleep, myMRgnHnd);
```

The everyEvent constant indicates that you wish to receive every type of event. To accept all events except mouse-up events, you can use the code:

```
myErr := WaitNextEvent(everyEvent - mUpMask, myEvent, mySleep,
                         myMRgnHnd);
```

Masking out specific types of events does not remove those events from the event stream. If a type of event is masked out, the Event Manager simply ignores it when reporting events from the event stream. Note that you cannot mask out null events by setting the event mask. The Event Manager always returns a null event if no other events are pending. However, if you do not want the Event Manager to report null events to your application when it is in the background, you can set the cannotBackground flag in your application's 'SIZE' resource.

In most cases you should always use everyEvent as your event mask. The user expects most applications to respond to keyboard, mouse, update, and other events.

The types of events returned to your application are also affected by the system event mask. The Event Manager maintains a system event mask for each application. The system event mask controls which low-level event types get posted in the Operating System event queue. The Event Manager uses the system event mask of the current process (the process that is currently executing and the process associated with the CurrentA5 global variable) when determining which low-level events to post in the Operating System event queue. The system event mask is an integer with 1 bit for each corresponding low-level event type. These constants refer to the bits that represent the corresponding low-level event types in the system event mask:

```
CONST mDownMask          =  2;      {mouse-down    (bit 1)}
      mUpMask            =  4;      {mouse-up      (bit 2)}
      keyDownMask        =  8;      {key-down      (bit 3)}
      keyUpMask          =  16;     {key-up        (bit 4)}
      autoKeyMask        =  32;     {auto-key      (bit 5)}
      diskMask           =  128;    {disk-inserted (bit 7)}
```

When a low-level event (other than an update or activate event) occurs, the Operating System Event Manager posts the low-level event in the Operating System event queue only if the bit corresponding to the low-level event type is set in the system event mask of the current process. When your application starts, the Operating System initializes the system event mask of your application to post mouse-up, mouse-down, key-down, auto-key, and disk-inserted events in the Operating System event queue. Thus, the system event mask has this initial setting:

```
systemEventMask := everyEvent - keyUpMask;
```

Your application should not change the system event mask except to enable key-up events if your application needs to respond to key-up events. (Most applications ignore key-up events.) If your application needs to receive key-up events, you can change the system event mask using the Operating System Event Manager procedure `SetEventMask`. Note that your application cannot rely on receiving key-up events when it is not the current process. For example, if your application is the foreground (and current) process and a minor switch occurs, the Event Manager uses the system event mask of the background process (now the current process) when posting low-level event types. When your application becomes the current process again, the Event Manager uses the system event mask of your application when posting low-level events.

Handling Events in a Dialog Box

If your application uses alert boxes, modal dialog boxes, movable modal dialog boxes, or modeless dialog boxes, you need to make sure your application handles events for them appropriately.

To display and handle events in alert boxes, you use the Dialog Manager functions `Alert`, `NoteAlert`, `CautionAlert`, and `StopAlert`. The Dialog Manager handles all of the events generated by the user until the user clicks a button (typically the OK or Cancel button). When the user clicks the OK or Cancel button, the alert box functions highlight the button that was clicked, close the alert box, and report the user's selection to your application. Your application is responsible for performing the appropriate action associated with that button.

For modal dialog boxes, you can use the Dialog Manager procedure `ModalDialog`. The Dialog Manager handles most of the user interaction until the user selects an item. The `ModalDialog` procedure then reports that the user selected an enabled item, and your application is responsible for performing the action associated with that item. Your application typically calls `ModalDialog` repeatedly, responding to clicks on enabled items as reported by `ModalDialog`, until the user selects OK or Cancel.

For alert boxes and modal dialog boxes, you should also supply an event filter function as one of the parameters to the alert box functions or `ModalDialog` procedure. As the user interacts with the alert or modal dialog box, these functions pass events to your event filter function before handling each event. Your event filter function can handle any events not handled by the Dialog Manager or, if necessary, can choose to handle events normally handled by the Dialog Manager. For more information on filter functions for alert and dialog boxes, see the chapter "Dialog Manager" in this book.

To handle events in movable modal dialog boxes, you can use the Dialog Manager functions `IsDialogEvent` and `DialogSelect` or you can use other Toolbox routines to handle events without using the Dialog Manager.

For modeless dialog boxes, you can choose to handle events in them using an approach similar to the one you use to handle events in other windows; that is, when you receive an event, you first determine the type of event that occurred and then take the appropriate action based on the type of window that is in front. If a modeless dialog box is in front, you can provide code that takes any actions specific to that modeless dialog box and call the `DialogSelect` function to handle any events that your code doesn't handle. This is the approach used throughout this chapter. Alternatively, you can choose to call the `IsDialogEvent` function in your event loop. If you do this, you can use the `IsDialogEvent` function to determine whether the event involves a modeless dialog box that belongs to your application. If the event involves a modeless dialog box (including null events) and a modeless dialog box is active, `IsDialogEvent` returns TRUE. Otherwise, `IsDialogEvent` returns FALSE.

If `IsDialogEvent` returns TRUE, your application can check to see what type of event occurred and, depending on the type of event, it can choose to handle the event itself.

Regardless of the approach you use, if your application chooses not to handle the event, it should call `DialogSelect`. The `DialogSelect` function handles events for modeless dialog boxes (including null events). It also blinks the caret in editable text items when necessary.

For more information on the `DialogSelect` function and events in dialog boxes, see the chapter "Dialog Manager" in this book.

Creating a Size Resource

Your application should include a size (`'SIZE'`) resource. You use a `'SIZE'` resource to inform the Operating System about the memory size requirements for your application so that the Operating System can set up a partition of the appropriate size for your application. You also use the `'SIZE'` resource to indicate certain scheduling options to the Operating System, such as whether your application can accept suspend and resume events.

You can also specify additional information in the `'SIZE'` resource in System 7, indicating whether your application is 32-bit clean, whether your application supports stationery documents, whether your application uses TextEdit's inline input services, whether your application wishes to receive notification of the termination of any applications it has launched, and whether your application wishes to receive high-level events.

A `'SIZE'` resource consists of a 16-bit flags field, followed by two 32-bit size fields. The flags field specifies operating characteristics of your application, and the size fields indicate the minimum and preferred partition sizes for your application. The **minimum partition size** is the actual limit below which your application will not run. The **preferred partition size** is the memory size at which your application can run most effectively and that the Operating System attempts to secure upon launch of your application. If that amount of memory is unavailable, your application is placed into the largest contiguous block available, provided that it is larger than the specified minimum size.

Note

If the amount of available memory is between the minimum and the preferred sizes, the Finder displays a dialog box asking if the user wants to run the application using the amount of memory available. If your application does not have a `'SIZE'` resource, it is assigned a default partition size of 512 KB and the Process Manager uses a default value of FALSE for all specifications normally defined by constants in the flags field. u

When you define a 'SIZE' resource, you should give it a resource ID of –1. A user can modify the preferred size in the Finder's information window for your application. If the user does alter the partition size, the Operating System creates a new 'SIZE' resource having a resource ID of 0. At application launch time, the Process Manager looks for a 'SIZE' resource with ID 0; if this resource is not found, it uses your original 'SIZE' resource with ID –1. This new 'SIZE' resource is also created when the user modifies any of the other settings in the resource.

When creating a 'SIZE' resource, you first need to determine the various operating characteristics of your application. For example, if your application has nothing useful to do when it is in the background, then you should not set the canBackground flag. Similarly, if you have not tested your application in an environment that uses all 32 bits of a handle or pointer for memory addresses, then you should not set the is32BitCompatible flag.

Listing 2-4 shows the Rez input for a sample 'SIZE' resource. (Rez is a resource compiler available with the MPW environment.)

Listing 2-4 The Rez input for a sample 'SIZE' resource

```
resource 'SIZE' (-1) {
   reserved,                     /*reserved*/
   acceptSuspendResumeEvents,    /*accepts suspend&resume events*/
   reserved,                     /*reserved*/
   canBackground,                /*can use background null */
                                 /* events*/
   doesActivateOnFGSwitch,       /*activates own windows in */
                                 /* response to OS events*/
   backgroundAndForeground,      /*application has a user */
                                 /* interface*/
   dontGetFrontClicks,           /*don't return mouse events */
                                 /* in front window on resume*/
   ignoreAppDiedEvents,          /*doesn't want app-died events*/
   is32BitCompatible,            /*works with 24- or 32-bit addr*/
   isHighLevelEventAware,        /*supports high-level events*/
   localAndRemoteHLEvents,       /*also remote high-level events*/
   isStationeryAware,            /*can use stationery documents*/
   dontUseTextEditServices,      /*can't use inline input */
                                 /* services*/
   reserved,                     /*reserved*/
   reserved,                     /*reserved*/
   reserved,                     /*reserved*/
   kPrefSize * 1024,             /*preferred memory size*/
   kMinSize * 1024               /*minimum memory size*/
};
```

The `'SIZE'` resource specification in Listing 2-4 indicates, among other things, that the application accepts suspend and resume events, does processing in the background using null events, activates or deactivates any windows as necessary in response to operating-system events, can execute in both the foreground and background, and doesn't want to receive any mouse event associated with a resume event that was caused by the user clicking in the application's front window. It also indicates that the application doesn't want to receive Application Died events, can work in 24-bit or 32-bit addressing mode, does accept high-level events, including both local and network high-level events, does handle stationery documents, and doesn't use TextEdit's inline input services. In this example, the Rez-input file must define values for the constants `kPrefSize` and `kMinSize`; for example, if `kPrefSize` is set to 50, the preferred partition size is 50 KB.

The numbers you specify as your application's preferred and minimum memory sizes depend on the particular memory requirements of your application. Your application's memory requirements depend on the size of your application's static heap, dynamic heap, A5 world, and stack. (See "Introduction to Memory Management" in *Inside Macintosh: Memory* for complete details about these areas of your application's partition.)

The static heap size includes objects that are always present during the execution of your application—for example, code segments, Toolbox data structures for window records, and so on.

Dynamic heap requirements come from various objects created on a per-document basis (which may vary in size proportionally with the document itself) and objects that are required for specific commands or functions.

The size of the A5 world depends on the amount of global data and the number of intersegment jumps your application contains.

The stack contains variables, return addresses, and temporary information. The size of the application stack varies among computers, so you should base your values for the stack size according to the stack size required on a Macintosh Plus computer (8 KB). The Process Manager automatically adjusts your requested amount of memory to compensate for the different stack sizes on different machines. For example, if you request 512 KB, more stack space (approximately 16 KB) will be allocated on machines with larger default stack sizes.

Unfortunately, it is difficult to forecast all of these conditions with any great degree of reliability. You should be able to determine reasonably accurate estimates for the stack size, static heap size, A5 world, and jump table. In addition, you can use tools such as MacsBug's heap-exploring commands to help you empirically determine your application's dynamic memory requirements.

See "The Size Resource" beginning on page 2-115 for additional information on the meaning of each of the fields and flags of a `'SIZE'` resource.

Handling Low-Level Events

Low-level events include hardware-related occurrences stored in the Operating System event queue and activate and update events generated by the Window Manager. When your application receives a low-level event, your application needs to determine the type of event and respond appropriately. The following sections discuss how to respond to mouse events, keyboard events (including certain specific keyboard events, such as when the user presses the Command key and period key at the same time), update events, activate events, disk-inserted events, and null events.

Responding to Mouse Events

Whenever the user presses or releases the mouse button, the Operating System Event Manager records the action in the Operating System event queue. These actions are stored in the event queue as mouse-down and mouse-up events. Your application can retrieve these events using the `WaitNextEvent` function.

Events related to movements of the mouse are not stored in the event queue. The mouse driver automatically tracks the mouse and displays the cursor as the user moves the mouse. Therefore, the Operating System Event Manager does not report an event if the user simply moves the mouse.

However, you can request that the Event Manager report mouse-moved events if the user moves the cursor out of a region that you specify to the `WaitNextEvent` function. For example, your application can use mouse-moved events in this way to change the shape of the cursor from an I-beam to an arrow when the user moves the cursor from a text area to the scroll bar of a window.

The rest of this section describes how your application responds to mouse-down or mouse-up events. See "Responding to Mouse-Moved Events" beginning on page 2-62 for specific details on mouse-moved events.

The user expects that pressing the mouse button correlates to particular actions in an application. Your application is responsible for providing feedback or performing any actions in response to the user. For example, if the user presses the mouse button while the cursor is in the menu bar, your application should use the Menu Manager function `MenuSelect` to allow the user to choose a menu command.

Your application can receive and respond to mouse-down and mouse-up events. Most applications respond to mouse-down events and use the routines of various managers (such as `MenuSelect`, `DragWindow`, `TEClick`, `TrackBox`, `TrackGoAway`, and `TrackControl`) to handle the corresponding mouse-up events. You can also provide code to respond to mouse-up events if it's appropriate for your application. For example, if your application implements its own text-editing capabilities, you might let the user select lines of text by dragging the mouse and use mouse-up events to signal the end of the selection.

In System 7, your application receives mouse-down events only when it is the foreground process and the user clicks in the menu bar, in a window belonging to your application, or in a window belonging to a desk accessory that was launched in your application's partition. If the user clicks in a window belonging to another application, the Event Manager sends your application a suspend event and performs a major switch to the other application.

When your application receives a mouse-down event, you need to first determine the location of the cursor at the time the mouse button was pressed (the **mouse location**) and respond appropriately. You can use the Window Manager function `FindWindow` to find which of your application's windows, if any, the mouse button was pressed in and, if applicable, to find which part of the window it was pressed in. The `FindWindow` function also reports whether the given mouse location is in the menu bar or, in some cases, in a window belonging to a desk accessory (if the desk accessory was launched in your application's partition).

The `what` field of the event record for a mouse event contains the `mouseDown` or `mouseUp` constant to report that the mouse button was pressed or released. The `message` field is undefined. The `when` field contains the number of ticks since the system last started up. You can use the `when` field to compare how much time has elapsed between successive mouse events; for example, you might use this information to help detect mouse double clicks.

The `where` field of the event record contains the location of the cursor at the time the mouse button was pressed or released. You can pass this location to the `FindWindow` function; the `FindWindow` function maps the given mouse location to particular areas of the screen.

The `modifiers` field contains information about the state of the modifier keys at the time the mouse button was pressed or released. Your application can perform different actions based on the state of the modifier keys. For example, your application might let the user extend a selection or select multiple objects at a time if the Shift key was down at the time of the mouse-down event.

Listing 2-5 shows code that handles mouse-down events. The `DoMouseDown` procedure is an application-defined procedure that is called from the `DoEvent` procedure. (Listing 2-3 on page 2-26 shows the `DoEvent` procedure.)

Listing 2-5 Handling mouse-down events

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:             Integer;
   thisWindow:       WindowPtr;
BEGIN
   {map location of the cursor (at the time of mouse-down event) }
   { to general areas of the screen}
   part := FindWindow(event.where, thisWindow);

   CASE part OF {take action based on the mouse location}

   inMenuBar: {mouse down in menu bar, respond appropriately}
      BEGIN
         {first adjust marks and enabled state of menu items}
         MyAdjustMenus;
         {let user choose a menu command}
         DoMenuCommand(MenuSelect(event.where));
      END;

  inSysWindow: {cursor in a window belonging to a desk accessory}
      SystemClick(event, thisWindow);
  inContent: {mouse down occurred in the content area of }
             { one of your application's windows}
   IF thisWindow <> FrontWindow THEN
   BEGIN {mouse down occurred in a window other than the front }
         { window—-make the window clicked in the front window, }
         { unless the front window is movable modal}
      IF MyIsMovableModal(FrontWindow) THEN
         SysBeep(30)
      ELSE
         SelectWindow(thisWindow);
   END
   ELSE  {mouse down was in the content area of front window}
      DoContentClick(thisWindow, event);

  inDrag:               {handle mouse down in drag area}
      IF (thisWindow <> FrontWindow) AND
         (MyIsMovableModal(FrontWindow))
      THEN
         SysBeep(30)
      ELSE
        DragWindow(thisWindow, event.where, GetGrayRgn^^.rgnBBox);

   inGrow:              {handle mouse down in grow region}
      DoGrowWindow(thisWindow, event);
   inGoAway:            {handle mouse down in go-away region}
      IF TrackGoAway(thisWindow, event.where) THEN
         DoCloseCmd;
   inZoomIn, inZoomOut: {handle mouse down in zoom box region}
      IF TrackBox(thisWindow, event.where, part) THEN
         DoZoomWindow(thisWindow, part);
   END; {end of CASE}
END;{of DoMouseDown}
```

When your application retrieves a mouse-down event, call the Window Manager function `FindWindow` to map the location of the cursor to particular areas of the screen. Given a mouse location, the `FindWindow` function returns as its function result a value that indicates whether the mouse location is in the menu bar, in one of your application's windows, or, in some cases, in a desk accessory window. If the mouse location is in an application window, the function result indicates which part of the window the mouse location is in. You can test the function result of `FindWindow` against these constants to determine the mouse location at the time of the mouse-down event:

```
CONST inDesk      = 0;{none of the following}
      inMenuBar   = 1;{in the menu bar}
      inSysWindow = 2;{in a desk accessory window}
```

```
inContent   = 3;{anywhere in content region except the }
                { grow region if the window is active, }
                { anywhere in content region including the }
                { grow region if the window is inactive}
inDrag      = 4;{in drag (title bar) region}
inGrow      = 5;{in grow region (active window only)}
inGoAway    = 6;{in go-away region (active window only)}
inZoomIn    = 7;{in zoom-in region (active window only)}
inZoomOut   = 8;{in zoom-out region (active window only)}
```

The `FindWindow` function reports the `inDesk` constant if the mouse location is not in the menu bar, desk accessory window, or any window of your application. For example, the `FindWindow` function may report this constant if the location of the cursor is inside a window frame but not in the drag region or go-away region of the window; your application seldom receives the `inDesk` constant.

If `FindWindow` returns the `inMenuBar` constant, the mouse location is in the menu bar. In this case your application should first adjust its menus. The application-defined `MyAdjustMenus` procedure adjusts its menus—enabling and disabling items and setting marks—based on the context of the active window. For example, if the active window is a document window that contains a selection, your application should enable the Cut and Copy commands in the Edit menu, add marks to the appropriate items in the Font, Size, and Style menus, and adjust any other menu items accordingly. After adjusting your application’s menus, call the Menu Manager function `MenuSelect`, passing it the location of the mouse, to allow the user to choose a menu command. The `MenuSelect` function handles all user interaction until the user releases the mouse button. The `MenuSelect` function returns as its function result a long integer indicating the menu selection made by the user. As shown in Listing 2-5 on page 2-34, the `DoMouseDown` routine calls an application-defined routine, `DoMenuCommand`, to perform the menu command selected by the user. See the chapter “Menu Manager” in this book for a listing that gives the code for the `MyAdjustMenus` and `DoMenuCommand` routines and for more information about responding to specific menu commands.

In System 7, the `FindWindow` function seldom returns the `inSysWindow` constant. The `FindWindow` function returns this constant only when a mouse-down event occurred in a desk accessory that was launched in the application’s partition. Normally, if the user clicks in a desk accessory’s window, the Event Manager sends your application a suspend event and brings the desk accessory to the foreground. From that point on, mouse-down events and other events are handled by the desk accessory until the user again clicks in one of your application’s windows.

If `FindWindow` does return the `inSysWindow` constant, the mouse location is in a window belonging to a desk accessory that was launched in your application’s partition. In this case, your application should call the `SystemClick` procedure. The `SystemClick` procedure routes the event to the desk accessory as appropriate. If the mouse button was pressed while the cursor was in the content region of the desk accessory’s window and the window is inactive, `SystemClick` makes it the active window. It does this by sending your application an activate event to deactivate its front window and directing an event to the desk accessory to activate its window.

FindWindow can return any of the constants inContent, inDrag, inGrow, inGoAway, inZoomIn, or inZoomOut if the given mouse location is in your application's active window. If the cursor is in the content area, your application should perform any actions appropriate to your application. Note that scroll bars are part of the content region. In most cases, if the cursor is in the content area, your application first needs to determine whether the mouse location is in the scroll bar or any other controls and then respond appropriately. The DoMouseDown procedure calls the application-defined procedure DoContentClick to handle mouse-down events in the content area of the active window. If your application needs to determine whether the mouse-down event caused a foreground switch (and you set the getFrontClicks flag in your application's 'SIZE' resource), your DoContentClick procedure can test bit 0 in the modifiers field of the event record (normally your application does not test for this condition). See the chapter "Control Manager" in this book for an example DoContentClick procedure and for detailed information on implementing controls in your application's windows.

If the mouse location is in any of the other specified regions of an active application window, your application should perform the action corresponding to that region. For example, if the cursor is in the drag region, your application should call the Window Manager procedure DragWindow to allow the user to drag the window to a new location.

If the mouse location is in an inactive application window, FindWindow can return the inContent or inDrag constant, but does not distinguish between any other areas of the window. In this case, if FindWindow reports the inContent constant, your application should bring the inactive window to the front using the SelectWindow procedure (unless the active window is a movable modal dialog box). If the active window is a movable modal dialog box, then your application should use the SysBeep procedure to play the system alert sound rather than activating the selected window. Also, if your application interprets the first mouse click in an inactive window as a request to activate the window *and* perform an action, you can process the event again. However, note that most users expect the first click in an inactive window to activate the window without performing any additional action. If FindWindow reports inDrag for an inactive application window, your application should call the DragWindow procedure to allow the user to drag the window to a new location (unless the active window is a movable modal dialog box, in which case your application should simply play the system alert sound).

If you're using TextEdit to handle text editing and call TEClick, TEClick automatically interprets mouse double clicks appropriately, including allowing the user to select a word by double-clicking it. Your application must provide the means to allow double-clicking in this manner in all other contexts.

You can detect mouse double clicks by comparing the time and location of a mouse-up event with that of the immediately following mouse-down event. The GetDblTime function returns the recommended difference in ticks that should exist between the occurrence of a mouse-up and mouse-down event for those two mouse events to be considered a double click.

You should interpret mouse events as a double click if both of these conditions are true:

- The times of the mouse-up event and mouse-down event differ by a number of ticks less than or equal to the value returned by the `GetDblTime` function.
- The locations of the two mouse-down events separated by the mouse-up event are sufficiently close to each other. How you determine this value depends on your application and the context in which the mouse-down events occurred. For example, in a word-processing application, you might consider two mouse-down events a double click if the mouse locations both mapped to the same character, whereas in a graphics application you might consider it a double click if the sum of the horizontal and vertical difference between the two mouse locations is no more than five pixels.

The Event Manager also provides other routines that give information about the mouse. You can find the current mouse location using the `GetMouse` procedure. You can determine the current state of the mouse button using the `Button`, `StillDown`, and `WaitMouseUp` functions. See "Reading the Mouse" beginning on page 2-108 for detailed information on these routines.

Responding to Keyboard Events

Your application can receive keyboard events to notify you when the user has pressed or released a key or continued to hold down a key. When the user presses a key, the Operating System Event Manager stores a key-down event in the Operating System event queue. Your application can retrieve the event from the queue; determine which key was pressed; determine which modifier keys, if any, were pressed at the time of the event; and respond appropriately. Typically, your application provides feedback by echoing (displaying) the glyph representing the character generated by the pressed key on the screen.

When the user holds down a key for a certain amount of time, the Event Manager generates auto-key events. The Event Manager generates an auto-key event after a certain initial delay (the auto-key threshold) has elapsed since the original key-down event. The Event Manager generates subsequent auto-key events whenever a certain repeat interval (the auto-key rate) has elapsed since the last auto-key event and while the original key is still held down. The user can set the initial delay and rate of repetition using the Keyboard control panel. The default value for the auto-key threshold is 16 ticks, and the default value for the auto-key rate is 4 ticks. Current values of the auto-key threshold and auto-key rate are stored in the system global variables `KeyThresh` and `KeyRepThresh`.

In addition to getting keyboard events when the user presses or releases a key, you can directly read the keyboard (and keypad) using the `GetKeys` procedure.

When the user presses a key or a combination of keys, your application should respond appropriately. Your application should follow the guidelines in *Macintosh Human Interface Guidelines* for consistent use of and response to keyboard events. For example, your application should allow the user to choose a frequently used menu command by using a keyboard equivalent for that menu command—usually a combination of the Command key and another key. Your application should also respond to the user pressing the arrow keys, Shift key, or other keys according to the guidelines provided in *Macintosh Human Interface Guidelines.*

Also note that certain keyboards have different physical layouts or contain additional keys, such as function keys. If your application supports function keys or other special keys, you should follow the guidelines in *Macintosh Human Interface Guidelines* when determining what action to take when the user presses one of these keys.

Certain keystroke combinations are handled by the Event Manager and not returned to your application. If the user holds down the Command and Shift keys while pressing a numeric key to produce a special effect, that special effect occurs. Apple provides three standard Command–Shift–number key sequences. The standard Command–Shift–number key sequences are 1 for ejecting the disk in the internal drive, 2 for ejecting the disk in a second internal drive or for ejecting the disk in an external drive if the computer has only one internal drive, and 3 for taking a snapshot of the screen and storing it as a TeachText document on the startup volume.

The action corresponding to a Command–Shift–number key sequence is implemented as a routine that takes no parameters and is stored in an 'FKEY' resource with a resource ID that corresponds to the number that activates it. Apple reserves 'FKEY' resources with resource IDs 1 through 4 for its own use; if you provide an 'FKEY' resource, use a resource ID between 5 and 9.

You can disable the Event Manager's processing of Command–Shift–number key sequences for numbers 3 through 9 by setting the system global variable `ScrDmpEnb` (a byte) to 0. However, in most cases you should not disable the Event Manager's processing of these events.

The `what` field of the event record for a keyboard-related event contains either the `keyDown` or `keyUp` constant to indicate that the key was pressed or released, or the `autokey` constant to indicate that the key is being held down.

The Event Manager sets the system event mask of your application to accept all events except key-up events. Most applications ignore key-up events. If your application needs to receive key-up events, you can change the system event mask of your application using the Operating System Event Manager procedure `SetEventMask`.

In the low-order word the `message` field contains the character code and virtual key code that corresponds to the key pressed by the user.

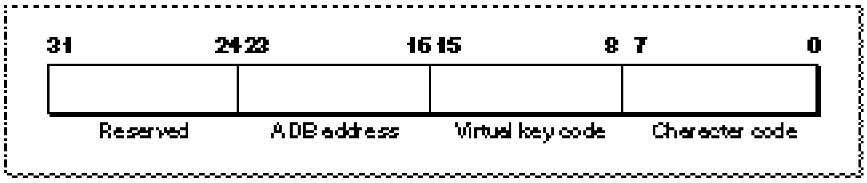
The **virtual key code** represents the key pressed or released by the user; this value is always the same for a specific physical key on a particular keyboard. For example, on the Apple Keyboard II, ISO layout, the virtual key code for the fifth key to the right of the Tab key (the key labeled "T") is always $11, regardless of which modifier keys are also pressed.

To determine the virtual key code that corresponds to a specific physical key, system software uses a hardware-specific key-map (`'KMAP'`) resource that specifies the virtual key codes for a particular keyboard. After determining the virtual key code of the key pressed by the user, system software uses a script-specific keyboard-layout (`'KCHR'`) resource to map a virtual key code to a specific character code. Any given script system has one or more `'KCHR'` resources. For example, a particular computer might contain the French `'KCHR'` resource in addition to the standard U.S. `'KCHR'` resource. In this situation, the current `'KCHR'` resource determines whether virtual key codes are mapped to the French or U.S. character set.

The **character code** represents a particular character. The character code that is generated depends on the virtual key code, the state of the modifier keys, and the current 'KCHR' resource. For example, the U.S. 'KCHR' resource specifies that for the virtual key code $2D (the fifth key to the left of the Shift key and labeled "N" on an Apple Keyboard II, Domestic layout), the character code is $6E when no modifier keys are pressed; the character code is $4E when this key is pressed in combination with the Shift key. Character codes for the Roman script system are specified in the extended version of ASCII (the American Standard Code for Information Interchange).

The `message` field contains additional information for ADB keyboards. The low-order byte of the high-order word contains the ADB address of the keyboard where the keyboard event occurred. Figure 2-7 shows the structure of the `message` field of the event record for keyboard events.

Figure 2-7 The `message` field of the event record for keyboard events



Usually your application uses the character code, rather than the virtual key code, when responding to keyboard events. You can use these two constants to access the virtual key code and character code in the `message` field:

```
CONST charCodeMask      = $000000FF;{mask for character code}
      keyCodeMask       = $0000FF00;{mask for virtual key code}
```

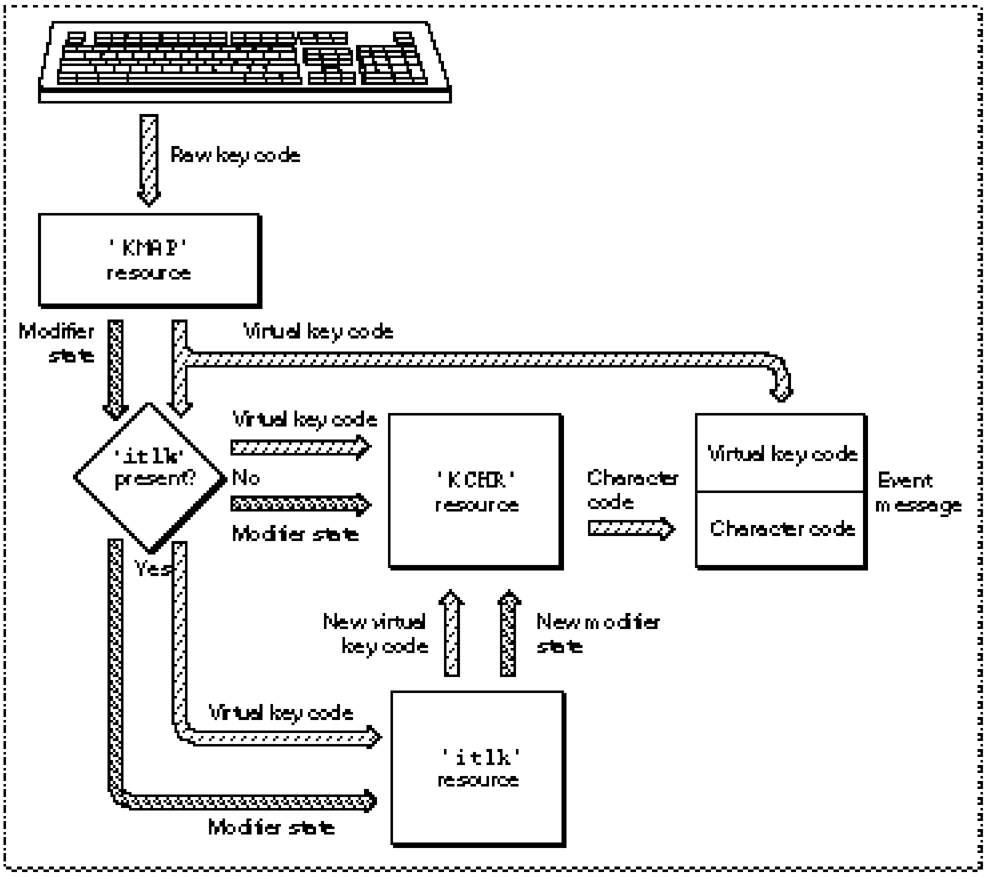
The `when` field contains the number of ticks since the system last started up. You can use the `when` field to compare how much time has expired between successive keyboard events.

The `where` field of the event record contains the location of the cursor at the time the key was pressed or released. You typically disregard the mouse location when processing keyboard events.

The `modifiers` field contains information about the state of the modifier keys at the time the key was pressed or released. Your application can perform different actions based on the state of the modifier keys. For example, your application might perform an action associated with a corresponding menu command if the Command key was down at the time of the key-down event.

System software can support a number of different types of keyboards, for example, the Apple Keyboard II, the Apple Extended keyboards, or other keyboards. The system software uses various keyboard resources and international resources to manage different types of keyboards. Figure 2-8 illustrates how system software maps keys to character codes.

Figure 2-8 Keyboard translation



When a user presses or releases a key on the keyboard, the keyboard generates a raw key code. The system software uses a 'KMAP' resource to map the raw key code to a hardware-independent virtual key code and to set bits indicating the state of the modifier keys. A 'KMAP' resource specifies the physical arrangement of a particular keyboard and indicates the virtual key codes that correspond to each physical key.

If the optional key-remap ('itlk') resource is present, the system software remaps the virtual key codes and modifier state for some key combinations on certain keyboards before using the 'KCHR' resource. The 'itlk' resource can reintroduce hardware dependence because certain scripts, languages, and regions need subtle differences in layout for specific keyboards. If present, the 'itlk' resource affects only a few keys.

After mapping the virtual key code and the state of the modifier keys through an optional 'itlk' resource, the system software uses a 'KCHR' resource to produce the character code representing the key that was pressed or released. The 'KCHR' resource specifies how to map the setting of the modifier keys and a virtual key code to a character code.

After mapping the key, the Event Manager returns the virtual key code and the character code in the `message` field of the event record.

Figure 2-9 shows the virtual key codes as specified by the `'KMAP'` resource for the Apple Keyboard II, ISO layout. The labels for the keys on the keyboard are shown using the U.S. keyboard layout. The virtual key codes are shown in hexadecimal.

Figure 2-9 Virtual key codes for the Apple Keyboard II, ISO layout

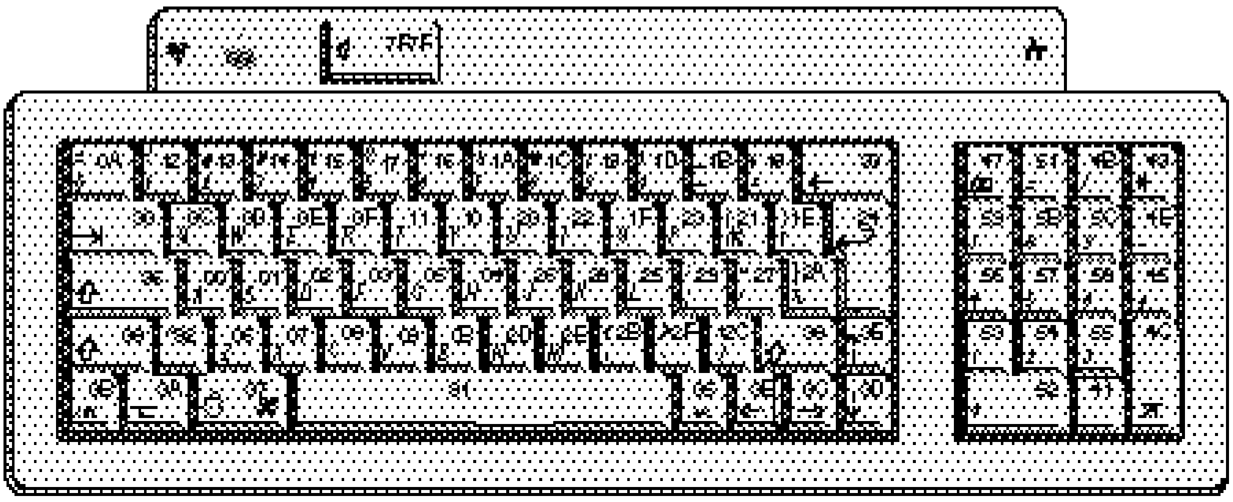


Figure 2-10 shows the virtual key codes as specified by the `'KMAP'` resource for the Apple Extended Keyboard II, one that uses the Domestic (ANSI) layout, and one that uses the ISO layout. The labels for the keys on the ISO keyboard are shown using the French keyboard layout. The virtual key codes are shown in hexadecimal.

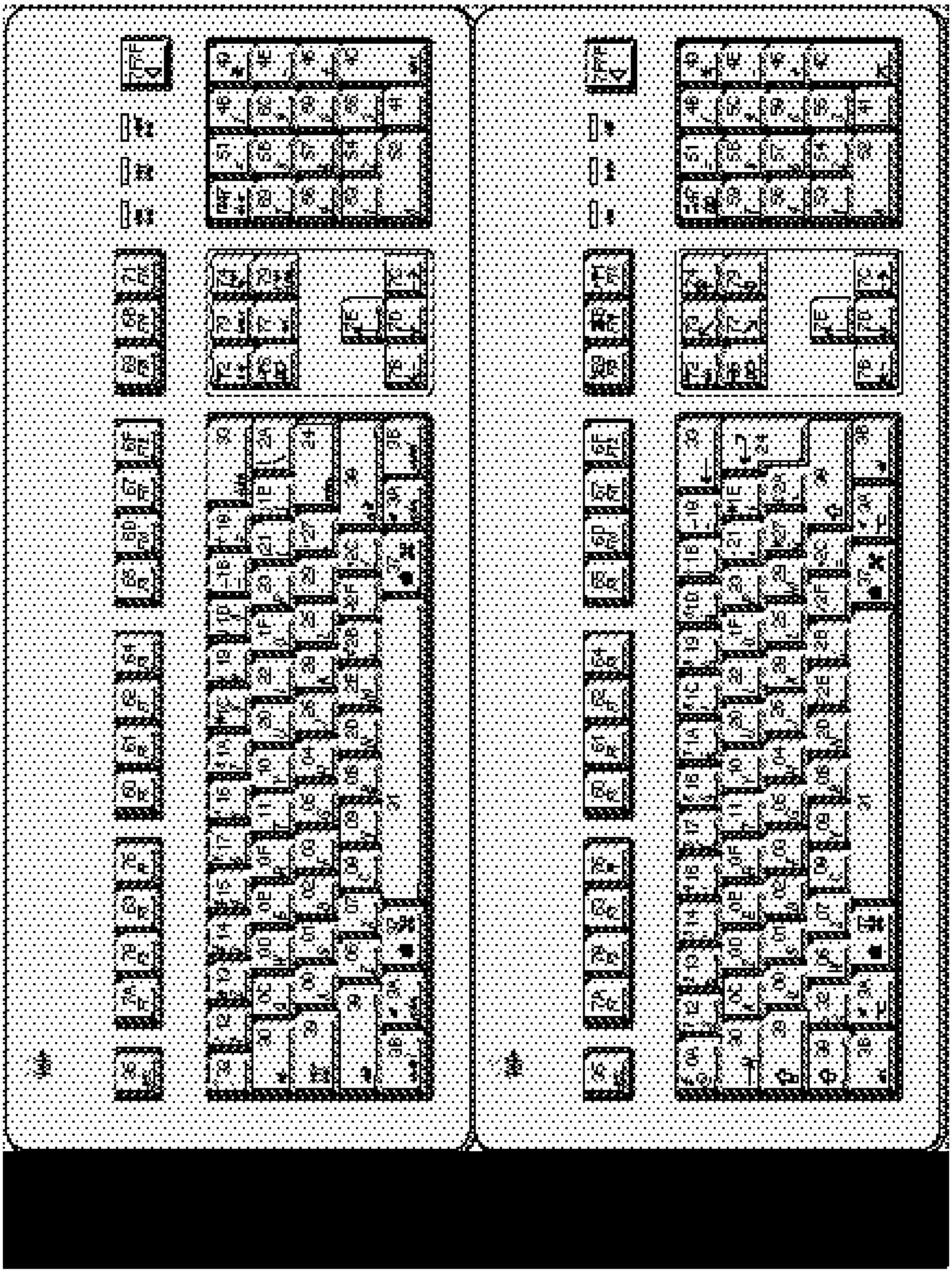
If a user of an Apple Extended Keyboard II (using the U.S. `'KCHR'` resource) presses the key labeled "C" and no modifier keys, the system software maps this through the `'KMAP'` and `'KCHR'` resources to produce a virtual key code of $08 and the character code $63 (the character "c") in the `message` field of the event record. If the user presses the key labeled "C" and the Option key, then the system software maps this to virtual key code $08 and the character code $8D (the character "ç") in the `message` field.

As another example, if a user of an Apple Extended Keyboard II, Domestic layout, is using the U.S. `'KCHR'` resource and presses the key labeled "M" the system software maps this through the `'KMAP'` and `'KCHR'` resources to produce a virtual key code of $2E and the character code $6D (the character "m") in the `message` field of the event record.

If a user of an Apple Extended Keyboard II, ISO layout, is using the French `'KCHR'` resource and presses the key labeled "M" the system software maps this through the `'KMAP'` and `'KCHR'` resources to produce a virtual key code of $29 and the character code $6D (the character "m") in the `message` field of the event record.

See *Inside Macintosh: Text* for additional information about the keyboard resources and how the Script Manager manages various scripts.

Figure 2-10 Virtual key codes for the Apple Extended Keyboard II



Listing 2-6 shows code that handles key-down and auto-key events. The `DoKeyDown` procedure is an application-defined procedure that is called from the `DoEvent` procedure. (Listing 2-3 on page 2-26 shows the `DoEvent` procedure.)

Listing 2-6 Handling key-down and auto-key events

```
PROCEDURE DoKeyDown (event: EventRecord);
VAR
   key:  Char;
BEGIN
   key := CHR(BAnd(event.message, charCodeMask));
   IF BAnd(event.modifiers, cmdKey) <> 0 THEN
   BEGIN                      {Command key down}
      IF event.what = keyDown THEN
      BEGIN {first enable/disable/check menu items as needed-- }
            { the MyAdjustMenus procedure adjusts the menus }
            { as appropriate for the current window}
         MyAdjustMenus;
         DoMenuCommand(MenuKey(key)); {handle the menu command}
      END;
   END
   ELSE
      MyHandleKeyDown(event);
END;
```

The `DoKeyDown` procedure in Listing 2-6 first extracts the character code of the key pressed from the `message` field of the event record. It then checks the `modifiers` field of the event record to determine if the Command key was pressed at the time of the event. If so, and if the event is a key-down event, the code calls the application-defined procedure `MyAdjustMenus`, and then calls another application-defined routine, `DoMenuCommand`, to perform the menu command associated with that key. (The `MyAdjustMenus` procedure adjusts the menus appropriately, and according to whether the current window is a document window or modeless dialog box. See the chapter “Menu Manager” in this book for code that defines the `MyAdjustMenus` procedure.) Otherwise, the code calls the application-defined procedure `MyHandleKeyDown` to handle the event.

Listing 2-7 shows the application-defined routine `MyHandleKeyDown`.

Listing 2-7 Handling key-down events

```
PROCEDURE MyHandleKeyDown (event: EventRecord);
VAR
   key:           Char;
   window:        WindowPtr;
   myData:        MyDocRecHnd;
   te:            TEHandle;
   windowType:    Integer;
BEGIN
   window := FrontWindow;
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN
      key := CHR(BAnd(event.message, charCodeMask));
      IF window <> NIL THEN
      BEGIN
         IF key = char(kTab) THEN {handle special characters}
            MyDoTab(event)
         ELSE
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window));
            te := myData^^.editRec;
            IF
            (te^^.teLength - (te^^.selEnd - te^^.selStart) + 1
               < kMaxTELength) THEN
            BEGIN
               TEKey(key, te);   {insert character in document}
               MyAdjustScrollBars(window, FALSE);
               MyAdjustTE(window);
               myData^^.windowDirty := TRUE;
            END;
         END;
      END;
   END
   ELSE
      MyHandleKeyDownInModeless(event, windowType);
END;
```

The `MyHandleKeyDown` procedure in Listing 2-7 handles key-down events in any window of the application. For document windows, the code inserts the character represented by the key pressed by the user into the active document. It first finds the active document using the `FrontWindow` function, then handles the event as appropriate for the document window. For example, it treats the Tab key as a special character and calls an application-defined routine, `MyDoTab`, to handle this character appropriately for the document. For all other keys directed to the document window, the code gets the edit record associated with the document, and then it simply inserts the character into the document, using the TextEdit `TEKey` procedure. It also calls two other application-defined routines, `MyAdjustScrollBars` and `MyAdjustTE`, to update the document and edit record.

The `MyHandleKeyDown` procedure calls an application-defined routine, `MyHandleKeyDownInModeless`, to handle key-down events in modeless dialog boxes. See the chapter "Dialog Manager" in this book for more information on handling events in dialog boxes.

Scanning for a Cancel Event

Your application should allow the user to cancel a lengthy operation by using the Command-period combination. Your application can implement this cancel operation by periodically examining the state of the keyboard using the `GetKeys` procedure, or your application can scan the event queue for a keyboard event.

Listing 2-8 shows an application-defined function that scans the event queue for any occurrence of a Command-period event.

The `UserDidCancel` function in Listing 2-8 first checks to see if the user changed the script. The application maintains a global variable, `gCurrentKeyScript`, that keeps track of this information. The application also uses a global variable, `gPeriodKeyCode`, to hold the key code that maps to the period key according to the current script. If the current script has changed, the `UserDidCancel` function calls an application-defined routine, `MySetPeriodKeyCode`, to change the value of the `gPeriodKeyCode` global variable as necessary.

The `UserDidCancel` function then determines whether A/UX is running. You must use a different method to scan the event queue if A/UX is running. This code uses an application-defined function called `MyCheckAUXEventQueue` to search for a Command-period event if A/UX is running. Otherwise, the code checks the `what` field for a key-down event. If it finds a key-down event, it then checks the `message` field to determine whether the user pressed the period key and checks the `modifiers` field to determine whether the user also pressed the Command key. If it finds the Command-period combination, it sets the `foundEvent` variable to `TRUE` and returns this value. Otherwise, it looks at the next entry in the queue and continues to search the queue until it either finds a Command-period event or reaches the end of the queue.

Listing 2-8 Scanning for a Command-period event

```
FUNCTION UserDidCancel: Boolean;
VAR
   foundEvent:    Boolean;
   eventQPtr:     EvQElPtr;
   eventQHdr:     QHdrPtr;
   keyCode:       LongInt;
   isCmdKey:      LongInt;
BEGIN
   foundEvent := FALSE;             {assume the event is not there}
   {Check to see if the script has changed}
   IF (gCurrentKeyScript <> GetEnvirons(smKeyScript)) THEN
      MySetPeriodKeyCode;           {set gPeriodKeyCode to match new script}
   IF (GetAUXVersion > 0) THEN      {if A/UX is running use this method}
      foundEvent := MyCheckAUXEventQueue(gPeriodKeyCode, cmdKey)
   ELSE
   BEGIN                                        {scan event queue}
      eventQHdr := GetEvQHdr;                   {get the event queue header}
      eventQPtr := EvQElPtr(eventQHdr^.qHead);  {get first entry}
      WHILE (eventQPtr <> NIL) AND (NOT(foundEvent)) DO
      BEGIN                                     {look for key-down event}
         IF (eventQPtr^.evtQWhat = keyDown) THEN {found key-down event, }
         BEGIN                                  { look for Command-period}
            keyCode := BAND(eventQPtr^.evtQMessage, keyCodeMask);
            keyCode := BSR(keyCode, 8);
            isCmdKey := BAND(eventQPtr^.evtQModifiers, cmdKey);
            IF isCmdKey <> 0 THEN               {Command key was pressed}
               IF keyCode = gPeriodKeyCode THEN
                  foundEvent := TRUE;           {key pressed was '.'}
         END;     {of found key-down}
         IF (NOT foundEvent) THEN               {go to next entry}
            eventQPtr := EvQElPtr(eventQPtr^.qLink);
      END;     {of while}
   END;     {of scan event queue}
   UserDidCancel := foundEvent;                 {return result of search}
END;
```

Responding to Update Events

The Event Manager reports update events to your application whenever one of your application's windows needs updating. Upon receiving an update event, your application should update the contents of the specified window. Your application can call the Window Manager procedure `BeginUpdate`, draw the window's contents, and then call `EndUpdate` when your application has finished updating the window's contents.

Your application can also let the Window Manager automatically update the contents of a window by supplying in the window record a handle to a picture that contains the contents of the window. This technique is generally useful only for windows that contain static information that doesn't change or can't be edited. For example, if your application provides a window that always displays a picture of the earth, you can supply the handle to the picture, and the Window Manager automatically updates the window as needed, without sending your application an update event. In most cases, your application needs to perform the update itself.

The Window Manager maintains an update region for each window. The Window Manager keeps track of all areas in a window's content region that need to be redrawn and accumulates them in the window's update region. When an application calls `WaitNextEvent` or `EventAvail` (or `GetNextEvent`), the Event Manager checks to see if any windows have an update region that is not empty. If so, the Event Manager reports update events to the appropriate applications; any applications with windows that require updating receive the necessary update events according to the normal processing of events.

If more than one window needs updating, the Event Manager issues update events for the frontmost window first. This means that updating of windows occurs in front-to-back order, which is what the user expects.

When one of your application's windows needs to be updated, the Window Manager calls the window definition function of that window, requesting that it draw the window frame. The Window Manager then generates an update event for that window. The Event Manager reports any update events for your application's windows to your application, and your application should update the window contents as necessary.

In response to an update event, your application should first call the `BeginUpdate` procedure. The `BeginUpdate` procedure temporarily replaces the visible region of the window's graphics port (that part of the window that is visible on the screen) with the intersection of the visible region and update region of the window. The `BeginUpdate` procedure then clears the update region of the window—preventing the update event for this occurrence from being reported again.

After calling `BeginUpdate`, your application should draw the window's contents, either entirely or in part. You can draw either the entire content region or only the area in the visible region. In either case, the Window Manager allows only what falls within the visible region to be drawn on the screen. (Because the `BeginUpdate` procedure intersects the visible region with the update region, the visible region at this point corresponds to any visible parts of the old update region.)

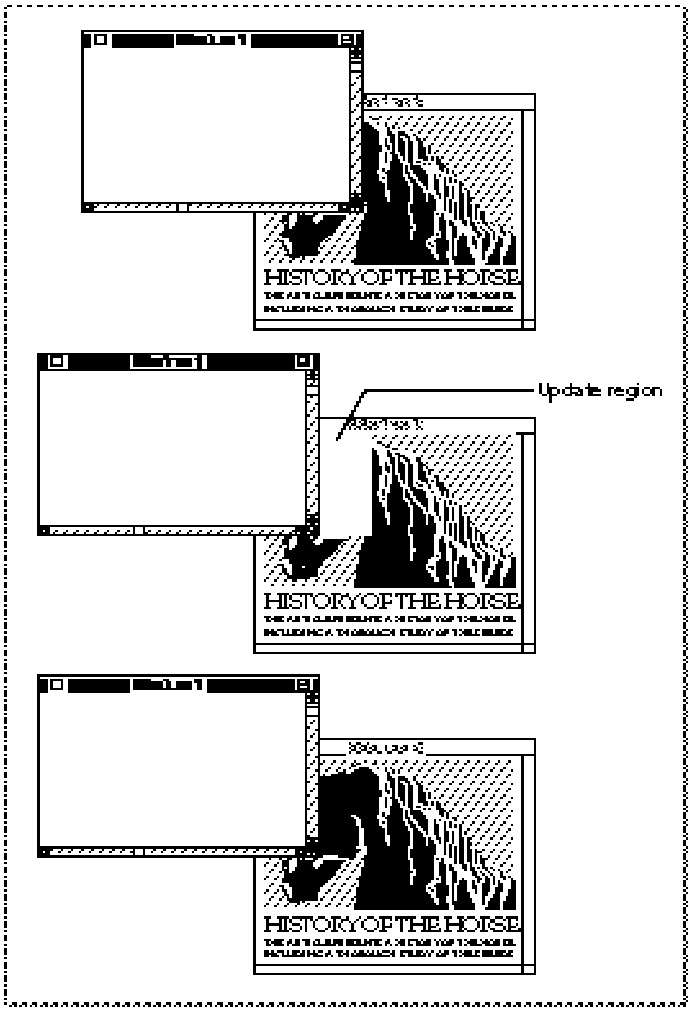
The `EndUpdate` procedure restores the normal visible region of the window's graphics port.

Figure 2-11 shows how an application updates its windows. In this example, Window 1 partially covers Window 2. When the user moves Window 1 so that more of Window 2 is exposed, the Window Manager requests the window definition function of the window to update the window frame, and accumulates the area requiring updating in the update region of the window.

When the application receives an update event for this window, the `message` field of the event record contains a pointer to the window that needs updating. Your application can call `BeginUpdate`, draw the window's contents, and then call `EndUpdate`. This completes the handling of the update event.

Your application can receive update events when it is in the foreground or in the background. In the example shown in Figure 2-11, Window 1 and Window 2 could belong to the same application or different applications. In either case, the Event Manager reports an update event to the application whose window contents need updating.

Figure 2-11 Responding to an update event for a window



Your application should respond to update events or at least call the `BeginUpdate` procedure in response to an update event. If you do not call the `BeginUpdate` procedure, your application continues to receive update events for the window (until the update region is empty). You should always make sure that you match a call to `BeginUpdate` with a call to `EndUpdate`. By calling the `BeginUpdate` and `EndUpdate` procedures, you indicate to the Window Manager that you have updated the window and handled the update event.

Listing 2-9 shows an example of an application-defined routine that responds to update events.

Listing 2-9 Responding to update events

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: Integer;
BEGIN
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         BEGIN
            BeginUpdate(window);
            MyDrawWindow(window);
            EndUpdate(window);
         END;
      OTHERWISE
         DoUpdateMyDialog(window);
   END; {of CASE}
END;
```

The `DoUpdate` procedure in Listing 2-9 first determines if the window is a document window or a modeless dialog box. The `MyGetWindowType` function is an application-defined routine that returns the `kMyDocWindow` constant if the window is a document window and returns other application-defined constants if the window is a modeless dialog box.

If the window is a document window, the procedure does all its drawing of the window within calls to the `BeginUpdate` and `EndUpdate` procedures. The application-defined routine `MyDrawWindow` performs the actual updating of the document window contents. See the chapter "Window Manager" in this book for code that shows the `MyGetWindowType` and `MyDrawWindow` routines.

If the window is a modeless dialog box, the code calls the application-defined `DoUpdateMyDialog` procedure to update the contents of the dialog box. See the chapter "Dialog Manager" in this book for details on handling update events in dialog boxes.

Responding to Activate Events

When several windows belonging to your application are open, you should allow the user to switch from one window to another by clicking in the appropriate window. To implement this, whenever your application receives a mouse-down event, you should first determine whether the user clicked in another window by using the Window Manager function `FindWindow`; if so, you can use the Window Manager procedure `SelectWindow` to generate the necessary activate events.

Before returning to your application and before your application receives any events relating to this occurrence, the `SelectWindow` procedure does some work for you, such as removing the highlighting from the window to be deactivated and highlighting the newly activated window. At your application's next request for an event, the Event Manager returns an activate event.

An activate event indicates the window involved and whether the window is being activated or deactivated. Your application should perform any other actions needed to complete the action of the window becoming active or inactive. For example, when a window becomes active, your application should show any scroll bars and restore selections as necessary.

Your application typically receives an activate event (with a flag that indicates the window should be deactivated) for the window being deactivated, followed by an activate event for the window becoming active.

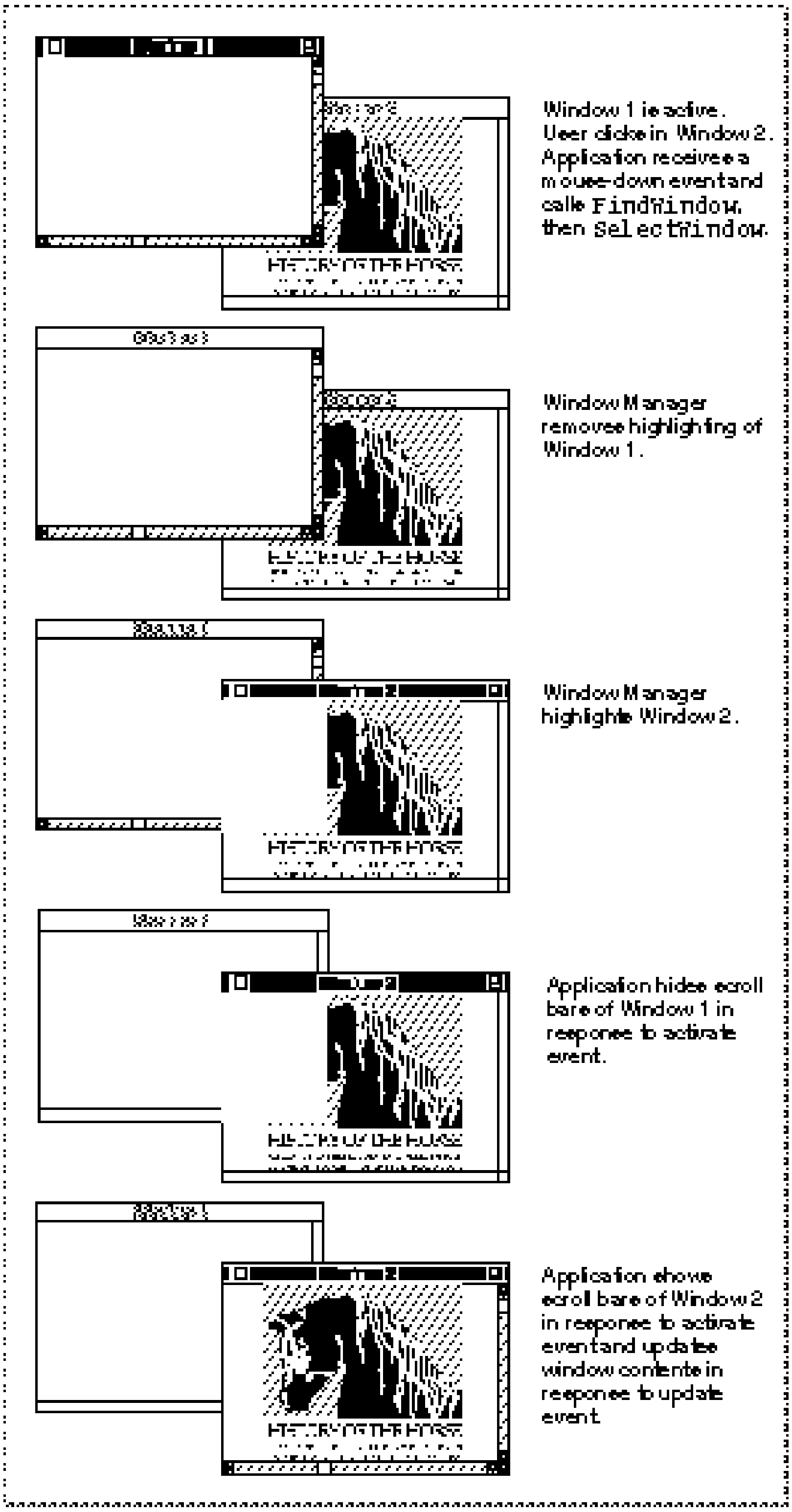
Activate events are not placed into the Operating System event queue but are sent directly to the Event Manager.

Figure 2-12 on the next page shows two documents belonging to the same application, with Window 1 the active window. When the user clicks in Window 2, your application receives a mouse-down event and can use the `FindWindow` function to determine whether the mouse location is in an inactive window. If so, your application should call the `SelectWindow` procedure. The `SelectWindow` procedure removes highlighting of Window 1, highlights Window 2, and generates activate events for both of these occurrences. The Event Manager reports the activate events one at a time to your application; in this example, the first activate event indicates that Window 1 should be deactivated. Your application should hide the scroll bars and remove the highlighting from any selections as necessary.

The next activate event indicates that Window 2 should be activated. Your application should show the scroll bars and restore any selections as necessary. If the window needs updating as a result of being activated, the Event Manager sends your application an update event so that your application can update the window contents.

Your application also needs to activate or deactivate windows in response to suspend and resume events. If you set the `acceptSuspendResumeEvents` flag and the `doesActivateOnFGSwitch` flag in your application's `'SIZE'` resource, your application is responsible for activating or deactivating your application's windows in response to handling suspend and resume events. If you set the `acceptSuspendResumeEvents` flag and do not set the `doesActivateOnFGSwitch` flag, your application receives an activate event immediately following a suspend or resume event. In most cases, you should set both the `acceptSuspendResumeEvents` and `doesActivateOnFGSwitch` flags in your application's `'SIZE'` resource.

Figure 2-12 Responding to activate events for a window


Window 1 is active. User clicks in Window 2. Application receives a mouse-down event and calls FindWindow, then SelectWindow.
Window Manager removes highlighting of Window 1.
Window Manager highlights Window 2.
Application hides scroll bars of Window 1 in response to activate event.
Application shows scroll bars of Window 2 in response to activate event and updates window contents in response to update event.

The what field of an event record for an activate event contains the activateEvt constant. The message field contains a pointer to the window being activated or deactivated. The modifiers field contains additional information about the activate event, along with information about the state of the modifier keys at the time the event was posted. Your application can examine bit 0 of the modifiers field of the event record to determine if the window should be activated or deactivated. Bit 0 of the modifiers field is 1 if the window should be activated and 0 if the window should be deactivated. You can use the activeFlag constant to test the state of this bit in the modifiers field.

The when field of the event record contains the number of ticks since the system last started up. The where field of the event record contains the location of the cursor at the time the activate event occurred.

Upon receiving an activate event that indicates the window is being deactivated, your application should hide any scroll bars and remove the highlighting from any selections as necessary.

Upon receiving an activate event that indicates the window is becoming active, your application should show any scroll bars, highlight any selections, and otherwise restore the window to the state it was in when it was last active. For example, your application should restore the insertion point to its previous position, and the document should be scrolled to the position in which the user last left it. Your application should also adjust its menus appropriately for the newly active window—adjusting the marks and enabled state of menu items based on the state of the active window.

Listing 2-10 shows an application-defined procedure that responds to activate events.

Listing 2-10 Responding to activate events

```
PROCEDURE DoActivate (window: windowPtr; activate: Boolean;
                      event: EventRecord);
VAR
   growRect:   Rect;          {window's grow rectangle}
   myData:     MyDocRecHnd;   {window's document record}
   windowType: Integer;
BEGIN
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:
   BEGIN
      myData := MyDocRecHnd(GetWRefCon(window));
      HLock(Handle(myData));
      WITH myData^^ DO
      IF activate THEN     {window is being activated}
      BEGIN
         {restore any selections or display caret}
         MyRestoreSelection(window);
         {adjust menus as appropriate for this document window}
         MyAdjustMenus;
         {activate any scroll bars}
         vScrollBar^^.contrlVis := kControlVisible;
         hScrollBar^^.contrlVis := kControlVisible;
         {invalidate area of scroll bars to force update}
         InvalRect(vScrollBar^^.contrlRect);
         InvalRect(hScrollBar^^.contrlRect);
         {invalidate area of size box, if any}
         growRect := window^.portRect;
         WITH growRect DO
         BEGIN
            top := bottom - kScrollbarAdjust;
            left := right - kScrollbarAdjust;
         END; {end of WITH growRect statement}
         InvalRect(growRect);
      END
      ELSE                 {window is being deactivated}
      BEGIN
         {unhighlight selection (if any) or hide the caret}
         MyHideSelection;
         HideControl(vScrollBar);   {hide any scroll bars}
         HideControl(hScrollBar);
         DrawGrowIcon(window);      {change size box immediately}
      END;
      HUnLock(Handle(myData));
   END; {end of kMyDocWindow}

   kMyGlobalChangesID:  {this window is a modeless dialog box }
                        { for this app's Global Changes command}
      MyDoActivateGlobalChangesDialog(window, event);
   {handle other modeless dialog boxes as appropriate}
   END; {of CASE}
END;
```

Listing 2-10 uses the application-defined function `MyGetWindowType` to determine what type of window is involved with the activate event. If the window is a document window, the `DoActivate` procedure uses the `GetWRefCon` function to get a handle to the window's document record. (The `DoActivate` procedure, and other application-defined routines, maintain information about the document associated with a window in a document record; the application stores a handle to the document record as the window's reference constant value when it creates a new window. See the chapter "Window Manager" in this book for information on defining a document record.)

If the document window should be activated, the code calls an application-defined routine, `MyRestoreSelection`. Your application should restore any selection or display the caret as appropriate. For example, if your application uses TextEdit to display text in the content area of windows, you can call the TextEdit procedure `TEActivate` to restore any selection or display a caret at the insertion point. The `DoActivate` procedure then calls another application-defined procedure, `MyAdjustMenus`, to adjust the menus as appropriate for the document window. (See the chapter "Menu Manager" for a listing of the `MyAdjustMenus` procedure.) After restoring any selections and adjusting its menus, the code shows the scroll bars and size box of the window being activated. It does this by invalidating the area of the scroll bars and size box, accumulating these areas into the update region. This causes an update event to be generated. The application redraws its controls as appropriate in response to update events.

If the document window should be deactivated, the code in Listing 2-10 unhighlights the selection and hides the caret by calling the application-defined procedure `MyHideSelection`. The code then hides the scroll bars and size box of the deactivated window.

If the window associated with the activate event is a modeless dialog box, for example, a Global Changes modeless dialog box, the `DoActivate` procedure calls an application-defined procedure to activate or deactivate the dialog box as needed. See the "Dialog Manager" chapter in this book for information on handling activate events in modeless dialog boxes.

Responding to Disk-Inserted Events

When your application uses the Standard File Package to allow the user to choose a file to open or choose a location for storing a file, the Standard File Package responds to disk-inserted events for your application while interacting with the user. In most cases, if your application receives an unexpected disk-inserted event, it can simply check to see if the disk was successfully mounted and use the Disk Initialization Manager function `DIBadMount` to notify the user if the disk was not successfully mounted.

When the user inserts a disk, the Operating System attempts to mount the volume on the disk by calling the File Manager function `PBMountVol`. If the volume is successfully mounted, an icon representing the disk appears on the desktop. The Operating System Event Manager then generates a disk-inserted event. If the user is interacting with a standard file dialog box, the Standard File Package intercepts the disk-inserted event and handles it. Otherwise, the event is left in the event queue for your application to retrieve. The Desk Manager also intercepts and handles disk-inserted events if a desk accessory is in front.

Usually your application should handle and not mask out disk-inserted events. The user might insert a disk at any time and expects to be warned if the disk is uninitialized or damaged. If your application receives a disk-inserted event and the volume was successfully mounted, your application usually does not need to take any further action. However, if the volume was not successfully mounted, then your application should give the user a chance to initialize or eject the uninitialized or damaged disk.

If you do mask out disk-inserted events, the event stays in the Operating System event queue until your application calls the Standard File Package or until an application that does handle disk-inserted events becomes the foreground process. This situation can be confusing to the user, so your application should handle disk-inserted events at the time that they occur.

If the volume was successfully mounted and your application either does not use the Standard File Package or prompts the user to insert a disk, then you can choose to respond to disk-inserted events in whatever way is appropriate for your application.

The Dialog Manager procedure `ModalDialog` masks out disk-inserted events. (The Standard File Package changes the mask in order to receive disk-inserted events.) If one of your application's modal dialog boxes needs to respond to disk-inserted events, then you can change the event mask from within the event filter function that you supply as one of the parameters to `ModalDialog`. Otherwise, your application can respond to the disk-inserted event after the user dismisses the modal dialog box.

The `what` field of the event record contains the `diskEvt` constant to indicate a disk-inserted event. The `message` field contains the drive number in the low-order word and the result code from the `PBMountVol` function in the high-order word. Your application can examine the high-order word to determine if the attempt to mount the volume was successful. If the volume was not successfully mounted, your application can notify the user using the Disk Initialization Manager function `DIBadMount`. If the volume was successfully mounted, your application can use the drive number returned in the low-order word for accessing the disk.

Listing 2-11 shows a procedure that handles disk-inserted events. If the disk was not successfully mounted, the procedure notifies the user using the `DIBadMount` function. Otherwise, it does not take any action. See the chapter "Disk Initialization Manager" in *Inside Macintosh: Files* for information on the routines provided by the Disk Initialization Manager.

Listing 2-11 Responding to disk-inserted events

```
PROCEDURE DoDiskEvent (event: EventRecord);
VAR
   thisPoint:  Point;
   myErr:      OSErr;
BEGIN
   IF HiWord(event.message) <> noErr THEN
   BEGIN                      {attempt to mount was unsuccessful}
      DILoad;                 {load Disk Initialization Manager}
      SetPt(thisPoint, 120, 120);
                              {notify the user}
      myErr := DIBadMount(thisPoint, event.message);
      DIUnload;               {unload Disk Initialization Manager}
   END
   ELSE                        {attempt to mount was successful}
      ;        {record the drive number or do other processing}
END;
```

Responding to Null Events

When the Event Manager has no other events to report, it returns a null event. The WaitNextEvent function reports a null event by returning a function result of FALSE and setting the what field of the returned event record to nullEvt. (The EventAvail and GetNextEvent functions also return null events in this manner.)

When your application receives a null event, it can perform idle processing. Your application should do minimum processing in response to a null event, so that other processes can use the CPU and so that the foreground process (or your application, if it is in the foreground) can respond promptly to the user.

For example, if your application receives a null event and it is in the foreground, it can make the caret blink in the active window.

If your application receives a null event in the background, it can perform tasks or do other processing while in the background. However, your application should not perform any tasks that would slow down the responsiveness of the foreground process. Your application also should not interact with the user if it is in the background.

If you don't want your application to receive null events when it is in the background, set the cannotBackground flag in your application's 'SIZE' resource.

Listing 2-12 shows a procedure that performs idle processing in response to a null event. If the application is not in the background and the active window is a document window, this code calls the TextEdit procedure TEIdle. The TEIdle procedure makes a blinking caret appear at the insertion point in the text referred to by the edit record. (This application uses TextEdit to display text in its document windows; if you don't use TextEdit for your document windows, provide your own routine to blink the caret.) If the active window is a modeless dialog box, the DoIdle procedure calls the Dialog Manager function DialogSelect to blink the caret in any editable text item of the dialog box.

Listing 2-12 Handling null events

```
PROCEDURE DoIdle (event: EventRecord);
VAR
   window:     WindowPtr;
   myData:     MyDocRecHnd;
   windowType: Integer;
   itemHit:    Integer;
   result:     Boolean;
```

```
BEGIN
   window := FrontWindow;
   {determine the type of window--document, modeless, etc.}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         IF (NOT gInBackground) THEN
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window));
            TEIdle(myData^^.editRec);
         END;
      kMyGlobalChangesID:
         result := DialogSelect(event, window, itemHit);
   END; {of CASE}
END;
```

Handling Operating-System Events

Operating-system events include suspend, resume, and mouse-moved events. Your application receives suspend and resume events as a result of changes in its processing status. Your application can request that the Event Manager return mouse-moved events whenever the cursor is outside a specified region by specifying a nonempty region in the `mouseRgn` parameter to `WaitNextEvent`. If you specify an empty region or a `NIL` region handle in the `mouseRgn` parameter, the Event Manager does not report mouse-moved events.

Your application examines the event record to determine which event it received and to obtain additional information associated with the event.

The `what` field in the event record of an operating-system event contains the `osEvt` constant.

The `message` field in the event record of an operating-system event contains information indicating whether the event is a suspend, resume, or mouse-moved event. The `message` field also indicates whether Clipboard conversion is required when the application resumes execution. The bits in the `message` field give this information:

Bit	Contents
0	0 if a suspend event
	1 if a resume event
1	0 if Clipboard conversion not required
	1 if Clipboard conversion required
2–23	Reserved
24–31	`suspendResumeMessage` if a suspend or resume event `mouseMovedMessage` if a mouse-moved event

Note that you need to examine bits 24–31 of the `message` field to determine what kind of operating-system event you have received. Bits 24–31 in the `message` field contain one of these two constants:

```
CONST suspendResumeMessage = $01;      {suspend or resume event}
      mouseMovedMessage    = $FA;      {mouse-moved event}
```

If the event is a suspend or resume event, you need to examine bit 0 to determine whether that event is a suspend or resume event. Bits 0 and 1 are meaningful only if bits 24–31 indicate that the event is a suspend or resume event. You can use the `resumeFlag` constant to determine whether the event is a suspend or resume event. If the event is a resume event, you can use the `convertClipboardFlag` constant to determine whether Clipboard conversion from the Clipboard to your application's scrap is required:

```
CONST resumeFlag           = 1;  {resume event}
      convertClipboardFlag = 2;  {Clipboard conversion required}
```

Whenever the user performs a copy or cut operation, your application should copy the selected data either to its private scrap or, if your application doesn't have a private scrap, to the Clipboard. If your application uses a private scrap, you need to convert the data from your private scrap to the Clipboard whenever your application receives a suspend event. Likewise, you need to convert any data from the Clipboard (if it has changed) when your application receives a resume event. For resume events, the value of bit 1 of the `message` field is 1 if your application needs to read in the new contents of the Clipboard.

Listing 2-13 shows a procedure that responds to operating-system events.

Listing 2-13 Responding to operating-system events

```
PROCEDURE DoOSEvent (event: EventRecord);
BEGIN
   CASE BAnd(BRotL(event.message, 8), $FF) OF {get high byte}
      mouseMovedMessage:
         DoIdle(event); {mouse-moved same as idle for this app}
      suspendResumeMessage:
         DoSuspendResumeEvent(event);{handle supend/resume event}
   END;
END;
```

The `DoOSEvent` procedure in Listing 2-13 is called from the `DoEvent` procedure (shown in Listing 2-3 on page 2-26) whenever the application receives an operating-system event. The `DoOSEvent` procedure examines the high byte of the `message` field to determine whether the event is a mouse-moved, suspend, or resume event, and it then calls an application-defined procedure to handle the event. Note that most applications either adjust the cursor in response to mouse-moved events or adjust the cursor in their event loop whenever any type of event is received. The code in this chapter uses the latter approach, and thus the `DoOSEvent` procedure simply calls its `DoIdle` procedure in response to mouse-moved events. The next two sections show the code that handles suspend, resume, and mouse-moved events.

Responding to Suspend and Resume Events

The `WaitNextEvent` function returns a suspend event when your application is about to be switched to the background. `WaitNextEvent` returns a resume event when your application becomes the foreground process again.

Upon receiving a suspend event, your application should deactivate the front window, remove the highlighting from any selections, and hide any floating windows. Your application should also convert any private scrap into the global scrap, if necessary. If your application shows a window that displays the Clipboard contents, you should hide this window also, as the user might change the contents of the Clipboard before returning to your application. Your application can also do anything else necessary to get ready for a major switch. Then your application should call `WaitNextEvent` to relinquish the processor and allow the Operating System to schedule other processes for execution.

Upon receiving a resume event, your application should activate the front window and restore any windows to the state the user left them in at the time of the previous suspend event. For example, your application should show scroll bars, restore any selections that were previously in effect, and show any floating windows. Your application should copy the contents of the Clipboard and convert the data back to its private scrap, if necessary. If your application shows a window that displays the Clipboard contents, you can update the contents of the window after reading in the scrap. Your application can then resume interacting with the user.

Responding to a suspend or resume event usually involves activating or deactivating windows. If you set the `acceptSuspendResumeEvents` flag and the `doesActivateOnFGSwitch` flag in your application's `'SIZE'` resource, your application is responsible for activating or deactivating your application's windows in response to handling suspend and resume events.

Note

If you set the `acceptSuspendResumeEvents` flag and do not set the `doesActivateOnFGSwitch` flag in your application's `'SIZE'` resource, your application receives an activate event immediately following a suspend or resume event. In most cases, you should set both the `acceptSuspendResumeEvents` and `doesActivateOnFGSwitch` flags in your application's `'SIZE'` resource. u

Your application can use the Scrap Manager functions `InfoScrap`, `ZeroScrap`, `PutScrap`, and `GetScrap` to read data from and write data to the Clipboard. See the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox* for additional details.

Note
If your application does not handle suspend and resume events (as indicated by a flag in its 'SIZE' resource), then the Operating System has to trick your application into performing scrap coercion to ensure that the contents of the Clipboard can be transferred from one application to another. This process adds to the time it takes to move the foreground application to the background and vice versa. u

Listing 2-14 shows a procedure that responds to suspend and resume events. The DoSuspendResumeEvent procedure first gets a pointer to the front window using the Window Manager function FrontWindow. It then examines bit 0 of the message field of the event record to determine whether the event is a suspend or resume event. If the event is a resume event, the code examines bit 1 of the message field of the event record to determine whether it needs to read in the contents of the scrap. If so, the code calls an application-defined routine, MyConvertScrap, that reads in the scrap and converts the contents to its private scrap. It then sets a private global flag, gInBackground, to FALSE, to indicate that the application is not in the background. It then calls another application-defined routine, DoActivate (shown in Listing 2-10), to activate the application's front window.

For suspend events, the DoSuspendResumeEvent procedure calls the application-defined MyConvertScrap procedure to copy the contents of its private scrap to the global scrap. It then sets a private global flag, gInBackground, to TRUE, to indicate that the application is in the background. Finally, it calls another application-defined routine to deactivate the application's front window.

Listing 2-14 Responding to suspend and resume events

```
PROCEDURE DoSuspendResumeEvent (event: EventRecord);
VAR
   currentFrontWindow: WindowPtr;
BEGIN                         {handle suspend/resume event}
   currentFrontWindow := FrontWindow;
   IF (BAnd(event.message, resumeFlag) <> 0) THEN
   BEGIN                {it's a resume event}
      IF (BAnd(event.message, convertClipboardFlag) <> 0) THEN
         MyConvertScrap(kClipboardToPrivate);
      gInBackground := FALSE;
                        {activate front window}
      DoActivate(currentFrontWindow, NOT gInBackground, event);
      MyShowClipboardWindow;  {show Clipboard window if it was }
                              { showing at last suspend event}
      MyShowFloatingWindows;  {show any floating windows}
   END
   ELSE
   BEGIN                {it's a suspend event}
      MyConvertScrap(kPrivateToClipboard);
      gInBackground := TRUE;
                        {deactivate front window}
      DoActivate(currentFrontWindow, NOT gInBackground, event);
      MyHideClipboardWindow; {hide Clipboard window if showing}
      MyHideFloatingWindows; {hide any floating windows}
   END;
END;
```

Your application can receive processing time while in the background and perform tasks in the background, but your application should not interact with the user or perform tasks that would slow down the responsiveness of the foreground process.

If you need to notify the user of some special occurrence while your application is executing in the background, you should use the Notification Manager to queue a notification request. See the chapter “Notification Manager” in *Inside Macintosh: Processes* for examples of how to post notification requests.

Responding to Mouse-Moved Events

Whenever the user moves the mouse, the mouse driver, the Event Manager, and your application are responsible for providing feedback to the user. The mouse driver performs low-level functions, such as continually polling the mouse for its location and status and maintaining the current location of the mouse in a global variable.

As the user moves the mouse, the user expects the cursor to move to a corresponding relative location on the screen. The low-level interrupt routines of the mouse driver map the movement of the mouse to relative locations on the screen. Whenever the user moves the mouse, a low-level interrupt routine of the mouse driver moves the cursor displayed on the screen and aligns the hot spot of the cursor with the new mouse location. A **hot spot** is a point that the mouse driver uses to align the cursor with the mouse location.

Your application is responsible for setting the initial appearance of the cursor, for restoring the cursor after `WaitNextEvent` returns, and for changing the appearance of the cursor as appropriate for your application. For example, most applications set the cursor to the I-beam when the cursor is inside a text-editing area of a document, and change the cursor to an arrow when the cursor is inside the scroll bar of a document. Your application can achieve this effect by requesting that the Event Manager report mouse-moved events if the user moves the cursor out of a region you specify in the `mouseRgn` parameter to the `WaitNextEvent` function.

The mouse driver and your application control the shape and appearance of the cursor. A cursor can be any 256-bit image, defined by a 16-by-16 bit square. The mouse driver displays the current cursor, which your application can change by using various cursor-handling routines (for example, the `SetCursor` procedure).

Figure 2-13 shows the standard arrow cursor. You can initialize the cursor to the standard arrow cursor using the `InitCursor` procedure. In Figure 2-13, the hot spot for the arrow cursor is at location (1,1). See *Inside Macintosh: Imaging* for information on the cursor-handling routines and for specific details of how your application can define its own cursors.

Figure 2-13 The standard arrow cursor

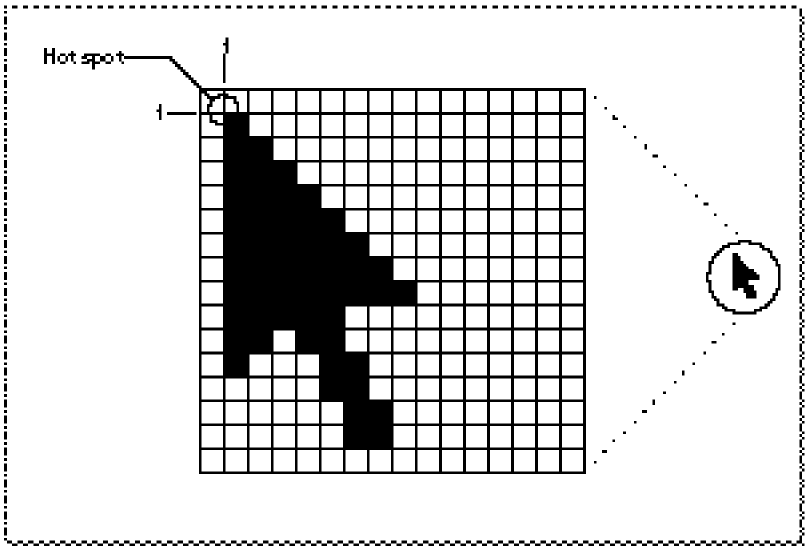
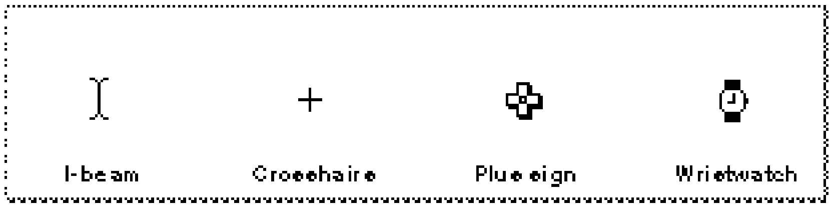


Figure 2-14 shows four other common cursors that are available to your application: the I-beam, crosshairs, plus sign, and wristwatch cursors.

Figure 2-14 The I-beam, crosshairs, plus sign, and wristwatch cursors



The I-beam, crosshairs, plus sign, and wristwatch cursors are defined as resources, and your application can get a handle to any of these cursors by specifying their corresponding resource IDs to the `GetCursor` function. These constants specify the resource IDs for the I-beam, crosshairs, plus sign, and wristwatch cursors:

```
CONST iBeamCursor = 1;{used in text editing}
      crossCursor = 2;{often used for manipulating graphics}
      plusCursor  = 3;{often used for selecting fields in }
                      { an array}
      watchCursor = 4;{used to mean a lengthy operation }
                      { is in progress}
```

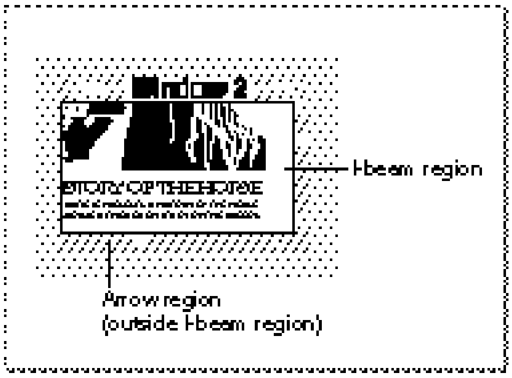
You can change the appearance of the cursor using the `SetCursor` procedure or other cursor-handling routines. You can also define your own cursors, store them in resources, and use them as needed in your application.

Your application usually needs to change the shape of the cursor as the user moves the cursor to different areas within a document. Your application can use mouse-moved events to accomplish this. Your application also needs to adjust the cursor in response to resume events. Most applications adjust the cursor once through the event loop in response to almost all events.

You can request that the Event Manager report mouse-moved events whenever the cursor is outside of a specified region that you pass as a parameter to the `WaitNextEvent` function. If you specify an empty region or a `NIL` handle to the `WaitNextEvent` function, `WaitNextEvent` does not report mouse-moved events.

If you specify a nonempty region in the `mouseRgn` parameter to the `WaitNextEvent` function, `WaitNextEvent` returns a mouse-moved event whenever the cursor is out of this region. For example, Figure 2-15 shows a document window. An application might define two regions: a region that encloses the text area of a window (the *I-beam region*), and a region that defines the scroll bars and all other areas outside the text area (the *arrow region*). By specifying the I-beam region to `WaitNextEvent`, the mouse driver continues to display the I-beam cursor until the user moves the cursor out of this region.

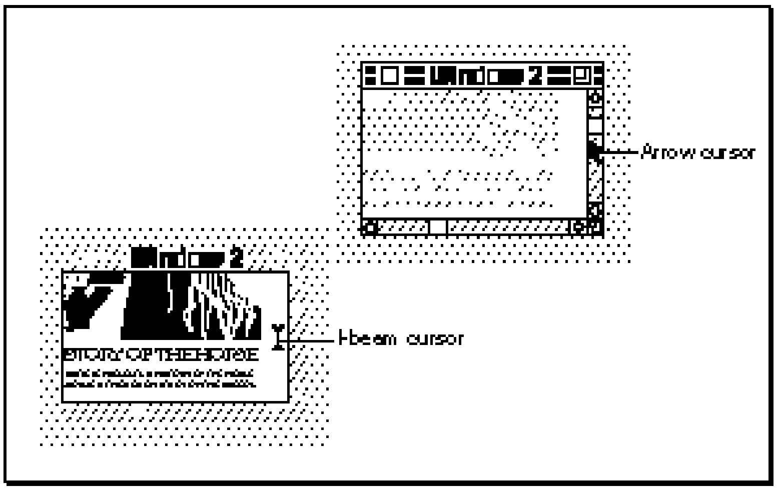
Figure 2-15 The arrow region and the I-beam region



When the user moves the cursor out of the I-beam region, `WaitNextEvent` reports a mouse-moved event. Your application can then change the I-beam cursor to the arrow cursor and change the `mouseRgn` parameter to the area defined by the scroll bars and all other areas outside of the I-beam region. The cursor now remains an arrow until the user moves the cursor out of this region, at which point your application receives a mouse-moved event.

Figure 2-16 shows how an application might change the cursor from the I-beam cursor to the arrow cursor after receiving a mouse-moved event.

Figure 2-16 Changing the cursor from the I-beam cursor to the arrow cursor



Note that your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event; otherwise, it will continue to receive mouse-moved events as long as the cursor position is outside the original region.

After receiving any event other than a high-level event, the `MyEventLoop` procedure (shown in Listing 2-2 on page 2-24) calls the application-defined procedure `MyAdjustCursor` to adjust the cursor. After adjusting the cursor, if the event is an operating-system event, the `DoEvent` procedure calls the `DoOSEvent` procedure. The `DoOSEvent` procedure calls the `DoIdle` procedure for mouse-moved events. The `DoIdle` procedure simply calls `TEIdle` to blink the caret in the text-editing window.

Listing 2-15 shows the application-defined routine `MyAdjustCursor`.

Listing 2-15 Changing the cursor

```
PROCEDURE MyAdjustCursor (mouse: Point; VAR region: RgnHandle);
VAR
   window:        WindowPtr;
   arrowRgn:      RgnHandle;
   iBeamRgn:      RgnHandle;
   iBeamRect:     Rect;
   myData:        MyDocRecHnd;
   windowType:    Integer;
BEGIN
window := FrontWindow;
{determine the type of window--document, modeless, etc.}
windowType := MyGetWindowType(window);
CASE windowType OF
   kMyDocWindow:
```

```
BEGIN
   {initialize regions for arrow and I-beam}
   arrowRgn := NewRgn;
   ibeamRgn := NewRgn;

   {set arrow region to large region at first}
   SetRectRgn(arrowRgn, -32768, -32768, 32766, 32766);

   {calculate I-beam region}
   {first get the document's TextEdit view rectangle}
   myData := MyDocRecHnd(GetWRefCon(window));
   iBeamRect := myData^^.editRec^^.viewRect;
   SetPort(window);
   WITH iBeamRect DO
   BEGIN
      LocalToGlobal(topLeft);
      LocalToGlobal(botRight);
   END;
   RectRgn(iBeamRgn, iBeamRect);
   WITH window^.portBits.bounds DO
      SetOrigin(-left, -top);
   {intersect I-beam region with window's visible region}
   SectRgn(iBeamRgn, window^.visRgn, iBeamRgn);
   SetOrigin(0,0);

   {calculate arrow region by subtracting I-beam region}
   DiffRgn(arrowRgn, iBeamRgn, arrowRgn);

   {change the cursor and region parameter as necessary}
   IF PtInRgn(mouse, iBeamRgn) THEN {cursor is in I-beam rgn}
   BEGIN
      SetCursor(GetCursor(iBeamCursor)^^);      {set to I-beam}
      CopyRgn(iBeamRgn, region);     {update the region param}
   END;

   {update cursor if in arrow region}
   IF PtInRgn(mouse, arrowRgn) THEN {cursor is in arrow rgn}
   BEGIN
      SetCursor(arrow);             {set cursor to the arrow}
      CopyRgn(arrowRgn, region);    {update the region param}
   END;
   DisposeRgn(iBeamRgn);
   DisposeRgn(arrowRgn);
END; {of kMyDocWindow}
```

```
   kMyGlobalChangesID:
      MyCalcCursorRgnForModelessDialogBox(window, region);

   kNil:
   BEGIN
      MySetRegionNoWindows(kNil, region);
      SetCursor(arrow);
   END;
 END; {of CASE}
END;
```

The `MyAdjustCursor` procedure sets the cursor appropriately, according to whether a document window or modeless dialog box is active.

For a document window, the code in Listing 2-15 defines two regions, specified by the `arrowRgn` and `iBeamRgn` variables. If the cursor is inside the region described by the `arrowRgn` variable, the code sets the cursor to the arrow cursor and returns the region described by `arrowRgn`. Similarly, if the cursor is inside the region described by the `iBeamRgn` variable, the code sets the cursor to the I-beam cursor and returns the region described by `iBeamRgn`.

The `MyAdjustCursor` procedure calculates the two regions by first setting the arrow region to the largest possible region. It then sets the I-beam region to the region described by the document's TextEdit view rectangle. This region typically corresponds to the content area of the window minus the scroll bars. (If your application doesn't use TextEdit for its document window, then set this region as appropriate to your application.) The code then adjusts the I-beam region so that it includes only the part of the content area that is in the window's visible region (for example, to take into account any floating windows that might be over the window). The code then sets the arrow region to include the entire screen except for the region occupied by the I-beam region.

The procedure then determines which region the cursor is in and sets the cursor and region parameter appropriately.

For modeless dialog boxes (for example, the Global Changes modeless dialog box), the `MyAdjustCursor` procedure calls an application-defined routine to appropriately adjust the cursor for the modeless dialog box. The `MyAdjustCursor` procedure also appropriately adjusts the cursor if no windows are currently open.

Handling High-Level Events

High-level events provide a means of communication between applications. Apple events are high-level events that follow the Apple Event Interprocess Messaging Protocol (AEIMP). In most cases, you should use Apple events rather than define your own high-level events if you wish to communicate with other applications. If you plan to use Apple events, see *Inside Macintosh: Interapplication Communication* for specific information on Apple events, and refer to this section for specific details about how the Event Manager reports high-level events.

To receive high-level events, you must set the appropriate flags in your application's 'SIZE' resource. You must set the isHighLevelEventAware flag if your application is to receive any high-level events. You must set the localAndRemoteHLEvents flag for your application to receive high-level events sent from another computer on the network. In addition, to receive high-level events from another computer, your application must be shared and Program Linking must be enabled. The user shares your application by selecting your application in the Finder and choosing Sharing from the File menu and enables Program Linking from the Sharing Setup control panel.

If you set the isHighLevelEventAware flag in your application's 'SIZE' resource, your application receives the Finder information in the form of Apple events. The Finder information is the information your application can use to determine which files to open or print. Your application must respond to the required Apple events (Open Application, Open Documents, Print Documents, and Quit Application) that are sent by the Finder if your application sends or receives high-level events.

The what field in the event record of a high-level event contains the kHighLevelEvent constant.

To determine the type of high-level event received, your application needs to examine the message and where fields of the event record. For high-level events, these two fields of the event record have special meanings.

The message field and the where field of the event record together define the specific type of high-level event received. Your application should interpret these fields as having the data type OSType, not LongInt or Point.

The message field contains the event class of the high-level event. For example, Apple events sent by the Edition Manager have the event class 'sect'. You can define your own group of events that are specific to your application. If you have registered your application signature with Apple Computer, Inc., then you can use your signature to define the class of events that belong to your application. Note, however, that Apple reserves the use of all event classes whose names contain only lowercase letters and nonalphabetic characters.

For high-level events, the where field in the event record contains a second message specifier, called the *event ID*. The event ID defines the particular type of event (or message) within the class of events defined by the event class. For example, the Section Read event sent by the Edition Manager has event class 'sect' and event ID 'read'. The Open Documents event sent by the Finder has event class 'aevt' and event ID 'odoc'. You can define your own set of event IDs corresponding to your own event class. For example, if the message field contains 'biff' and the where field contains 'cmd1', then the high-level event indicates the type of event defined by 'cmd1' within the class of events defined by the application with the signature 'biff'.

Note

If your application supports Apple events, you can call the AEProcessAppleEvent function to determine the type of Apple event received, rather than examining the message and where fields. u

Note that because the where field of an event record for a high-level event is used to select a specific kind of event (within the class determined by the message field), high-level event records do not contain the mouse location at the time of the event. You should not interpret the where field before interpreting the what field because different event classes can contain overlapping sets of event IDs.

Unlike low-level events and operating-system events, high-level events may not be completely determined by the event record returned to your application when it calls WaitNextEvent. For example, you might still need to know which other application sent you the high-level event or what additional data that application wants to send you. Your application can obtain this further information about the high-level event by calling the AcceptHighLevelEvent function. The additional information associated with a high-level event includes

- the identity of the sender of the event
- a unique number that identifies the request associated with the event or associates the particular event with a request from a previous event
- the address and length of a data buffer that can contain optional data

To obtain this additional information, your application must call AcceptHighLevelEvent before calling WaitNextEvent again. By convention, calling AcceptHighLevelEvent indicates that your application intends to process the high-level event.

To accept an Apple event, call the AEProcessAppleEvent function instead of the AcceptHighLevelEvent function. The Apple Event Manager also extracts any additional information associated with the Apple event at your application's request. This chapter discusses how to accept high-level events using the AcceptHighLevelEvent function; for information on the AEProcessAppleEvent function, see *Inside Macintosh: Interapplication Communication*.

Responding to Events From Other Applications

You can identify high-level events by the value in the what field of the event record. The message and where fields further classify the type of high-level event. Your application can choose to recognize as many events as are appropriate. Some high-level events may be fully specified by their event record only, while others may include additional information in an optional buffer. To get that additional information or to find the sender of the event, use the AcceptHighLevelEvent function.

Note

To respond to an Apple event, use the Apple Event Manager, as described in *Inside Macintosh: Interapplication Communication*. u

Listing 2-16 on the next page illustrates how to respond to a high-level event.

The DoHighLevelEvent procedure in Listing 2-16 first determines the type of high-level event received by checking the message and where fields of the event record. It then uses AcceptHighLevelEvent to get any additional data associated with the event. This particular application recognizes only one type of high-level event. If the event is not of this type, the code assumes that the event is an Apple event and calls AEProcessAppleEvent to handle the event.

In general, you cannot know in advance how big the optional data buffer is, so you can allocate a zero-length buffer and then resize it if the call to AcceptHighLevelEvent returns the bufferIsSmall result code.

Listing 2-16 Accepting a high-level event

```
PROCEDURE DoHighLevelEvent (event: EventRecord);
VAR
   myTarg:     TargetID;      {target ID record}
   myRefCon:   LongInt;
   myBuff:     Ptr;
   myLen:      LongInt;
   myErr:      OSErr;
BEGIN
   IF (event.message = LongInt(kMySpecialHLEventClass)) AND
      (LongInt(event.where) = LongInt(kMySpecialHLEventID)) THEN
   BEGIN
      {it's a high-level event that doesn't use AEIMP}
      myLen := 0;               {start with a 0-byte buffer}
      myBuff := NIL;
      myErr:=AcceptHighLevelEvent(myTarg,myRefCon, myBuff, myLen);
      IF myErr = bufferIsSmall THEN
      BEGIN
         myBuff := NewPtr(myLen);{allocate needed storage}
         myErr := AcceptHighLevelEvent(myTarg, myRefCon, myBuff,
                                       myLen);
         IF myErr = noErr THEN
            ;  {perform any action requested by the event}
      END;
      IF myErr <> noErr THEN
         DoError(myErr);{perform the necessary error handling}
   END
   ELSE
   BEGIN {otherwise, assume that the event is an Apple event}
      myErr := AEProcessAppleEvent(event);
      IF myErr <> noErr THEN
         DoError(myErr);{perform the necessary error handling}
   END;
END;
```

The AcceptHighLevelEvent function returns additional information and data associated with the event. The ID of the sender of the event is returned in the first parameter, which is a target ID record. You can inspect the fields of that record to determine which application sent the event. The target ID record contains the session reference number that identifies the connection with the other application as well as the port name and location name of the sender. If the high-level event requires that you return information, you can use the information returned in the target ID record to send an event back to the requesting application. See “Determining the Sender of a High-Level Event” on page 2-72 and “Sending High-Level Events” on page 2-73 for specific information on the target ID record.

The second parameter to `AcceptHighLevelEvent`, the reference constant parameter, is a unique number that identifies the request associated with the event or identifies that the particular event is related to a request from a previous event. If you send a response to this event, you should use the same value for the reference constant so that the sender of the event can associate the reply with the original request.

The third parameter points to any additional data associated with the event. Any data in this additional buffer is defined by the particular high-level event. On input, the fourth parameter to `AcceptHighLevelEvent`, the length parameter, contains the size of the buffer. If no error occurs, on output the length parameter contains the size of the message accepted. If the `AcceptHighLevelEvent` function returns the result code `bufferIsSmall`, the length parameter contains the size of the message yet to be received.

Searching for a Specific High-Level Event

Sometimes you do not want to accept the next available high-level event pending for your application. Instead, you might want to select one event from among all the high-level events in your application’s high-level event queue. For example, you might want to look for a return receipt for a high-level event you previously posted before processing other high-level events.

You can select a specific high-level event by calling the `GetSpecificHighLevelEvent` function. One of the parameters you pass to this function is a filter function that you provide. Your filter function should examine an event in your application’s high-level event queue and determine whether it is the kind of event you wish to receive. If it is, your filter function returns `TRUE`. This indicates that your filter function does not want to inspect any more events. If the filter function finds an event of the desired type, it should call `AcceptHighLevelEvent` to retrieve the event. When your function returns `TRUE`, the `GetSpecificHighLevelEvent` function itself returns `TRUE`.

If your filter function returns `FALSE` for an event in the high-level event queue, then `GetSpecificHighLevelEvent` looks at the next event in the high-level event queue and executes your filter function. If the filter function returns `FALSE` for all the high-level events in the queue, then `GetSpecificHighLevelEvent` itself returns `FALSE` to your application.

Here’s how you declare the filter function whose address you pass to the `GetSpecificHighLevelEvent` function:

```
FUNCTION MyFilter (yourDataPtr: Ptr;
                   msgBuff: HighLevelEventMsgPtr;
                   sender: TargetID): Boolean;
```

When your application calls `GetSpecificHighLevelEvent`, you pass it a parameter that indicates the criteria your filter function should use to search for a specific event. The `GetSpecificHighLevelEvent` function passes this information to your filter function in the `yourDataPtr` parameter. The `GetSpecificHighLevelEvent` function also provides your filter function with information about the event record of the high-level event in the `msgBuff` parameter as well as information about the sender of the high-level event in the `sender` parameter.

The `msgBuff` parameter contains a pointer to a high-level event message record that has this structure:

```
TYPE  HighLevelEventMsg =
      RECORD
         HighLevelEventMsgHeaderLength:      Integer;
         version:                            Integer;
         reserved1:                          LongInt;
         theMsgEvent:                        EventRecord;
         userRefCon:                         LongInt;
         postingOptions:                     LongInt;
         msgLength:                          LongInt;
      END;

      HighLevelEventMsgPtr= ^HighLevelEventMsg;
```

When you call `GetSpecificHighLevelEvent` and it executes your filter function for a high-level event waiting in the high-level event queue, the fields of the high-level event message record are filled in by the Event Manager. You can then compare the fields of this record to the information in the `yourDataPtr` parameter to determine whether that event suits your needs. For example, the `yourDataPtr` parameter might contain the signature of a return receipt. You can test its value against the event class of the event record contained in the `theMsgEvent` field of the high-level event message record.

Determining the Sender of a High-Level Event

When you receive a high-level event, part of the information returned by `AcceptHighLevelEvent` is the identity of the sender of the event. You can use that information to respond selectively to requests made by other applications or to find which application to send any replies to. The information about the sender is provided in the form of a target ID record, defined as follows:

```
TYPE  TargetID =
      RECORD
         sessionID:  LongInt;          {session reference number}
         name:       PPCPortRec;       {sender's port name}
         location:   LocationNameRec;  {sender's location name}
         recvrName:  PPCPortRec;       {reserved}
      END;
```

The `sessionID` field corresponds to the session reference number created by the PPC Toolbox. This is a 32-bit number that uniquely identifies a PPC Toolbox session (or connection) with another application. The `name` and `location` fields contain the sender's port name and location name. If the sending application is on the same computer as the receiving application, you can determine the sending application's process serial number by calling the `GetProcessSerialNumberFromPortName` function.

Sending High-Level Events

You use the `PostHighLevelEvent` function to send a high-level event to another application. When doing so, you need to provide six pieces of information:

- an event record with the event class and event ID assigned appropriately
- the identity of the recipient of the event
- a unique number that identifies the communication associated with this particular event
- a data buffer that can contain optional data
- the length of the data buffer
- options determining how the event is posted

Note
To send an Apple event, use the Apple Event Manager function `AESend`. The Apple Event Manager uses the Event Manager to post Apple events. For information on posting Apple events, see *Inside Macintosh: Interapplication Communication*. u

When you post a high-level event to an application on the same computer, you can specify its recipient in one of four ways:

- by port name and location name (specified in a target ID record)
- by a session reference number
- by the application's creator signature
- by a process serial number

To specify the recipient of a high-level event sent across a network, you can use only the receiving application's port name and location name or its session reference number. You can use any of the four ways when sending high-level events to applications on the local computer.

You specify the recipient of a high-level event in the `receiverID` parameter when you use the `PostHighLevelEvent` function. To specify a port name and location name, provide the address of a target ID record in the `receiverID` parameter. To specify a process serial number, provide its address in the `receiverID` parameter. To specify a session reference number, or signature, provide the data in the `receiverID` parameter.

When you are replying to a high-level event, it is easy to identify the recipient because you can use the target ID record that you receive from `AcceptHighLevelEvent`, the session reference number contained in that target ID record, or the process serial number (if the receiving process is local). Note that replying by session reference number is always the fastest way to respond to a high-level event.

When you are not replying to a previous event, you need to determine the identity of the target application yourself. You can use one of several methods to do this. If the target application is on the local computer, you can search for that application's creator signature or its process serial number by calling the `GetProcessInformation` function. See the chapter "Process Manager" in *Inside Macintosh: Processes* for a detailed explanation of the `GetProcessInformation` function and for examples of how to use it to generate a list of process serial numbers of all open processes on the local computer.

If the application to which you want to send a high-level event is located on a remote computer, you need to identify it either by its session reference number or by its port name and location name. You can call the `PPCBrowser` function to let the user browse for a specific port. You can call the `IPCListPorts` function to obtain a list of all ports registered with the target PPC Toolbox. See the chapter "Program-to-Program Communications Toolbox" in *Inside Macintosh: Interapplication Communication* for an explanation of both of these functions.

As just described, you can identify the recipient of the high-level event in one of four ways. Listing 2-17 illustrates how to send a high-level event to an application on the local computer using the application's creator signature. In this example, an application is sending a high-level event to the application with the creator signature of `'boff'`. The specific high-level event being sent is identified by the event class `'boff'` and the event ID `'cmd1'`.

Listing 2-17 Posting a high-level event by application signature

```
PROCEDURE MyPostTest;
VAR
   myEvent:    EventRecord;   {an event record}
   myRecvID:   OSType;        {receiver ID}
   myOpts:     LongInt;       {posting options}
   myErr:      OSErr;
BEGIN
   myEvent.what := kHighLevelEvent;
   myEvent.message := LongInt('boff');       {event class}
   myEvent.where := Point(LongInt('cmd1'));  {event ID}
   {the receiver is identified by its signature and }
   { a return receipt is requested}
   myOpts := receiverIDisSignature + nReturnReceipt;
   myRecvID := 'boff';                    {receiver's signature}
   myErr := PostHighLevelEvent(myEvent, Ptr(myRecvID), 0, NIL, 0,
                                 myOpts);
   IF myErr <> noErr THEN
      DoError(myErr);
END;
```

In this example of using the `PostHighLevelEvent` function, there is no additional data to transmit, so the sending application provides `NIL` as the pointer to the data buffer and sets the buffer length to 0. The `myOpts` variable specifies posting options.

Posting options are of two types: delivery options and options associated with the `receiverID` parameter. You can specify one or more delivery options to indicate if you want the other application to receive the event at the next opportunity and to indicate if you want acknowledgment that the other application received the event. You use the options associated with the `receiverID` parameter to indicate how you are specifying the recipient of the event. To set the various posting options, use these constants:

```
CONST nAttnMsg               = $00000001;{give this message priority}
      nReturnReceipt         = $00000200;{return receipt requested}
      receiverIDisTargetID   = $00005000;{ID is port name and location name}
      receiverIDisSessionID  = $00006000;{ID is PPC session ref number}
      receiverIDisSignature  = $00007000;{ID is creator signature}
      receiverIDisPSN        = $00008000;{ID is process serial number}
```

When you specify the receiving application in the `receiverID` parameter, you can use these constants to specify the receiver of the event by port name and location name, session reference number, process serial number, or signature. Any of these specifications allows you to send an event to another application on the local computer. For example, in Listing 2-17 the `myOpts` variable indicates that the receiver is identified by its creator signature, and the `myRecvID` variable contains the receiver's creator signature. To send events to an application on a remote computer, you can specify the recipient only by the session reference number or by the port name and location name.

When you specify the receiver of the event by port name and location name, use the `receiverIDisTargetID` constant in the posting options parameter and specify the address of a target ID record in the `receiverID` parameter.

```
TYPE  TargetID =
      RECORD
         sessionID:  LongInt;         {unused for posting}
         name:       PPCPortRec;      {recipient's port name}
         location:   LocationNameRec;{recipient's port loc}
         recvrName:  PPCPortRec;      {unused for posting}
      END;
```

When you pass a target ID record, you need to specify only the `name` and `location` fields. You can use the `IPCListPorts` function to list all of the existing port names along with information on whether the port will accept authenticated service on the computer specified by the location name. For information on how to use the `IPCListPorts` function, see the chapter "Program-to-Program Communications Toolbox" in *Inside Macintosh: Interapplication Communication*.

You can also use the `PPCBrowser` function to fill in a target ID record. Listing 2-18 on the next page illustrates how to use the `PPCBrowser` function to post a high-level event. In this example, the sending application wants to locate a dictionary application and have the dictionary return the definition of a word to it.

Listing 2-18 Using the `PPCBrowser` function to post a high-level event

```
FUNCTION MyPostWithPPCBrowser (aTextPtr: Ptr; textlength: LongInt): OSErr;
VAR
   myHLEvent:     EventRecord;
   myErr:         OSErr;
   myNumTries:    Integer;
   myPortInfo:    PortInfoRec;
   myTarget:      TargetID;
BEGIN
   {use PPCBrowser to get the target}
   myErr := PPCBrowser('Select an Application', 'Application', FALSE,
                       myTarget.location, myPortInfo, NIL, '');
   IF myErr = NoErr THEN
   BEGIN
      {copy port name into myTarget.name}
      myTarget.name := myPortInfo.name;

      myHLEvent.what := kHighLevelEvent;
      myHLEvent.message := LongInt('Dict');
      myHLEvent.where := Point(LongInt('Defn'));

      {if a connection is broken, then sessClosedErr is returned to }
      { PostHighLevelEvent; to reestablish the connection, just post }
      { the event one more time}
      myNumTries := 0;
      REPEAT
         myErr := PostHighLevelEvent(myHLEvent, @myTarget, 0, aTextPtr,
                                     textlength, receiverIDisTargetID);
         myNumTries := myNumTries + 1;
      UNTIL (myErr <> sessClosedErr) OR (myNumTries > 1);
   END;
   MyPostWithPPCBrowser := myErr;   {return any error}
END;
```

The application-defined function in Listing 2-18 uses the `PPCBrowser` function to display a dialog box asking the user to select a dictionary. (For additional information on the `PPCBrowser` function, see *Inside Macintosh: Interapplication Communication*.) If the user selects a dictionary, this code posts a high-level event to that dictionary application asking for the definition of the selected text. Note that the sending application and the receiving application must both agree that definition queries are to be of event class '`Dict`' and event ID '`Defn`'. It is necessary to define a private protocol only in cases in which no suitable Apple event exists.

Note
You should avoid passing handles to the receiving application in an attempt to share a block of data. It is better to put the relevant data into a buffer (as illustrated in Listing 2-18) and pass the address of the buffer. If you absolutely must share data by passing a handle, make sure that the block of data is located in the system heap. u

If a high-level event is posted successfully, `PostHighLevelEvent` returns the result code `noErr`, which indicates only that the event was successfully passed to the PPC Toolbox. Your application needs to call another Event Manager routine (`EventAvail`, `GetNextEvent`, or `WaitNextEvent`) to give the other application an opportunity to receive the event.

The event you send might require the other application to return some information to your application by sending a high-level event back to your application. You can scan for the response by using `GetSpecificHighLevelEvent`. If your application must wait for this event, you might want to display a wristwatch cursor or take other action as appropriate to your application. You also might want to implement a timeout mechanism in case your application never receives a response to the event.

Requesting Return Receipts

When you post a high-level event, you can request a return receipt by including the `nReturnReceipt` constant as one of the posting options. This requests that the Event Manager send your application a high-level event that tells you whether the other application accepted your event. Note that this does not necessarily mean that the other application performed any action you might have requested from it.

A **return receipt** is a high-level event having an event class and an event ID indicated by these two constants:

```
CONST HighLevelEventMsgClass  = 'jaym';
      rtrnReceiptMsgID        = 'rtrn';
```

Return receipts are posted by the Event Manager on the computer of the receiving application (and not by the receiving application itself). No data buffer is associated with a return receipt. However, the posting Event Manager sets the `modifiers` field of the high-level event record to one of the following values:

```
CONST msgWasNotAccepted       = 0;
      msgWasFullyAccepted     = 1;
      msgWasPartiallyAccepted = 2;
```

The `msgWasNotAccepted` constant indicates that your event was not accepted by the receiving application. This means that the receiving application was notified of the arrival of your event (through `WaitNextEvent`) but did not call `AcceptHighLevelEvent` to accept the event. The `msgWasFullyAccepted` constant indicates that the receiving application did call `AcceptHighLevelEvent` and retrieved all the data in the optional data buffer. The `msgWasPartiallyAccepted` constant indicates that the receiving application called `AcceptHighLevelEvent`, but the application's data buffer was too small to hold the data sent with your application, and the receiving application called `WaitNextEvent` before retrieving the rest of the buffer.

Note that a return receipt does not indicate the identity of the receiving application. To determine on whose behalf the Event Manager has sent you a particular return receipt, you need to call `AcceptHighLevelEvent`. When `AcceptHighLevelEvent` returns successfully, the `sender` parameter contains a target ID record with the fields filled in for the receiving application. With return receipts, the `msgLen` parameter is 0, the `msgBuff` parameter is `NIL`, and the `msgRefCon` parameter contains the unique number of the `refCon` parameter of the original high-level event sender (that is, your application).

Handling Apple Events

If your application uses high-level events, your application must respond to the required Apple events sent by the Finder. The four required Apple events are Open Application, Open Documents, Print Documents, and Quit Application. See *Inside Macintosh: Interapplication Communication* for information on how to handle the required Apple events.

When your application receives a high-level event (as indicated by the `kHighLevelEvent` constant in the `what` field of the event record), and if your application supports Apple events, call the `AEProcessAppleEvent` function. The `AEProcessAppleEvent` function provides an easy way for your application to identify the event class and event ID of the Apple event and to direct the Apple Event Manager to call the code in your program that handles the Apple event.

To send Apple events to other applications, use the `AESend` function.

To ensure compatibility and smooth interaction with other Macintosh applications, you should use the Apple event protocol for high-level events whenever possible. By implementing the capabilities to send Apple events to and receive Apple events from other applications, you allow other applications to interact with your application and provide enhanced capabilities to your users.

See *Inside Macintosh: Interapplication Communication* for complete information on how to send and receive Apple events.

Event Manager Reference

This section describes the data structures and routines for the Event Manager and Operating System Event Manager. It also describes the `'SIZE'` resource.

Data Structures

This section describes the event record, target ID record, high-level event message record, and structure of the Operating System event queue. The Event Manager uses event records to return information about events. You can use a target ID record to specify or identify the address of another application or process with which your application is communicating. If your application supplies a filter function as a parameter to the `GetSpecificHighLevelEvent` function, your filter function receives information about high-level events in a high-level event message record.

The Event Record

When your application uses an Event Manager routine to retrieve an event, the Event Manager returns information about the retrieved event in an event record. The `EventRecord` data type defines the event record.

```
TYPE  EventRecord =
      RECORD
         what:       Integer;       {event code}
         message:    LongInt;       {event message}
         when:       LongInt;       {ticks since startup}
         where:      Point;         {mouse location}
         modifiers:  Integer;       {modifier flags}
      END;
```

Field descriptions

`what` The `what` field indicates the type of event received. The type of event can be identified by these constants:

```
CONST
nullEvent         = 0; {no other pending events}
mouseDown         = 1; {mouse button pressed}
mouseUp           = 2; {mouse button released}
keyDown           = 3; {key pressed}
keyUp             = 4; {key released}
autoKey           = 5; {key repeatedly held down}
updateEvt         = 6; {window needs updating}
diskEvt           = 7; {disk inserted}
activateEvt       = 8; {activate/deactivate window}
osEvt             = 15;{operating-system event-- }
                       { resume, suspend, or }
                       { mouse-moved}
kHighLevelEvent   = 23;{high-level event}
```

Note that in System 7, event types with the values 9 through 14 are undefined and reserved for future use by Apple. All other values for the `what` field are also reserved for use by Apple.

`message` Additional information associated with the event. The interpretation of this information depends on the event type. The contents of the `message` field for each event type are summarized here:

Event type	Event message
null, mouse-up, mouse-down	Undefined.
key-up, key-down, auto-key	Character code and virtual key code in low-order word. For Apple Desktop Bus (ADB) keyboards, the low byte of the high-order word contains the ADB address of the keyboard where the keyboard event occurred. The high byte of the high-order word is reserved.
update, activate	Pointer to the window to update, activate, or deactivate.
disk-inserted	Drive number in low-order word, File Manager result code in high-order word.
resume	The `suspendResumeMessage` constant in bits 24–31 and a 1 in bit 0 to indicate the event is a resume event. Bit 1 contains either a 1 or a 0 to indicate if Clipboard conversion is required, and bits 2–23 are reserved.
suspend	The `suspendResumeMessage` constant in bits 24–31 and a 0 in bit 0 to indicate the event is a suspend event. Bit 1 is undefined, and bits 2–23 are reserved.
mouse-moved	The `mouseMovedMessage` constant in bits 24–31. Bits 2–23 are reserved, and bit 0 and bit 1 are undefined.
high-level	Class of events to which the high-level event belongs. The `message` and `where` fields of a high-level event define the specific type of high-level event received.

`when` The `when` field indicates the time when the event was posted (in ticks since system startup).

`where` For low-level events and operating-system events, the `where` field contains the location of the cursor at the time the event was posted (in global coordinates).

For high-level events, the `where` field contains a second event specifier, the event ID. The event ID defines the particular type of event within the class of events defined by the `message` field of the high-level event. For high-level events, you should interpret the `where` field as having the data type `OSType`, not `Point`.

`modifiers` The `modifiers` field contains information about the state of the modifier keys and the mouse button at the time the event was posted. For activate events, this field also indicates whether the window should be activated or deactivated. In System 7 it also indicates whether the mouse-down event caused your application to switch to the foreground.

Each of the modifier keys is represented by a specific bit in the `modifiers` field of the event record. Figure 2-5, on page 2-20, shows how to interpret the `modifiers` field. The modifier keys include the Option, Command, Caps Lock, Control, and Shift keys. If your application attaches special meaning to any of these keys in combination with other keys or when the mouse button is down, you can test the state of the `modifiers` field to determine the action your application should take. For example, you can use this information to determine whether the user pressed the Command key and another key to make a menu choice.

The Target ID Record

When you send a high-level event to another application, you can use the target ID record to specify the recipient of the event. When you receive a high-level event, the `AcceptHighLevelEvent` function uses a target ID record to return information about the sender of the event.

The `TargetID` data type defines the target ID record.

```
TYPE  TargetID =
      RECORD
         sessionID:  LongInt;          {session reference number}
         name:       PPCPortRec;       {port name}
         location:   LocationNameRec;  {location name}
         recvrName:  PPCPortRec;       {reserved}
      END;
```

Field descriptions

`sessionID` For high-level events that your application receives, this field contains the session reference number created by the PPC Toolbox. This is a 32-bit number that uniquely identifies a PPC Toolbox session (or connection) with another application. This field is not used by your application when sending a high-level event to another process. (To send a high-level event that specifies the recipient by session reference number, provide a pointer to a session reference number in the `receiverID` parameter and use the `receiverIDisSessionID` constant in the `postingOptions` parameter to `PostHighLevelEvent`.)

`name` For high-level events that your application receives, this field contains a PPC port record that specifies the port name of the process from which the high-level event originated. When sending a high-level event to a process on a local or remote computer, you can specify the port name of the recipient process in a PPC port record that you provide in this field.

	If the sending application is on the same computer as the receiving application, you can determine the sending application's process serial number by calling the `GetProcessSerialNumberFromPortName` function.
`location`	For high-level events that your application receives, this field contains a location name record that identifies the location name of the process from which the high-level event originated. When sending a high-level event to a process on a local or remote computer, you can specify the location name of the recipient process in a location name record that you provide in this field.
`recvrName`	This field is reserved.

The High-Level Event Message Record

You can search your application's high-level event queue for a specific high-level event by using the `GetSpecificHighLevelEvent` function and providing a filter function. Your filter function receives a pointer to a high-level event message record that contains information about a high-level event. (See "Filter Function for Searching the High-Level Event Queue" on page 2-114 for information on how to define a filter function.)

The `HighLevelEventMsg` data type defines the structure of a high-level event message record.

```
TYPE  HighLevelEventMsg =
      RECORD
         HighLevelEventMsgHeaderLength:      Integer;
         version:                            Integer;
         reserved1:                          LongInt;
         theMsgEvent:                        EventRecord;
         userRefCon:                         LongInt;
         postingOptions:                     LongInt;
         msgLength:                          LongInt;
      END;
```

Field descriptions

`HighLevelEventMsgHeaderLength`	Reserved for use by the Event Manager.
`version`	Reserved for use by the Event Manager.
`reserved1`	Reserved for use by the Event Manager.
`theMsgEvent`	The event record of a high-level event. Your filter function can compare the fields of this event record to determine whether the high-level event is the desired event. If your filter function finds the desired event, it should call `AcceptHighLevelEvent` to accept the event and remove the event from the high-level event queue, and return `TRUE` as its function result.

`userRefCon`	A unique number that identifies the communication associated with this event.
`postingOptions`	Reserved for use by the Event Manager.
`msgLength`	Reserved for use by the Event Manager.

The Event Queue

The event queue is a standard Macintosh Operating System queue that the Operating System Event Manager maintains. Only mouse-up, mouse-down, key-up, key-down, auto-key, and disk-inserted events are stored in the Operating System event queue. In most cases, your application should not access the event queue directly. Instead you usually use the `WaitNextEvent` function, which can retrieve events from this queue as well as from other sources.

The event queue consists of a header followed by the actual entries in the queue. The event queue has the same header as all standard Macintosh Operating System queues. The `Qhdr` data type defines the queue header.

```
TYPE  QHdr =
      RECORD
         qFlags:  Integer;    {queue flags}
         qHead:   QElemPtr;   {first queue entry}
         qTail:   QElemPtr;   {last queue entry}
      END;
```

The `EvQEl` data type defines an entry in the Operating System event queue.

```
TYPE  EvQEl =
      RECORD
         qLink:         QElemPtr;   {next queue entry}
         qType:         Integer;    {queue type (ORD(evType))}
         evtQWhat:      Integer;    {event code}
         evtQMessage:   LongInt;    {event message}
         evtQWhen:      LongInt;    {ticks since startup}
         evtQWhere:     Point;      {mouse location}
         evtQModifiers: Integer;    {modifier flags}
      END;
```

Each entry in the event queue begins with 4 bytes of flags followed by a pointer to the next queue entry. The flags are maintained by and internal to the Operating System Event Manager. The queue entries are linked by pointers, and the first field of the `EvQEl` data type, which represents the structure of a queue entry, begins with a pointer to the next queue entry. Thus you cannot directly access the flags using the `EvQEl` data type.

Event Manager Routines

The Event Manager includes routines for receiving events, receiving and sending high-level events, and searching for specific high-level events. The Event Manager also provides routines for converting between process serial numbers and port names, getting information about the state of the mouse button, reading the keyboard, and getting timing information.

Receiving Events

You can use the `WaitNextEvent` or `GetNextEvent` function to retrieve an event from the Event Manager and remove the event from the event stream. To provide greater support for multitasking, however, you should use the `WaitNextEvent` function instead of `GetNextEvent` whenever possible. You can use the `EventAvail` function to look at an event without removing it from the event stream. You can use the `AcceptHighLevelEvent` function to get additional information associated with a high-level event and `GetSpecificHighLevelEvent` to search for a specific high-level event.

The `FlushEvents` procedure removes all low-level events from the Operating System event queue. In general, your application should not empty the event queue.

You can use the `SystemClick` procedure to route events to desk accessories when necessary. The `SystemTask` and `SystemEvent` routines are used by the Event Manager, and your application usually does not need to call these two routines.

You usually use the functions provided by the Toolbox Event Manager to retrieve events from the event stream. Even if you are interested only in the events stored in the Operating System event queue, you can retrieve these events using the Toolbox Event Manager by setting the event mask to mask out all events except keyboard, mouse, and disk-inserted events. However, you can choose to use Operating System Event Manager routines to perform this task.

The Operating System Event Manager provides two functions, `GetOSEvent` and `OSEventAvail`, to retrieve events from the Operating System event queue. In most cases, your application will not need to use these two functions.

If your application needs to receive key-up events, you can change the system event mask of your application using the `SetEventMask` procedure. The `GetEvQHdr` function returns a pointer to the header of the Operating System event queue.

WaitNextEvent

You can use the `WaitNextEvent` function to retrieve events one at a time from the Event Manager.

```
FUNCTION WaitNextEvent (eventMask: Integer;
                        VAR theEvent: EventRecord; sleep: LongInt;
                        mouseRgn: RgnHandle): Boolean;
```

`eventMask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You can use these constants to specify the event mask:

```
CONST
mDownMask     = 2;      {mouse-down event    (bit 1)}
mUpMask       = 4;      {mouse-up event      (bit 2)}
keyDownMask   = 8;      {key-down event      (bit 3)}
keyUpMask     = 16;     {key-up event        (bit 4)}
autoKeyMask   = 32;     {auto-key event      (bit 5)}
updateMask    = 64;     {update event        (bit 6)}
diskMask      = 128;    {disk-inserted event (bit 7)}
activMask     = 256;    {activate event      (bit 8)}
highLevelEventMask
              = 1024;   {high-level event    (bit 10)}
osMask        = -32768; {operating-system    (bit 15)}
```

To accept all events, you can specify the `everyEvent` constant as the event mask:

```
CONST
everyEvent    = -1;     {every event}
```

If no event of any of the designated types is available, `WaitNextEvent` returns a null event. `WaitNextEvent` determines the next available event to return based on the `eventMask` parameter and the priority of the event.

Events not designated by the event mask remain in the event stream until retrieved by an application. Low-level events in the Operating System event queue are kept in the queue until they are retrieved by your application or another application or until the queue becomes full. Once the queue becomes full, the Operating System Event Manager begins discarding the oldest events in the queue.

`theEvent` The next available event of the specified type or types. The `WaitNextEvent` function removes the returned event from the event stream and returns the information about the event in an event record. The event record includes the type of event received and other information. See “The Event Record,” beginning on page 2-79, for a description of the fields in the event record.

In addition to the event record, high-level events can contain additional data; you use the `AcceptHighLevelEvent` or `AEProcessAppleEvent` functions to get additional data associated with these events.

`sleep`	The number of ticks (a tick is approximately 1/60 of a second) indicating the amount of time your application is willing to relinquish the processor if no events (other than null events) are pending for your application. If you specify a value greater than 0 for the `sleep` parameter, your application relinquishes the processor for the specified time or until an event occurs. You should usually specify a value greater than 0 for the `sleep` parameter to allow background processes to receive processing time. You should not set the `sleep` parameter to a value greater than the number of ticks returned by `GetCaretTime` if your application provides text-editing capabilities. When the specified time expires, and if there are no pending events for your application, `WaitNextEvent` returns a null event in the parameter `theEvent`.
`mouseRgn`	A handle to a region that specifies a region inside of which mouse movement does not cause mouse-moved events. In other words, your application receives mouse-moved events only when the cursor is outside the specified region. You should specify the region in global coordinates. If you pass an empty region or a `NIL` region handle, the Event Manager does not report mouse-moved events to your application. Note that your application should recalculate the `mouseRgn` parameter when it receives a mouse-moved event, or it will continue to receive mouse-moved events as long as the cursor position is outside the original `mouseRgn`.

DESCRIPTION

The `WaitNextEvent` function returns `FALSE` as its function result if the event being returned is a null event or if `WaitNextEvent` has intercepted the event; otherwise, `WaitNextEvent` returns `TRUE`. The `WaitNextEvent` function calls the Operating System Event Manager function `SystemEvent` to determine whether the event should be handled by the application or the Operating System.

If no events are pending for your application, `WaitNextEvent` waits for a specified amount of time for an event. (During this time, processing time may be allocated to background processes.) If an event occurs, it is returned as the value of the parameter `theEvent`, and `WaitNextEvent` returns a function result of `TRUE`. If the specified time expires and there are no pending events for your application, `WaitNextEvent` returns a null event in `theEvent` and a function result of `FALSE`.

Before returning an event to your application, `WaitNextEvent` performs other processing and may intercept the event.

The `WaitNextEvent` function intercepts Command–Shift–number key sequences and calls the corresponding 'FKEY' resource to perform the associated action. The Event Manager's processing of Command–Shift–number key sequences with numbers 3 through 9 can be disabled by setting the `ScrDmpEnable` global variable (a byte) to 0.

The `WaitNextEvent` function also makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

The `WaitNextEvent` function also calls the `SystemTask` procedure, which gives time to each open desk accessory or device driver to perform any periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

Some high-level events may be fully specified by their event records only, while others may include additional information in an optional buffer. To get any additional information and to find the sender of the event, use the `AcceptHighLevelEvent` function.

If the returned event is a high-level event and your application supports Apple events, use the Apple Event Manager function `AEProcessAppleEvent` to respond to the Apple event and to get additional information associated with the Apple event.

SPECIAL CONSIDERATIONS

In System 7, if your application is in the foreground and the user opens a desk accessory or other item from the Apple menu, clicks in the window belonging to another application or desk accessory, or chooses another process from the Application menu, the next event reported to your application by the `WaitNextEvent` function is a suspend event. After your application is switched out, the Event Manager directs events (other than events your application can receive in the background) to the newly activated process until the user switches back to your application or another application.

Note

In a single-application environment in System 6, and in a multiple-application environment in which the desk accessory is launched in the application's partition (for example, a desk accessory opened by the user from the Apple menu while holding down the Option key), the Event Manager handles events for desk accessories in a slightly different manner.

In these environments, when mouse-up, activate, update, and keyboard events (including keyboard equivalents of menu commands) occur, the Event Manager checks to see whether the active window belongs to a desk accessory and whether the desk accessory can handle the event. If so, it sends the event to the desk accessory and `WaitNextEvent` returns `FALSE` to your application. Also note that in these environments, the Event Manager returns `TRUE` for mouse-down events, regardless of whether the mouse-down event is for a desk accessory or not. For mouse-down events in these situations, if the mouse button was pressed while the cursor was in a desk accessory window (as indicated by the `inSystem` constant returned by the `FindWindow` function), your application should call the `SystemClick` procedure. The `SystemClick` procedure handles mouse-down events as appropriate for desk accessories, including sending your application an activate event to deactivate its front window if the desk accessory window needs to be activated. u

SEE ALSO

For examples that use the `WaitNextEvent` function, see Listing 2-1 on page 2-23 and Listing 2-2 on page 2-24.

To get information about the sender of a high-level event and to retrieve any additional data associated with the high-level event, see the description of the `AcceptHighLevelEvent` function on page 2-90. For details on how to process an Apple event, see the description of the `AEProcessAppleEvent` function in *Inside Macintosh: Interapplication Communication*.

For information on how to retrieve an event without removing it from the event stream, see the description of the `EventAvail` function, immediately following.

EventAvail

You can use the `EventAvail` function to retrieve the next available event from the Event Manager without removing the returned event from your application's event stream.

```
FUNCTION EventAvail (eventMask: Integer;
                     VAR theEvent: EventRecord): Boolean;
```

`eventMask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. You can use these constants to specify the event mask:

```
CONST
mDownMask    = 2;       {mouse-down event    (bit 1)}
mUpMask      = 4;       {mouse-up event      (bit 2)}
keyDownMask  = 8;       {key-down event      (bit 3)}
keyUpMask    = 16;      {key-up event        (bit 4)}
autoKeyMask  = 32;      {auto-key event      (bit 5)}
updateMask   = 64;      {update event        (bit 6)}
diskMask     = 128;     {disk-inserted event (bit 7)}
activMask    = 256;     {activate event      (bit 8)}
highLevelEventMask
             = 1024;    {high-level event    (bit 10)}
osMask       = -32768;  {operating-system    (bit 15)}
```

To accept all events, you can specify the `everyEvent` constant as the event mask:

```
CONST
everyEvent   =  -1;     {every event}
```

If no event of any of the designated types is available, `EventAvail` returns a null event.

`theEvent` The next available event of the specified type or types. The `EventAvail` function does not remove the returned event from the event stream, but does return the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

`EventAvail` returns `FALSE` as its function result if the event being returned is a null event; otherwise, `EventAvail` returns `TRUE`.

Like `WaitNextEvent`, the `EventAvail` function calls the `SystemTask` procedure to give time to each open desk accessory or device driver to perform any periodic action defined for it. The `EventAvail` function also makes the alarm go off if the alarm is set and the current time is the alarm time. The user sets the alarm using the Alarm Clock desk accessory.

SPECIAL CONSIDERATIONS

If `EventAvail` returns a low-level event from the Operating System event queue, the event will not be accessible later if, in the meantime, the event queue becomes full and the event is discarded from it; however, this is not a common occurrence.

SEE ALSO

See “The Event Record,” beginning on page 2-79, for a description of the fields in the event record.

GetNextEvent

Although you should normally use `WaitNextEvent`, you can also use the `GetNextEvent` function to retrieve events one at a time from the Event Manager.

```
FUNCTION GetNextEvent (eventMask: Integer;
                       VAR theEvent: EventRecord): Boolean;
```

`eventMask` A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants (listed in “Setting the Event Mask” beginning on page 2-26). If no event of any of the designated types is available, `GetNextEvent` returns a null event.

`theEvent` The next available event of the specified type or types. The `GetNextEvent` function removes the returned event from the event stream and returns the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

GetNextEvent returns FALSE as its function result if the event being returned is a null event or if GetNextEvent has intercepted the event; otherwise, GetNextEvent returns TRUE. The GetNextEvent function calls the Operating System Manager function SystemEvent to determine whether the event should be handled by the application or the Operating System.

Like WaitNextEvent, the GetNextEvent function calls the SystemTask procedure to give time to each open desk accessory or device driver to perform any periodic action defined for it. The GetNextEvent function also makes the alarm go off if the alarm is set and the current time is the alarm time. (The user sets the alarm using the Alarm Clock desk accessory.)

The GetNextEvent function also intercepts Command–Shift–number key sequences and calls the corresponding 'FKEY' resource to perform the associated action. The Event Manager's processing of Command–Shift–number key sequences with numbers 3 through 9 can be disabled by setting the ScrDmpEnable global variable (a byte) to 0.

SPECIAL CONSIDERATIONS

For greater support of the multitasking environment, your application should use WaitNextEvent instead of GetNextEvent whenever possible. If your application does call GetNextEvent, it should also call the SystemTask procedure.

SEE ALSO

See "The Event Record," beginning on page 2-79, for a description of the fields in the event record. For information on the SystemTask procedure, see page 2-95.

AcceptHighLevelEvent

After receiving a high-level event (other than an Apple event), use the AcceptHighLevelEvent function to get any additional information associated with the event.

```
FUNCTION AcceptHighLevelEvent (VAR sender: TargetID;
                                 VAR msgRefcon: LongInt;
                                 msgBuff: Ptr;
                                 VAR msgLen: LongInt): OSErr;
```

sender	Identifies the sender of the event; this information is returned in a target ID record. The sender parameter contains the session reference number that identifies the connection with the other application and the port name and location name of the sender.

`msgRefcon`	Uniquely identifies the communication associated with this event. If you send a response to this event, you should specify the same value for the `msgRefcon` parameter so that the sender of the event can associate the reply with the original request.
`msgBuff`	Specifies where the `AcceptHighLevelEvent` function should return any additional data associated with the event. Your application is responsible for allocating the memory for the additional data pointed to by the `msgBuff` parameter and for setting the `msgLen` parameter to the number of bytes that you have allocated for the data. If the `msgBuff` parameter points to an area in memory that is not large enough to hold all the data associated with the event, `AcceptHighLevelEvent` returns as much data as the specified memory area can hold, returns the amount of data remaining in the `msgLen` parameter, and returns the result code `bufferIsSmall`.
`msgLen`	Contains the size of the data (in bytes) pointed to by the `msgBuff` parameter. If `AcceptHighLevelEvent` returns the result code `bufferIsSmall`, the `msgLen` parameter contains the number of bytes remaining. You can call `AcceptHighLevelEvent` again to receive the rest of the data.

DESCRIPTION

When your application receives a high-level event, you can use the `AcceptHighLevelEvent` function to get additional data associated with the event. The `AcceptHighLevelEvent` function returns information that identifies the sender of the event and the unique message reference constant of the event.

Your application should allocate memory for any additional data associated with the event, then supply a pointer to the data area and also provide the length in bytes of the data area.

SPECIAL CONSIDERATIONS

The `AcceptHighLevelEvent` function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `AcceptHighLevelEvent` function are

Trap macro	Selector
`_OSDispatch`	$0033

RESULT CODES

`noErr`	0	No error
`bufferIsSmall`	–607	Buffer is too small
`noOutstandingHLE`	–608	No outstanding high-level event

SEE ALSO

For details on how to process an Apple event using the `AEProcessAppleEvent` function, see *Inside Macintosh: Interapplication Communication*.

GetSpecificHighLevelEvent

You can use the `GetSpecificHighLevelEvent` function to select and optionally retrieve a specific high-level event from your application's high-level event queue.

```
FUNCTION GetSpecificHighLevelEvent
             (aFilter: GetSpecificFilterProcPtr;
              yourDataPtr: UNIV Ptr; VAR err: OSErr): Boolean;
```

`aFilter`	Specifies the filter function that `GetSpecificHighLevelEvent` should use to search for a specific event. `GetSpecificHighLevelEvent` calls your filter function once for each event in your application's high-level event queue until your filter function returns `TRUE` or the end of the queue is reached.
`yourDataPtr`	Specifies the criteria your filter function should use to select a specific event. For example, in the `yourDataPtr` parameter you can specify a reference constant to search for a particular event, a pointer to a target ID record to search for a specific sender of an event, or an event class to search for a specific class of event.
`err`	`GetSpecificHighLevelEvent` returns in this parameter a value indicating if any errors occurred. The `err` parameter contains the `noErr` constant if no errors occurred or `noOutstandingHLE` if no high-level events are pending in your application's high-level event queue.

DESCRIPTION

You can use the `GetSpecificHighLevelEvent` function to search for a specific high-level event in your application's high-level event queue. You provide a pointer to a filter function as one of the parameters to `GetSpecificHighLevelEvent`. The `GetSpecificHighLevelEvent` function calls your filter function once for every event in your application's high-level event queue, until your filter function returns `TRUE` or the end of the queue is reached.

The `GetSpecificHighLevelEvent` function passes the value you specify in the `yourDataPtr` parameter to your filter function. Your filter function also receives as parameters the event record associated with the high-level event and the target ID record that identifies the sender of the event. Your filter function can compare the contents of the `yourDataPtr` parameter with any of the other information it receives.

If your filter function finds a match, it can call `AcceptHighLevelEvent` if necessary, and then return `TRUE`. If your filter function does not find a match, then it should return `FALSE`.

If your filter function returns TRUE, the GetSpecificHighLevelEvent function returns TRUE. If your filter function returns FALSE for all high-level events in your application's event queue, or if there are no high-level events in the queue, GetSpecificHighLevelEvent returns FALSE.

SPECIAL CONSIDERATIONS

The GetSpecificHighLevelEvent function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the GetSpecificHighLevelEvent function are

Trap macro	Selector
_OSDispatch	$0045

SEE ALSO

See "Filter Function for Searching the High-Level Event Queue" on page 2-114 for more information about how to define a filter function and the parameters that GetSpecificHighLevelEvent passes to your filter function.

FlushEvents

The FlushEvents procedure removes low-level events from the Operating System event queue. Note that FlushEvents does not remove any types of events not stored in the Operating System event queue.

You can choose to use the FlushEvents procedure when your application first starts to empty the Operating System event queue of any keystrokes or mouse events generated by the user while the Finder loaded your application. In general, however, your application should not empty the queue at any other time as this loses user actions and makes your application and the computer appear unresponsive to the user.

```
PROCEDURE FlushEvents (whichMask: Integer; stopMask: Integer);
```

whichMask	A value that indicates which kinds of low-level events are to be removed from the Operating System event queue; this parameter is interpreted as a sum of event mask constants. The whichMask and stopMask parameters together specify which events to remove.

`stopMask`	A value that limits which low-level events are to be removed from the Operating System event queue; this parameter is interpreted as a sum of event mask constants. `FlushEvents` does not remove any low-level events that are specified by the `stopMask` parameter. To remove all events specified by the `whichMask` parameter, specify 0 as the `stopMask` parameter.

DESCRIPTION

`FlushEvents` removes only low-level events stored in the Operating System event queue; it does not remove activate, update, operating-system, or high-level events.

You specify which low-level events to remove using the `whichMask` and `stopMask` parameters. `FlushEvents` removes the low-level events specified by the `whichMask` parameter, up to but not including the first event of any type specified by the `stopMask` parameter.

If the event queue doesn't contain any of the events specified by the `whichMask` parameter, `FlushEvents` does not remove any events from the queue.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register D0 with the event mask (`whichMask`) and stop mask before calling `FlushEvents`. When `FlushEvents` returns, register D0 contains 0 if all events were removed from the queue or, if all events were not removed from the queue, an event code that specifies the type of event that caused the removal process to stop.

Registers on entry

D0	Event mask (low-order word)
	Stop mask (high-order word)

Registers on exit

D0	0 if all events were removed from the queue, or the event code of the event that stopped the search (low-order word)

SEE ALSO

See "Setting the Event Mask" beginning on page 2-26 for information on how to specify an event mask.

SystemClick

After receiving a mouse-down event, your application should call the Window Manager function `FindWindow` to determine where the cursor was when the mouse button was pressed. If `FindWindow` returns the `inSysWindow` constant, call the `SystemClick` procedure to handle the event.

```
PROCEDURE SystemClick (theEvent: EventRecord;
                       theWindow: WindowPtr);
```

`theEvent`	The event record for the event.
`theWindow`	The window in which the mouse-down event occurred. Pass the window pointer returned by `FindWindow` in this parameter.

DESCRIPTION

If a mouse-down event occurred in a desk accessory's window, the `SystemClick` procedure determines which part of the desk accessory's window the cursor was in when the mouse button was pressed and routes the event to the appropriate desk accessory as necessary.

If the mouse button was pressed while the cursor was in the content region of the desk accessory's window and the window is active, `SystemClick` sends the mouse-down event to the desk accessory to process. If the mouse-down event occurred in the content region of the window and the window is inactive, `SystemClick` makes it the active window. It does this by sending your application an activate event to deactivate its front window and directing an event to the desk accessory to activate its window.

If the mouse button was pressed while the cursor was in the drag region or go-away region, `SystemClick` calls the Window Manager routine `DragWindow` or `TrackGoAway`, as appropriate. If `TrackGoAway` reports that the user closed the desk accessory, `SystemClick` sends a close message to the desk accessory.

SEE ALSO

See "The Event Record," beginning on page 2-79, for a description of the fields in the event record.

SystemTask

In a multiple-application environment, the `WaitNextEvent` function is responsible for giving time to each open desk accessory or driver to perform any periodic action. You should not call `SystemTask` if your application calls `WaitNextEvent`.

If your application calls `GetNextEvent`, your application should call the `SystemTask` procedure.

```
PROCEDURE SystemTask;
```

DESCRIPTION

The `SystemTask` procedure gives time to each open desk accessory or driver to perform the periodic action defined for it. A desk accessory or device driver specifies how often the periodic action should occur, and `SystemTask` gives time to the desk accessory or device driver at the appropriate interval.

If your application calls `GetNextEvent`, your application should call `SystemTask` at least every sixtieth of a second. This usually corresponds to calling `SystemTask` once each time through your event loop. If your application does a large amount of processing, you may need to call `SystemTask` more than once in your event loop.

SEE ALSO

For a description of the `WaitNextEvent` function and the `GetNextEvent` function, see page 2-85 and page 2-89, respectively.

SystemEvent

The `WaitNextEvent` and `GetNextEvent` functions call the `SystemEvent` function. In most cases your application should not call the `SystemEvent` function.

The `SystemEvent` function determines if a specific event should be handled by the application or the Operating System.

```
FUNCTION SystemEvent (theEvent: EventRecord): Boolean;
```

`theEvent`	The event record for the event.

DESCRIPTION

`SystemEvent` returns `FALSE` as its function result if the event should be handled by the application; otherwise, `SystemEvent` takes any appropriate actions and returns `TRUE`.

For activate, update, mouse-up, and keyboard events (including keyboard equivalents of commands), `SystemEvent` checks to see whether the active window belongs to a desk accessory and whether that desk accessory can handle that type of event. If so, `SystemEvent` sends the event to the desk accessory and returns `TRUE`. Otherwise, `SystemEvent` returns `FALSE`.

For mouse-down events and null events, `SystemEvent` returns `FALSE`.

For disk-inserted events, `SystemEvent` attempts to mount the disk using the `PBMountVol` function but returns `FALSE` so that the application can perform further processing if necessary.

ASSEMBLY-LANGUAGE INFORMATION

If the `SEvtEnb` global variable (a byte) contains 0, `SystemEvent` always returns `FALSE`.

SEE ALSO

See “The Event Record,” beginning on page 2-79, for a description of the fields in the event record. For a description of the `PBMountVol` function, see the chapter “File Manager” in *Inside Macintosh: Files*.

GetOSEvent

The Toolbox Event Manager calls the `GetOSEvent` function to retrieve low-level events stored in the Operating System event queue. In most cases your application should not use this function.

```
FUNCTION GetOSEvent (mask: Integer;
                     VAR theEvent: EventRecord): Boolean;
```

`mask`	A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. `GetOSEvent` returns only low-level events stored in the Operating System event queue; it does not return activate, update, operating-system, or high-level events. If no low-level event of any of the designated types is available, `GetOSEvent` returns a null event.
`theEvent`	The next available low-level event of the specified type or types in the Operating System event queue. The `GetOSEvent` function removes the returned event from the Operating System event queue and returns the information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

The `GetOSEvent` function retrieves and removes an event from the Operating System event queue. `GetOSEvent` returns `FALSE` as its function result if the event being returned is a null event; otherwise, `GetOSEvent` returns `TRUE`. `GetOSEvent` does not intercept or respond to the event in any way. It also does not process Command–Shift–number key combinations or process any alarms set by the user through the Alarm Clock desk accessory.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the address of an event record and register D0 with the event mask before invoking `GetOSEvent`. When `GetOSEvent` returns, register D0 indicates whether the returned event is a null event or an event other than a null event and the returned event is accessible through register A0.

Registers on entry

A0	Address of event record
D0	Event mask (low-order word)

Registers on exit

A0	Address of event record
D0	0 if `GetOSEvent` returns any event other than a null event, or –1 if it returns a null event (low-order byte)

SEE ALSO

See "The Event Record," beginning on page 2-79, for a description of the fields in the event record. See "Setting the Event Mask," beginning on page 2-26, for information on how to specify an event mask.

OSEventAvail

The Toolbox Event Manager uses the `OSEventAvail` function to retrieve an event from the Operating System event queue without removing it. In most cases your application does not need to use this function.

```
FUNCTION OSEventAvail (mask: Integer;
                        VAR theEvent: EventRecord): Boolean;
```

`mask`	A value that indicates which kinds of events are to be returned; this parameter is interpreted as a sum of event mask constants. `OSEventAvail` returns only low-level events stored in the Operating System event queue; it does not return activate, update, operating-system, or high-level events. If no low-level event of any of the designated types is available, `OSEventAvail` returns a null event.
`theEvent`	The next available event of the specified type or types. The `OSEventAvail` function does not remove the returned event from the Operating System event queue but does return information about the event in an event record. The event record includes the type of event received and other information.

DESCRIPTION

The `OSEventAvail` function retrieves an event from the Operating System event queue without removing it from the queue. The `OSEventAvail` function returns `FALSE` as its function result if the event being returned is a null event; otherwise, `OSEventAvail` returns `TRUE`.

`OSEventAvail` does not intercept or respond to the event in any way. It also does not process Command–Shift–number key combinations or process any alarms set by the user through the Alarm Clock desk accessory.

SPECIAL CONSIDERATIONS

If the `OSEventAvail` function returns a low-level event from the Operating System event queue, the event will not be accessible later if, in the meantime, the event queue becomes full and the event is discarded from it; however, this is not a common occurrence.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the address of an event record and register D0 with the event mask before invoking `OSEventAvail`. When `OSEventAvail` returns, register D0 indicates whether the returned event is a null event or some other event, and the returned event is accessible through register A0.

Registers on entry

A0	Address of event record
D0	Event mask (low-order word)

Registers on exit

A0	Address of event record
D0	0 if `OSEventAvail` returns any event other than a null event, or –1 if it returns a null event (low-order byte)

SEE ALSO

See "The Event Record," beginning on page 2-79, for a description of the fields in the event record. See "Setting the Event Mask," beginning on page 2-26, for information on how to specify an event mask

SetEventMask

The `SetEventMask` procedure sets the system event mask of your application to the specified mask. Your application should not call the `SetEventMask` procedure to disable any event types from being posted. Use `SetEventMask` only to enable key-up events if your application needs to respond to key-up events.

```
PROCEDURE SetEventMask (theMask: Integer);
```

`theMask`	An event mask that specifies which events should be posted in the Operating System event queue.

DESCRIPTION

The `SetEventMask` procedure sets the system event mask of your application according to the parameter `theMask`. The Operating System Event Manager posts only low-level events (other than update or activate events) corresponding to bits in the system event mask of the current process when posting events in the Operating System event queue. The system event mask of an application is initially set to post mouse-up, mouse-down, key-down, auto-key, and disk-inserted events into the Operating System event queue.

ASSEMBLY-LANGUAGE INFORMATION

The system event mask of the current application is available in the `SysEvtMask` system global variable.

SEE ALSO

For additional information on event masks, see "Setting the Event Mask" beginning on page 2-26.

GetEvQHdr

The Event Manager uses the `GetEvQHdr` function to get a pointer to the header of the Operating System event queue. In most cases your application should not call the `GetEvQHdr` function.

```
FUNCTION GetEvQHdr: QHdrPtr;
```

DESCRIPTION

The `GetEvQHdr` function returns a pointer to the header of the Operating System event queue.

ASSEMBLY-LANGUAGE NOTE

The `EventQueue` system global variable contains the header of the event queue.

SEE ALSO

See "The Event Queue" on page 2-83 for information on the structure of the Operating System event queue.

Sending Events

You can send events to other applications or processes using the `PostHighLevelEvent` function. To send Apple events to other applications, use the Apple Event Manager function `AESend`. The Operating System Event Manager also provides the `PPostEvent` and `PostEvent` functions for posting low-level events to the Operating System event queue. The `PostEvent` function is used by the Toolbox Event Manager. In most cases your application should not use the `PostEvent` function.

PostHighLevelEvent

You can use the `PostHighLevelEvent` function to send a high-level event to another application.

```
FUNCTION PostHighLevelEvent (theEvent: EventRecord;
                             receiverID: Ptr; msgRefcon: LongInt;
                             msgBuff: Ptr; msgLen: LongInt;
                             postingOptions: LongInt): OSErr;
```

`theEvent` The event to send. Your application should fill out the `what`, `message`, and `where` fields of the event record. Specify the `kHighLevelEvent` constant in the `what` field, the event class of the high-level event in the `message` field, and the event ID in the `where` field. You do not need to fill out the `when` or `modifiers` fields; the Event Manager automatically assigns the appropriate values to these fields when you send the message.

`receiverID`

The recipient of the high-level event. When sending an event to another application on the local computer, you can specify the recipient of the event by session reference number, process serial number, signature, or port name and location name. When sending an event to an application on a remote computer, you can specify the recipient only by the session reference number or by the port name and location name.

To specify a port name and location name, provide the address of a target ID record in the `receiverID` parameter. To specify a process serial number, provide its address in the `receiverID` parameter. To specify a session reference number, or signature, provide the data (cast to the `Ptr` data type) in the `receiverID` parameter.

`msgRefcon` A unique number that identifies the communication associated with this event. Your application can set this field to any value it chooses. If you are replying to a high-level event, you should use the same value in the `msgRefcon` parameter as specified in the high-level event that originated the request.

`msgBuff` A pointer to a data buffer that contains any additional data for the event.

`msgLen` The size (in bytes) of the data buffer pointed to by the `msgBuff` parameter.

`postingOptions`

Options associated with the `receiverID` parameter and delivery options associated with the event. You can specify one or more delivery options to indicate whether you want the other application to receive the event at the next opportunity and to indicate whether you want acknowledgment that the event was received by the other application. You use the options associated with the `receiverID` parameter to indicate how you are specifying the recipient of the event—whether by port name and location name in a target ID record, by session reference number, by process serial number, or by signature.

You can use a combination of these constants in the `postingOptions` parameter:

```
CONST
nAttnMsg
   = $00000001; {give this message priority}
nReturnReceipt
   = $00000200; {return receipt requested}
receiverIDisTargetID
   = $00005000; {ID is port name and location name}
receiverIDisSessionID
   = $00006000; {ID is PPC session reference number}
receiverIDisSignature
   = $00007000; {ID is creator signature}
receiverIDisPSN
   = $00008000; {ID is process serial number}
```

DESCRIPTION

The `PostHighLevelEvent` function posts the high-level event to the specified process.

If the application to which you are sending a high-level event terminates, you receive the result code `sessionClosedErr` the next time your application calls `PostHighLevelEvent` to send another high-level event to the terminated application. If you do not care about any state information about that session, you can just resend your event. Otherwise, you must restart another session and resend your event.

If your application is running in the background and posts a high-level event that requires the network authentication dialog box to be displayed, `PostHighLevelEvent` returns the `noUserInteractionAllowed` result code, does not display the network authentication dialog box, and does not send the event. If your application receives the `noUserInteractionAllowed` result code, you can use the Notification Manager to inform the user that your application needs attention. When the user brings your application to the foreground, you can repost the event. If the reposting is successful, your application can continue to post high-level events without further user interaction. Note that `PostHighLevelEvent` can return `noUserInteractionAllowed` only on the first posting of a high-level event to a remote target.

SPECIAL CONSIDERATIONS

The `PostHighLevelEvent` function may move or purge memory. You should not call this function from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `PostHighLevelEvent` function are

Trap macro	Selector
`_OSDispatch`	$0034

SEE ALSO

For details on how to send Apple events to other applications using the `AESend` function, see *Inside Macintosh: Interapplication Communication*.

RESULT CODES

`noErr`	0	No error
`connectionInvalid`	–609	Connection is invalid
`noUserInteractionAllowed`	–610	Cannot interact directly with user
`sessionClosedErr`	–917	Session closed

PPostEvent

In most cases your application does not need to post events in the Operating System event queue; however, if you must do so, you can use the `PPostEvent` function.

```
FUNCTION PPostEvent (eventCode: Integer; eventMsg: LongInt;
                     VAR qEl: EvQElPtr): OSErr;
```

`eventCode`	A value that indicates the type of event to post into the Operating System event queue. The types of events that can be posted in this queue are represented by these constants: `mouseDown`, `mouseUp`, `keyDown`, `keyUp`, `autoKey`, and `diskEvt`. Do not attempt to post any other type of event in the Operating System event queue.
`eventMsg`	A long integer that contains the contents of the `message` field for the event that `PPostEvent` should post in the queue.
`qEl`	`PPostEvent` returns a pointer to the event queue entry of the posted event in this parameter.

DESCRIPTION

In the `eventCode` and `eventMsg` parameters, you specify the value for the `what` and `message` fields of the event's event record. The `PPostEvent` function fills out the `when`, `where`, and `modifiers` fields of the event record with the current time, current mouse location, and current state of the modifier keys and mouse button.

The `PPostEvent` function returns a pointer to the event queue entry of the posted event in the `qEl` parameter. You can change any fields of the posted event by changing the fields of its event queue entry. For example, you can change the posted event's modifier keys by changing the value of the `evtQModifiers` field of the event queue entry.

The `PPostEvent` function posts only events that are enabled by the system event mask. If the event queue is full, `PPostEvent` removes the oldest event in the queue and posts the new event.

s **WARNING**
Do not post any events other than mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events in the Operating System event queue. Attempting to post other events into the Operating System event queue interferes with the internal operation of the Event Manager. s

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 and register D0 before invoking `PPostEvent`. The `PPostEvent` function returns values in registers A0 and D0.

Registers on entry

A0	Event number (low-order word)
D0	Event message (long)

Registers on exit

A0	Pointer to an event queue entry (long)
D0	Result code (low-order word)

RESULT CODES

`evtNotEnb`	1	Event type not valid—event not posted
`noErr`	0	No error

SEE ALSO

For a description of the entries in the event queue, see “The Event Queue” on page 2-83.

PostEvent

The Toolbox Event Manager uses the `PostEvent` function to post events into the Operating System event queue. In most cases your application should not call the `PostEvent` function.

```
FUNCTION PostEvent (eventNum: Integer; eventMsg: LongInt): OSErr;
```

`eventNum`	A value that indicates the type of event to post into the Operating System event queue. The types of events that can be posted in this queue are represented by these constants: `mouseDown`, `mouseUp`, `keyDown`, `keyUp`, `autoKey`, and `diskEvt`. Do not attempt to post any other type of event in the Operating System event queue.
`eventMsg`	A long integer that contains the contents of the `message` field for the event that `PostEvent` should post in the queue.

DESCRIPTION

In the `eventNum` and `eventMsg` parameters, you specify the value for the `what` and `message` fields of the event's event record. The `PostEvent` function fills out the `when`, `where`, and `modifiers` fields of the event record with the current time, current mouse location, and current state of the modifier keys and mouse button.

The `PostEvent` function posts only events that are enabled by the system event mask. If the event queue is full, `PostEvent` removes the oldest event in the queue and posts the new event.

Note that if you use `PostEvent` to repost an event, the `PostEvent` function fills out the `when`, `where`, and `modifier` fields of the event record, giving these fields of the reposted event different values from the values contained in the original event.

s **WARNING**
Do not post any events other than mouse-down, mouse-up, key-down, key-up, auto-key, and disk-inserted events in the Operating System event queue. Attempting to post other events into the Operating System event queue interferes with the internal operation of the Event Manager. s

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with the event code and register D0 with the event message before invoking `PostEvent`. When `PostEvent` returns, register D0 contains the result code.

Registers on entry

A0	Event number (low-order word)
D0	Event message (long)

Registers on exit

D0	Result code (low-order word)

RESULT CODES

`evtNotEnb`	1	Event type not valid—event not posted
`noErr`	0	No error

Converting Process Serial Numbers and Port Names

The Event Manager provides two functions to convert between process serial numbers and port names (`GetProcessSerialNumberFromPortName` and `GetPortNameFromProcessSerialNumber`). Both functions are intended to map serial numbers to port names (or vice versa) for applications open on the local computer. They do not return useful results for applications open on remote computers.

GetProcessSerialNumberFromPortName

Use `GetProcessSerialNumberFromPortName` to get the process serial number of a process.

```
FUNCTION GetProcessSerialNumberFromPortName
            (portName: PPCPortRec;
             VAR PSN: ProcessSerialNumber): OSErr;
```

`portName`	The port name registered to a process whose serial number you want.
`PSN`	Returns the process serial number of the process designated by the `portName` parameter. You can use the returned process serial number to send a high-level event to that process. Do not interpret the value of the process serial number.

DESCRIPTION

The `GetProcessSerialNumberFromPortName` function returns the process serial number of the process registered at a specific port.

SPECIAL CONSIDERATIONS

The `GetProcessSerialNumberFromPortName` function does not move or purge memory but for other reasons should not be called from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetProcessSerialNumberFromPortName` function are

Trap macro	Selector
`_OSDispatch`	$0035

RESULT CODES

`noErr`	0	No error
`noPortErr`	–903	Invalid port name

SEE ALSO

For a description of the `PPCPortRec` data type, see the chapter “Program-to-Program Communications Toolbox” in *Inside Macintosh: Interapplication Communication*.

GetPortNameFromProcessSerialNumber

Use `GetPortNameFromProcessSerialNumber` to get the port name of a process.

```
FUNCTION GetPortNameFromProcessSerialNumber
            (VAR portName: PPCPortRec;
             PSN: ProcessSerialNumber): OSErr;
```

`portName`	Returns the port name of the process designated by the `PSN` parameter. You can use the returned port name to send a high-level event to that process.
`PSN`	The process serial number of the process whose port name you want.

DESCRIPTION

The `GetPortNameFromProcessSerialNumber` function returns the port name registered to a process having a specific process serial number.

SPECIAL CONSIDERATIONS

The `GetPortNameFromProcessSerialNumber` function does not move or purge memory but for other reasons should not be called from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPortNameFromProcessSerialNumber` function are

Trap macro	Selector
`_OSDispatch`	$0046

RESULT CODES

`noErr`	0	No error
`procNotFound`	–600	No eligible process with specified process serial number

SEE ALSO

For a description of the `PPCPortRec` data type, see the chapter “Program-to-Program Communications Toolbox” in *Inside Macintosh: Interapplication Communication*.

Reading the Mouse

The Event Manager provides routines you can use to get information about the mouse. You can get the current mouse location using the `GetMouse` procedure. You can use the `Button` function to determine whether the user pressed the mouse button. After receiving a mouse-down event, you can use the `StillDown` function to determine whether the mouse button is still down, and you can use `WaitMouseUp` to determine if the user subsequently released the mouse.

GetMouse

You can use the `GetMouse` procedure to obtain the current mouse location.

```
PROCEDURE GetMouse (VAR mouseLoc: Point);
```

`mouseLoc`	Returns the current mouse location in local coordinates of the current graphics port (for example, the active window). Note that this value differs from the value of the `where` field of the event record, which specifies the mouse location in global coordinates.

Button

You can use the `Button` function to determine whether the user pressed the mouse button.

```
FUNCTION Button: Boolean;
```

DESCRIPTION

The `Button` function looks in the Operating System event queue for a mouse-down event. If it finds one, the `Button` function returns `TRUE`; otherwise, it returns `FALSE`. To determine whether the mouse button is still down after a mouse-down event, use the `StillDown` function.

SEE ALSO

See “The Event Queue” on page 2-83 for information about the Operating System event queue.

StillDown

After receiving a mouse-down event, you can use the `StillDown` function to determine if the mouse button is still down.

```
FUNCTION StillDown: Boolean;
```

DESCRIPTION

The `StillDown` function looks in the Operating System event queue for a mouse event. If it finds one, the `StillDown` function returns `FALSE`. If it does not find any mouse events pending in the Operating System event queue, the `StillDown` function returns `TRUE`.

SEE ALSO

See "The Event Queue" on page 2-83 for information about the Operating System event queue.

WaitMouseUp

After receiving a mouse-down event, you can use the `WaitMouseUp` function to determine if the user subsequently released the mouse.

```
FUNCTION WaitMouseUp: Boolean;
```

DESCRIPTION

The `WaitMouseUp` function looks in the Operating System event queue for a mouse-up event. If it finds one, the `WaitMouseUp` function removes the mouse-up event from the queue and returns `TRUE`. If it does not find any mouse events pending in the Operating System event queue, the `WaitMouseUp` function returns `FALSE`.

SEE ALSO

See "The Event Queue" on page 2-83 for information about the Operating System event queue.

Reading the Keyboard

The Event Manager reports keyboard events one at a time at your application's request when you use the `WaitNextEvent`, `EventAvail`, or `GetNextEvent` function. In addition to getting keyboard events when the user presses or releases a key, you can directly read the keyboard (and keypad) at any time using the `GetKeys` procedure.

You can also use the `KeyTranslate` function to convert virtual key codes to character code values using a specified 'KCHR' resource.

GetKeys

You can use the `GetKeys` procedure to obtain the current state of the keyboard.

```
PROCEDURE GetKeys (VAR theKeys: KeyMap);
```

`theKeys` Returns the current state of the keyboard, including the keypad, if any. The `GetKeys` procedure returns this information using the `KeyMap` data type.

```
TYPE KeyMap = PACKED ARRAY[0..127] OF Boolean;
```

Each key on the keyboard or keypad corresponds to an element in the `KeyMap` array. The index for a particular key is the same as the key's virtual key code minus 1. For example, the key with virtual key code 38 (the "J" key on the Apple Keyboard II) can be accessed as `KeyMap[37]` in the returned array. A `KeyMap` element is `TRUE` if the corresponding key is down and `FALSE` if it isn't. The maximum number of keys that can be down simultaneously is two character keys plus any combination of the five modifier keys.

DESCRIPTION

You can use the `GetKeys` procedure to determine the current state of the keyboard at any time. For example, you can determine whether one of the modifier keys is down by itself or in combination with another key using the `GetKeys` procedure.

KeyTranslate

You can use the `KeyTranslate` function to convert a virtual key code to a character code based on a `'KCHR'` resource. The `KeyTranslate` function is also available as the `KeyTrans` function.

```
FUNCTION KeyTranslate (transData: Ptr; keycode: Integer;
                        VAR state: LongInt): LongInt;
```

`transData` A pointer to the 'KCHR' resource that you want the `KeyTranslate` function to use when converting the key code to a character code.

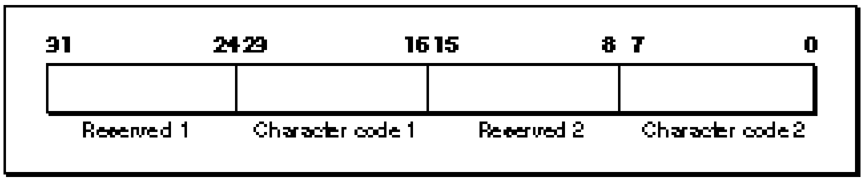
`keycode` A 16-bit value that your application should set so that bits 0–6 contain the virtual key code and bit 7 contains either 1 to indicate an up stroke or 0 to indicate a down stroke of the key. Bits 8–15 have the same interpretation as the high byte of the `modifiers` field of the event record and should be set according to the needs of your application.

`state` A value that your application should set to 0 the first time it calls `KeyTranslate` or any time your application calls `KeyTranslate` with a different `'KCHR'` resource. Thereafter, your application should pass the same value for the `state` parameter as `KeyTranslate` returned in the previous call.

DESCRIPTION

The `KeyTranslate` function returns a 32-bit value that gives the character code for the virtual key code specified by the `keycode` parameter. Figure 2-17 shows the structure of the 32-bit number that `KeyTranslate` returns.

Figure 2-17 Structure of the `KeyTranslate` function result



The `KeyTranslate` function returns the values that correspond to one or possibly two characters that are generated by the specified virtual key code. For example, a given virtual key code might correspond to an alphabetic character with a separate accent character. For example, when the user presses Option-E followed by N, you can map this through the `KeyTranslate` function using the U.S. `'KCHR'` resource to produce ´n, which `KeyTranslate` returns as two characters in the bytes labeled Character code 1 and Character code 2. If `KeyTranslate` returns only one character code, it is always in the byte labeled Character code 2. However, your application should always check both bytes labeled Character code 1 and Character code 2 in Figure 2-17 for possible values that map to the virtual key code.

SEE ALSO

For additional information on the `'KCHR'` resource and the `KeyTranslate` function, see *Inside Macintosh: Text*.

Getting Timing Information

You can get the current number of ticks since the system last started up using the `TickCount` function. You can use this function to compare the number of ticks that have expired since a given event or other action occurred.

By using the `GetDblTime` function, you can get the suggested maximum difference in ticks that should exist to consider two mouse events a double click. The user can adjust this value using the Mouse control panel. Using the `GetCaretTime` function you can get the suggested maximum difference in ticks that should exist between blinks of the caret in editable text. The user can adjust this value using the General Controls panel.

TickCount

You can use the `TickCount` function to get the current number of ticks (a tick is approximately $^1/_{60}$ of a second) since the system last started up.

```
FUNCTION TickCount: LongInt;
```

DESCRIPTION

The `TickCount` function returns a long integer that indicates the current number of ticks since the system last started up. You can use this value to compare the number of ticks that have elapsed since a given event or other action occurred. For example, you could compare the current value returned by `TickCount` with the value of the `when` field of an event record.

The tick count is incremented during the vertical retrace interrupt, but this interrupt can be disabled. Your application should not rely on the tick count to increment with absolute precision. Your application also should not assume that the tick count always increments by 1; an interrupt task might keep control for more than one tick. If your application keeps track of the previous tick count and then compares this value with the current tick count, your application should compare the two values by checking for a “greater than or equal” condition rather than “equal to previous tick count plus 1.”

s **WARNING**
Don’t rely on the tick count being exact; it’s usually accurate to within one tick, but this level of accuracy is not guaranteed. s

ASSEMBLY-LANGUAGE NOTE

The value returned by `TickCount` is also accessible in the global variable `Ticks`.

GetDblTime

To determine whether a sequence of mouse events constitutes a double click, your application measures the elapsed time (in ticks) between a mouse-up event and a mouse-down event. If the time between the two mouse events is less than the value returned by `GetDblTime`, your application should interpret the two mouse events as a double click.

```
FUNCTION GetDblTime: LongInt;
```

DESCRIPTION

The `GetDblTime` function returns the suggested maximum elapsed time, in ticks, between a mouse-up event and a mouse-down event. The user can adjust this value using the Mouse control panel.

If your application distinguishes a double click of the mouse from a single click, your application should use the value returned by `GetDblTime` to make this distinction. If your application uses TextEdit, the TextEdit procedures automatically recognize and handle double clicks of text within a TextEdit edit record by appropriately highlighting or unhighlighting the selection.

ASSEMBLY-LANGUAGE NOTE

The value returned by `GetDblTime` is also accessible in the system global variable `DoubleTime`.

GetCaretTime

You can use the `GetCaretTime` function to get the suggested difference in ticks that should exist between blinks of the caret (usually a vertical bar marking the insertion point) in editable text. The user can adjust this value using the General Controls panel.

```
FUNCTION GetCaretTime: LongInt;
```

DESCRIPTION

If your application supports editable text, your application should use the value returned by `GetCaretTime` to determine how often to blink the caret. If your application uses only TextEdit, you can use TextEdit procedures to automatically blink the caret at the time interval that the user specifies in the General Controls panel.

ASSEMBLY-LANGUAGE NOTE

The value returned by `GetCaretTime` is also accessible in the system global variable `CaretTime`.

Application-Defined Routine

When you use `GetSpecificHighLevelEvent`, you supply a filter function so that your application can search for a specific event in the high-level event queue of your application.

Filter Function for Searching the High-Level Event Queue

This section describes the filter function that you can provide to `GetSpecificHighLevelEvent`. For example, you might use a filter function to search for a high-level event sent from a specific application.

MyFilter

When you use `GetSpecificHighLevelEvent` to search the high-level event queue of your application for a specific event, you supply a pointer to a filter function. `GetSpecificHighLevelEvent` calls your filter function once for each event in the high-level event queue until your filter function returns `TRUE` or the end of the queue is reached. Your filter function can examine each event and determine whether that event is the desired event. If so, your filter function should return `TRUE`.

Here's how you declare the filter function `MyFilter`:

```
FUNCTION MyFilter (yourDataPtr: Ptr;
                   msgBuff: HighLevelEventMsgPtr;
                   sender: TargetID): Boolean;
```

`yourDataPtr`	Specifies the criteria your filter function should use to select a specific event. For example, you can specify the `yourDataPtr` parameter as a reference constant to search for a particular event, as a pointer to a target ID record to search for a specific sender of an event, or as an event class to search for a specific class of event.
`msgBuff`	Contains a pointer to a record of data type `HighLevelEventMsg`, which provides: the event record for the high-level event and the reference constant of the event. The `HighLevelEventMsg` data type is described in "The High-Level Event Message Record" on page 2-82.
`sender`	Contains the target ID record of the application that sent the event. The `TargetID` data type is described in "The Target ID Record" on page 2-81.

DESCRIPTION

Your filter function can compare the contents of the `yourDataPtr` parameter with the contents of the `msgBuff` and `sender` parameters. If your filter function finds a match, it can call `AcceptHighLevelEvent`, if necessary, and your filter function should return `TRUE`. If your filter function does not find a match, it should return `FALSE`.

SEE ALSO

For information about how to specify your filter function to the `GetSpecificHighLevelEvent` function, see page 2-92.

Resource

This section explains the structure of a `'SIZE'` resource and the meaning of each of its fields. You are responsible for creating the information in this resource.

The Size Resource

Every application executing in System 7, as well as every application executing under MultiFinder, should contain a size (`'SIZE'`) resource. One of the principal functions of the `'SIZE'` resource is to inform the Operating System about the memory size requirements for the application so that the Operating System can set up an appropriately sized partition for the application. The `'SIZE'` resource is also used to indicate certain scheduling options to the Operating System, such as whether the application can accept suspend and resume events. The `'SIZE'` resource in System 7 contains additional information indicating whether the application is 32-bit clean, whether it supports stationery documents, whether it uses TextEdit's inline input services, whether the application wishes to receive notification of the termination of any applications it has launched, and whether the application wishes to receive high-level events.

A `'SIZE'` resource consists of a 16-bit flags field followed by two 32-bit size fields. The flags field specifies operating characteristics of the application, and the size fields indicate the minimum and preferred partition sizes for the application. The minimum partition size is the actual limit below which your application will not run. The preferred partition size is the memory size at which your application can run most effectively and which the Operating System attempts to secure upon launching the application. If that amount of memory is unavailable, the application is placed into the largest contiguous block available, provided that it is larger than the specified minimum size.

Note

If the amount of available memory is between the minimum and the preferred sizes, the Finder displays a dialog box asking if the user wants to run the application using the amount of memory available. If your application does not have a `'SIZE'` resource, it is assigned a default partition size of 512 KB and the Process Manager uses a default value of FALSE for all specifications normally defined by constants in the flags field. u

When you define a `'SIZE'` resource, you should give it a resource ID of –1. A user can modify the preferred size in the Finder's information window for your application. If the user does alter the preferred partition size, the Operating System creates a new `'SIZE'` resource having resource ID 0. The Process Manager also creates a new `'SIZE'` resource when the user modifies any of the other settings in the resource.

In system software version 7.1 the user can also modify the minimum size in the Finder's information window for your application. In version 7.1, if the user alters either the minimum or the preferred partition size, the Operating System creates two new `'SIZE'` resources, one with resource ID 0 and one with resource ID 1.

At application launch time, the Process Manager looks for a `'SIZE'` resource with ID 0 for the preferred partition size; if this resource is not found, it uses your original `'SIZE'` resource with ID –1. In version 7.1, the Process Manager looks for a `'SIZE'` resource with ID 0 for the preferred size and looks for a `'SIZE'` resource with ID 1 for the minimum size; if these resources are not found, it uses your original `'SIZE'` resource with ID –1.

Listing 2-19 shows the structure of the `'SIZE'` resource in Rez format. See Listing 2-4 in "Creating a Size Resource," beginning on page 2-30 for a sample `'SIZE'` resource for an application.

Listing 2-19 A Rez template for a `'SIZE'` resource

```
type 'SIZE' {
   boolean  reserved;                  /*reserved*/
   boolean  ignoreSuspendResumeEvents, /*ignores suspend-resume events*/
            acceptSuspendResumeEvents; /*accepts suspend-resume events*/
   boolean  reserved;                  /*reserved*/
   boolean  cannotBackground,          /*can't use background null events*/
            canBackground;             /*can use background null events*/
   boolean  needsActivateOnFGSwitch,   /*needs activate event following */
                                       /* major switch*/
            doesActivateOnFGSwitch;    /*activates own windows in */
                                       /* response to OS events*/
   boolean  backgroundAndForeground,   /*app has a user interface*/
            onlyBackground;            /*app has no user interface*/
   boolean  dontGetFrontClicks,        /*don’t return mouse events */
                                       /* in front window on resume*/
            getFrontClicks;            /*do return mouse events */
                                       /* in front window on resume*/
   boolean  ignoreAppDiedEvents,       /*applications use this*/
            acceptAppDiedEvents;       /*app launchers use this*/
   boolean  not32BitCompatible,        /*works with 24-bit addr*/
            is32BitCompatible;         /*works with 24- or 32-bit addr*/
   boolean  notHighLevelEventAware,    /*can't use high-level events*/
            isHighLevelEventAware;     /*can use high-level events*/
   boolean  onlyLocalHLEvents,         /*only local high-level events*/
            localAndRemoteHLEvents;    /*also remote high-level events*/
   boolean  notStationeryAware,        /*can't use stationery documents*/
            isStationeryAware;         /*can use stationery documents*/
   boolean  dontUseTextEditServices,   /*can't use inline services*/
            useTextEditServices;       /*can use inline services*/
   boolean  reserved;                  /*reserved*/
   boolean  reserved;                  /*reserved*/
   boolean  reserved;                  /*reserved*/
                                       /*memory sizes are in bytes*/
   unsigned longint;                   /*preferred memory size*/
   unsigned longint;                   /*minimum memory size*/
};
```

The nonreserved bits in the flags field have the following meanings:

Flag descriptions

`acceptSuspendResumeEvents`
: When set, indicates that your application can process suspend and resume events (which the Operating System sends to your application before sending it into the background or when bringing it into the foreground).

Note

If you set the `acceptSuspendResumeEvents` flag, you should also set the `doesActivateOnFGSwitch` flag. u

`canBackground`
: When set, indicates that your application wants to receive null event processing time while in the background. If your application has nothing to do in the background, you should not set this flag.

`doesActivateOnFGSwitch`
: When set, indicates that your application takes responsibility for activating and deactivating any windows in response to a suspend or resume event. If the `acceptSuspendResumeEvents` flag is set, if the `doesActivateOnFGSwitch` flag is not set, and if your application is suspended, then your application receives an activate event following the suspend event. However, if you set the `doesActivateOnFGSwitch` flag, then your application won't receive activate events associated with operating-system events, and you must take care of activation and deactivation when it receives the corresponding suspend or resume event. This means that if a window of your application is frontmost, you should treat a suspend event as though a deactivate event were received as well (assuming that both the `doesActivateOnFGSwitch` and `acceptSuspendResumeEvents` flags are set). For example, you should hide scroll bars, hide any caret, and unhighlight any selected text if your application moves to the background. If you do not set this flag, the Process Manager creates an offscreen window to force the activate and deactivate events to occur.

`onlyBackground`
: When set, indicates that your application runs only in the background. Usually this is because it does not have a user interface and cannot run in the foreground.

`getFrontClicks`
: When set, indicates that your application is to receive the mouse-down and mouse-up events that are used to bring your application into the foreground when the user clicks in your

application's frontmost window. Typically, the user simply wants to bring your application into the foreground, so it is usually not desirable to receive the mouse events (which would probably move the insertion point or start drawing immediately, depending on the application). The Finder is one application, however, that has the `getFrontClicks` flag set.

When the user clicks in the front window of your application and causes a foreground switch, your application receives a resume event. Your application should activate its front window in response to the resume event. In this case if your application's `getFrontClicks` flag is not set, your application does not receive the associated mouse event that caused the foreground switch. If your application's `getFrontClicks` flag is set, your application does receive the associated mouse event.

Your application always receives the associated mouse event when the user clicks in one of your application's windows other than the front window and causes a foreground switch.

When your application receives a mouse-down event in System 7, your application can examine bit 0 of the `modifiers` field of the event record to determine if the mouse-down event caused a foreground switch. This information can be especially useful if your application sets its `getFrontClicks` flag. For example, your application can examine bit 0 to determine whether to process the mouse-down event (probably depending on whether the clicked item was visible before the foreground switch).

`acceptAppDiedEvents`

When set, indicates that your application is to be notified whenever an application launched by your application terminates or crashes. If the Process Manager is available, your application receives this information as an Apple event, the Application Died event. See the chapter "Process Manager" chapter in *Inside Macintosh: Processes* for more information about launching applications and receiving Application Died events.

Note

Some early versions of MultiFinder do not send application-died events, and your application should not depend on receiving them if it is running in System 6. These events are provided primarily for use by debuggers. u

`is32BitCompatible`

When set, indicates that your application can be run with the 32-bit Memory Manager. You should not set this flag unless you have thoroughly tested your application on a 32-bit system (such as a Macintosh IIci computer running System 7 in 32-bit mode or under A/UX).

`isHighLevelEventAware`

When set, indicates that your application can send and receive high-level events. If this flag is not set, the Event Manager does not give your application high-level events when you call

`WaitNextEvent`. There is no way to mask out specific types of high-level events; if this flag is set, your application receives all types of high-level events sent to your application.

Your application must support the four required Apple events if you set the `isHighLevelEventAware` flag. See *Inside Macintosh: Interapplication Communication* for information that describes how to respond to the four required Apple events.

`localAndRemoteHLEvents`
When set, indicates that your application is to be visible to applications running on other computers on a network (in addition to applications running on the local computer). If this flag is not set, your application does not receive high-level events across a network.

`isStationeryAware`
When set, indicates that your application can recognize stationery documents. If this flag is not set and the user opens a stationery document, the Finder duplicates the document and prompts the user for a name for the duplicate document. For information about how your application can use stationery documents, see the chapter "Finder Interface" in this book.

`useTextEditServices`
When set, indicates that your application can use the inline text services provided by TextEdit. See *Inside Macintosh: Text* for information about the inline input capabilities of TextEdit.

The numbers you specify as your application's preferred and minimum memory sizes depend on the particular memory requirements of your application. Your application's memory requirements depend on the size of your application's static heap, dynamic heap, A5 world, and stack. (See "Introduction to Memory Management" in *Inside Macintosh: Memory* for complete details about these areas of your application's partition.)

The static heap size includes objects that are always present during the execution of the application—for example, code segments, Toolbox data structures for window records, and so on.

Dynamic heap requirements depend on how many objects are created on a per-document basis (which may vary in size proportionally with the document itself) and the number of objects that are required for specific commands or functions.

The size of the A5 world depends on the amount of global data and the number of intersegment jumps the application contains.

Finally, the stack contains variables, return addresses, and temporary information. The application stack size varies among computers, so you should base your values for the stack size according to the stack size required on a Macintosh Plus (8 KB). The Process Manager automatically adjusts your requested amount of memory to compensate for the different stack sizes on different machines. For example, if you request 512 KB, more stack space (approximately 16 KB) will be allocated on machines with larger default stack sizes.

Summary of the Event Manager

Pascal Summary

Constants

```
CONST {event codes}
      nullEvent               = 0;        {no other pending events}
      mouseDown               = 1;        {mouse button pressed}
      mouseUp                 = 2;        {mouse button released}
      keyDown                 = 3;        {key pressed}
      keyUp                   = 4;        {key released}
      autoKey                 = 5;        {key repeatedly held down}
      updateEvt               = 6;        {window needs updating}
      diskEvt                 = 7;        {disk inserted}
      activateEvt             = 8;        {activate/deactivate window}
      osEvt                   = 15;       {operating-system events }
                                          { (suspend, resume, mouse-moved)}
      kHighLevelEvent         = 23;       {high-level events }
                                          { (includes Apple events)}

      {event masks}
      everyEvent              = -1;       {every event}
      mDownMask               =  2;       {mouse-down event       (bit 1)}
      mUpMask                 =  4;       {mouse-up event         (bit 2)}
      keyDownMask             =  8;       {key-down event         (bit 3)}
      keyUpMask               =  16;      {key-up event           (bit 4)}
      autoKeyMask             =  32;      {auto-key event         (bit 5)}
      updateMask              =  64;      {update event           (bit 6)}
      diskMask                =  128;     {disk-inserted event    (bit 7)}
      activMask               =  256;     {activate event         (bit 8)}
      highLevelEventMask      =  1024;    {high-level event       (bit 10)}
      osMask                  = -32768;   {operating-system event (bit 15)}

      {message codes for operating-system events}
      suspendResumeMessage    = $01;      {suspend or resume event}
      mouseMovedMessage       = $FA;      {mouse-moved event}
      osEvtMessageMask        = $FF000000;{can use to extract msg code}
```

```
{flags for suspend and resume events}
resumeFlag             = 1;          {resume event}
convertClipboardFlag   = 2;          {Clipboard conversion }
                                     { required}
{event message masks for keyboard events}
charCodeMask= $000000FF;             {use to get character code}
keyCodeMask = $0000FF00;             {use to get key code}
adbAddrMask = $00FF0000;             {ADB address for ADB keyboard}

{constants corresponding to bits in the modifiers field of event}
activeFlag  = 1;        {bit 0 of low byte--valid only for }
                        { activate and mouse-moved events}
btnState    = 128;      {bit 7 of low byte is mouse button state}
cmdKey      = 256;      {bit 0 of high byte}
shiftKey    = 512;      {bit 1 of high byte}
alphaLock   = 1024;     {bit 2 of high byte}
optionKey   = 2048;     {bit 3 of high byte}
controlKey  = 4096;     {bit 4 of high byte}

{high-level event posting options}
nAttnMsg               = $00000001;   {give this message priority}
priorityMask           = $000000FF;   {mask for priority options}
nReturnReceipt         = $00000200;   {return receipt requested}
systemOptionsMask      = $00000F00;
receiverIDisTargetID   = $00005000;   {ID is port name & location}
receiverIDisSessionID  = $00006000;   {ID is PPC session ref number}
receiverIDisSignature  = $00007000;   {ID is creator signature}
receiverIDisPSN        = $00008000;   {ID is process serial number}
receiverIDMask         = $0000F000;

{class and ID values for return receipt}
HighLevelEventMsgClass  = 'jaym';      {event class of return receipt}
rtrnReceiptMsgID        = 'rtrn';      {event ID of return receipt}

{modifiers values in return receipt}
msgWasNotAccepted       = 0;         {recipient did not accept }
                                     { the message}
msgWasFullyAccepted     = 1;         {recipient accepted the}
                                     { entire message}
msgWasPartiallyAccepted = 2;         {recipient did not accept }
                                     { the entire message}
```

Data Types

```
TYPE
   EventRecord =
   RECORD
      what:       Integer;       {event code}
      message:    LongInt;       {event message}
      when:       LongInt;       {ticks since startup}
      where:      Point;         {mouse location}
      modifiers:  Integer;       {modifier flags}
   END;

   KeyMap = PACKED ARRAY[0..127] OF Boolean; {records state of keyboard}

   TargetID =
   RECORD
      sessionID:        LongInt;          {session reference number (not }
                                          { used if posting an event)}
      name:             PPCPortRec;       {port name}
      location:         LocationNameRec;  {location name}
      recvrName:        PPCPortRec;       {reserved}
   END;

   TargetIDPtr          = ^TargetID;      {pointer to a target ID record}
   TargetIDHdl          = ^TargetIDPtr;   {handle to a target ID record}

   HighLevelEventMsg =
   RECORD
      HighLevelEventMsgHeaderLength:   Integer;       {reserved}
      version:                         Integer;       {reserved}
      reserved1:                       LongInt;       {reserved}
      theMsgEvent:                     EventRecord;   {event record}
      userRefCon:                      LongInt;       {reference constant}
      postingOptions:                  LongInt;       {reserved}
      msgLength:                       LongInt;       {reserved}
   END;

   HighLevelEventMsgPtr = ^HighLevelEventMsg;
   HighLevelEventMsgHdl = ^HighLevelEventMsgPtr;

   GetSpecificFilterProcPtr = ProcPtr;
```

```
EvQEl =                       {event queue entry}
RECORD
   qLink:         QElemPtr;   {next queue entry}
   qType:         Integer;    {queue type (ORD(evType))}
   evtQWhat:      Integer;    {event code}
   evtQMessage:   LongInt;    {event message}
   evtQWhen:      LongInt;    {ticks since startup}
   evtQWhere:     Point;      {mouse location}
   evtQModifiers: Integer;    {modifier flags}
END;

EvQElPtr = ^EvQEl;
```

Event Manager Routines

Receiving Events

```
FUNCTION WaitNextEvent       (eventMask: Integer; VAR theEvent: EventRecord;
                              sleep: LongInt; mouseRgn: RgnHandle): Boolean;
FUNCTION EventAvail          (eventMask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
FUNCTION GetNextEvent        (eventMask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
FUNCTION AcceptHighLevelEvent
                             (VAR sender: TargetID; VAR msgRefcon: LongInt;
                              msgBuff: Ptr; VAR msgLen: LongInt): OSErr;
FUNCTION GetSpecificHighLevelEvent
                             (aFilter: GetSpecificFilterProcPtr;
                              yourDataPtr: UNIV Ptr; VAR err: OSErr)
                              : Boolean;
PROCEDURE FlushEvents        (whichMask: Integer; stopMask: Integer);
PROCEDURE SystemClick        (theEvent: EventRecord; theWindow: WindowPtr);
PROCEDURE SystemTask;
FUNCTION SystemEvent         (theEvent: EventRecord): Boolean;
FUNCTION GetOSEvent          (mask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
FUNCTION OSEventAvail        (mask: Integer; VAR theEvent: EventRecord)
                              : Boolean;
PROCEDURE SetEventMask       (theMask: Integer);
FUNCTION GetEvQHdr           : QHdrPtr;
```

Sending Events

```
FUNCTION PostHighLevelEvent  (theEvent: EventRecord; receiverID: Ptr;
                              msgRefcon: LongInt; msgBuff: Ptr;
                              msgLen: LongInt; postingOptions: LongInt)
                              : OSErr;
FUNCTION PPostEvent          (eventCode: Integer; eventMsg: LongInt;
                              VAR qEl: EvQElPtr): OSErr;
FUNCTION PostEvent           (eventNum: Integer; eventMsg: LongInt): OSErr;
```

Converting Process Serial Numbers and Port Names

```
FUNCTION GetProcessSerialNumberFromPortName
                             (portName: PPCPortRec;
                              VAR PSN: ProcessSerialNumber): OSErr;
FUNCTION GetPortNameFromProcessSerialNumber
                             (VAR portName: PPCPortRec;
                              PSN: ProcessSerialNumber): OSErr;
```

Reading the Mouse

```
PROCEDURE GetMouse           (VAR mouseLoc: Point);
FUNCTION Button              : Boolean;
FUNCTION StillDown           : Boolean;
FUNCTION WaitMouseUp         : Boolean;
```

Reading the Keyboard

```
PROCEDURE GetKeys            (VAR theKeys: KeyMap);
{the KeyTranslate function is also available as the KeyTrans function}
FUNCTION KeyTranslate        (transData: Ptr; keycode: Integer;
                              VAR state: LongInt): LongInt;
```

Getting Timing Information

```
FUNCTION TickCount           : LongInt;
FUNCTION GetDblTime          : LongInt;
FUNCTION GetCaretTime        : LongInt;
```

Application-Defined Routine

Filter Function for Searching the High-Level Event Queue

```
FUNCTION MyFilter            (yourDataPtr: Ptr;
                              msgBuff: HighLevelEventMsgPtr;
                              sender: TargetID): Boolean;
```

C Summary

Constants

```
enum {
      /*event codes*/
      nullEvent               = 0,  /*no other pending events*/
      mouseDown               = 1,  /*mouse button pressed*/
      mouseUp                 = 2,  /*mouse button released*/
      keyDown                 = 3,  /*key pressed*/
      keyUp                   = 4,  /*key released*/
      autoKey                 = 5,  /*key repeatedly held down*/
      updateEvt               = 6,  /*window needs updating*/
      diskEvt                 = 7,  /*disk inserted*/
      activateEvt             = 8,  /*activate/deactivate window*/
      osEvt                   = 15, /*operating-system events */
                                    /* (suspend, resume, mouse-moved)*/

      /*event masks*/
      mDownMask               =  2,          /*mouse-down        (bit 1)*/
      mUpMask                 =  4,          /*mouse-up          (bit 2)*/
      keyDownMask             =  8,          /*key-down          (bit 3)*/
      keyUpMask               =  16,         /*key-up            (bit 4)*/
      autoKeyMask             =  32,         /*auto-key          (bit 5)*/
      updateMask              =  64,         /*update            (bit 6)*/
      diskMask                =  128,        /*disk-inserted     (bit 7)*/
      activMask               =  256,        /*activate          (bit 8)*/
      highLevelEventMask      =  1024,       /*high-level        (bit 10)*/
      osMask                  = -32768       /*operating-system  (bit 15)*/
};

enum {
      everyEvent                 = -1,          /*every event*/

      /*event message masks for keyboard events*/
      charCodeMask       = 0x000000FF,       /*use to get character code*/
      keyCodeMask        = 0x0000FF00,       /*use to get key code*/
      adbAddrMask        = 0x00FF0000,       /*ADB address if ADB keyboard*/
      osEvtMessageMask   = 0xFF000000,       /*can use to extract msg code*/
      /*message codes for operating-system events*/
      mouseMovedMessage       = 0xFA,        /*mouse-moved event*/
      suspendResumeMessage    = 0x01,        /*suspend or resume event*/
      /*flags for suspend and resume events*/
      resumeFlag              = 1,           /*resume event*/
      convertClipboardFlag    = 2,           /*Clipboard conversion */
                                             /* required*/

      /*constants corresponding to bits in the modifiers field of event*/
      activeFlag  = 1,        /*bit 0 of low byte--valid only for */
                              /* activate and mouse-moved events*/
      btnState    = 128,      /*bit 7 of low byte is mouse button state*/
      cmdKey      = 256,      /*bit 0 of high byte*/
      shiftKey    = 512,      /*bit 1 of high byte*/
      alphaLock   = 1024,     /*bit 2 of high byte*/
      optionKey   = 2048,     /*bit 3 of high byte*/
      controlKey  = 4096      /*bit 4 of high byte*/
};
enum {
      kHighLevelEvent         = 23, /*event code for high-level events */
                                    /* (includes Apple events)*/
      /*high-level event posting options*/
      receiverIDMask          = 0x0000F000,  /*mask for receiver ID bits*/
      receiverIDisPSN         = 0x00008000,  /*ID is proc serial number*/
      receiverIDisSignature   = 0x00007000,  /*ID is creator signature*/
      receiverIDisSessionID   = 0x00006000,  /*ID is session ref number*/
      receiverIDisTargetID    = 0x00005000,  /*ID is port name & location*/

      systemOptionsMask       = 0x00000F00,
      nReturnReceipt          = 0x00000200,  /*return receipt requested*/
      priorityMask            = 0x000000FF,  /*mask for priority options*/
      nAttnMsg                = 0x00000001,  /*give this message priority*/

      /*class and ID values for return receipt*/
      #define HighLevelEventMsgClass 'jaym'
      #define rtrnReceiptMsgID 'rtrn'

      /*modifiers values in return receipt*/
      msgWasPartiallyAccepted = 2,
      msgWasFullyAccepted     = 1,
      msgWasNotAccepted       = 0
};
```

Data Types

```
struct EventRecord {
      short    what;                   /*event code*/
      long     message;                /*event message*/
      long     when;                   /*ticks since startup*/
      Point    where;                  /*mouse location*/
      short    modifiers;              /*modifier flags*/
   };
typedef struct EventRecord EventRecord;

typedef long KeyMap[4];                /*records state of keyboard*/

struct TargetID {
      long              sessionID;     /*session reference number (not */
                                       /* used if posting an event)*/
      PPCPortRec        name;          /*port name*/
      LocationNameRec   location;      /*location name*/
      PPCPortRec        recvrName;     /*reserved*/
   };

typedef struct TargetID TargetID;
typedef TargetID *TargetIDPtr, **TargetIDHdl;

struct HighLevelEventMsg {
      unsigned short    HighLevelEventMsgHeaderLength;   /*reserved*/
      unsigned short    version;                         /*reserved*/
      unsigned long     reserved1;                       /*reserved*/
      EventRecord       theMsgEvent;                     /*event record*/
      unsigned long     userRefCon;                      /*ref constant*/
      unsigned long     postingOptions;                  /*reserved*/
      unsigned long     msgLength;                       /*reserved*/
   };

typedef struct HighLevelEventMsg HighLevelEventMsg;
typedef HighLevelEventMsg *HighLevelEventMsgPtr, **HighLevelEventMsgHdl;

   struct EvQEl {       /*event queue entry*/
      QElemPtr    qLink;            /*next queue entry*/
      short       qType;            /*queue type (evType)*/
      short       evtQWhat;         /*event code*/
      long        evtQMessage;      /*event message*/
      long        evtQWhen;         /*ticks since startup*/
      Point       evtQWhere;        /*mouse location*/
      short       evtQModifiers;    /*modifier flags*/
   };
   typedef struct EvQEl EvQEl;
   typedef EvQEl *EvQElPtr;

typedef pascal Boolean (*GetSpecificFilterProcPtr)
                              (void *yourDataPtr,
                              HighLevelEventMsgPtr msgBuff,
                              const TargetID *sender);
```

Event Manager Routines

Receiving Events

```
pascal Boolean WaitNextEvent(short eventMask, EventRecord *theEvent,
                              unsigned long sleep, RgnHandle mouseRgn);
pascal Boolean EventAvail   (short eventMask, EventRecord *theEvent);
pascal Boolean GetNextEvent (short eventMask, EventRecord *theEvent);
pascal OSErr AcceptHighLevelEvent
                             (TargetID *sender, unsigned long *msgRefcon,
                              Ptr msgBuff, unsigned long *msgLen);
pascal Boolean GetSpecificHighLevelEvent
                             (GetSpecificFilterProcPtr aFilter,
                              void *yourDataPtr, OSErr *err);
pascal void FlushEvents      (short whichMask, short stopMask);
pascal void SystemClick      (const EventRecord *theEvent,
                              WindowPtr theWindow);
pascal void SystemTask       (void);
pascal Boolean SystemEvent   (const EventRecord *theEvent);
pascal Boolean GetOSEvent    (short mask, EventRecord *theEvent);
pascal Boolean OSEventAvail  (short mask, EventRecord *theEvent);
pascal void SetEventMask     (short theMask);
#define GetEvQHdr()          ((QHdrPtr) 0x014A)
```

Sending Events

```
pascal OSErr PostHighLevelEvent
                             (const EventRecord *theEvent,
                              unsigned long receiverID,
                              unsigned long msgRefcon, Ptr msgBuff,
                              unsigned long msgLen,
                              unsigned long postingOptions);
```

```
pascal OSErr PPostEvent     (short eventCode, long eventMsg, EvQElPtr *qEl)
pascal OSErr PostEvent      (short eventNum, long eventMsg);
```

Converting Process Serial Numbers and Port Names

```
pascal OSErr GetPortNameFromProcessSerialNumber
                            (PPCPortPtr portName,
                             const ProcessSerialNumberPtr pPSN);
pascal OSErr GetProcessSerialNumberFromPortName
                            (const PPCPortPtr portName,
                             ProcessSerialNumberPtr pPSN);
```

Reading the Mouse

```
pascal void GetMouse        (Point *mouseLoc);
pascal Boolean Button       (void);
pascal Boolean StillDown    (void);
pascal Boolean WaitMouseUp  (void);
```

Reading the Keyboard

```
pascal void GetKeys         (KeyMap theKeys);
{the KeyTranslate function is also available as the KeyTrans function}
pascal long KeyTranslate    (const void *transData, short keycode,
                             long *state);
```

Getting Timing Information

```
pascal unsigned long TickCount
                            (void);
#define GetDblTime()        (* (unsigned long*) 0x02F0)
#define GetCaretTime()      (* (unsigned long*) 0x02F4)
```

Application-Defined Routine

Filter Function for Searching the High-Level Event Queue

```
pascal Boolean MyFilter     (void *yourDataPtr,
                             HighLevelEventMsgPtr msgBuff,
                             const TargetID *sender);
```

Assembly-Language Summary

Data Structures

Event Data Structure

0	`what`	word	event code
2	`message`	long	event message
6	`when`	long	ticks since startup
10	`where`	long	mouse location
14	`modifiers`	word	modifier flags

Target ID Data Structure

0	`sessionID`	long	session reference number (not used if posting event)
4	`name`	68 bytes	port name (specified in a `PPCPortRec` data structure)
72	`location`	34 bytes	location name (specified in a `LocationNameRec`)
106	`recvrName`	68 bytes	reserved

High-Level Event Message Data Structure

0	`HighLevelEventMsgHeaderLength`	word	reserved
2	`version`	word	reserved
4	`reserved1`	long	reserved
8	`theMsgEvent`	16 bytes	event record
22	`userRefCon`	long	reference constant
26	`postingOptions`	long	reserved
30	`msgLength`	long	reserved

Event Queue Header Data Structure

0	`qLink`	long	next queue entry
4	`qType`	word	queue type
6	`evtQWhat`	word	event code
8	`evtQMessage`	long	event message
12	`evtQWhen`	long	ticks since startup
16	`evtQWhere`	long	mouse location
20	`evtQModifiers`	word	modifier flags

Trap Macros

Trap Macros Requiring Routine Selectors
`_OSDispatch`

Selector	Routine
$0033	`AcceptHighLevelEvent`
$0034	`PostHighLevelEvent`

Selector	Routine
$0035	`GetProcessSerialNumberFromPortName`
$0045	`GetSpecificHighLevelEvent`
$0046	`GetPortNameFromProcessSerialNumber`

Trap Macros Requiring Register Setup

Trap macro name	Registers on entry	Registers on exit
`_FlushEvents`	D0: event mask (low-order word) stop mask (high-order word)	D0: 0 if all events were removed from the queue, or the event code of the event that stopped the search (low-order word)
`_GetOSEvent`	A0: address of event record D0: event mask (low-order word)	D0: 0 if `GetOSEvent` returns any event other than a null event, or –1 if it returns a null event (low-order byte)
`_OSEventAvail`	A0: address of event record D0: event mask (low-order word)	D0: 0 if `OSEventAvail` returns any event other than a null event, or –1 if it returns a null event (low-order byte)
`_PostEvent`	A0: event code (low-order word) D0: event message (long word)	D0: result code (low-order word)

Global Variables

`CaretTime`	The suggested difference in ticks that should exist between blinks of the caret in editable text.
`DoubleTime`	The suggested maximum difference in ticks that should exist between the time of a mouse-up event and a mouse-down event for your application to consider those two mouse events a double click.
`EventQueue`	The header of the event queue.
`KeyRepThresh`	The value of the auto-key rate (the amount of time, in ticks, that must elapse before the Event Manager generates a subsequent auto-key event).
`KeyThresh`	The value of the auto-key threshold (the amount of time, in ticks, that must elapse before the Event Manager generates an auto-key event).
`ScrDmpEnable`	A byte that, if set to 0, disables the Event Manager’s processing of Command–Shift–number key combinations with numbers 3 through 9.
`SEvtEnb`	A byte that, if set to 0, causes the `SystemEvent` function to always return `FALSE`.
`SysEvtMask`	The system event mask of the current application.
`Ticks`	A long integer that indicates the current number of ticks since the system last started up.

Result Codes

`noErr`	0	No error
`procNotFound`	–600	No eligible process with specified process serial number
`bufferIsSmall`	–607	Buffer is too small
`noOutstandingHLE`	–608	No outstanding high-level event
`connectionInvalid`	–609	Connection is invalid
`noUserInteractionAllowed`	–610	Cannot interact directly with user
`noPortErr`	–903	Invalid port name

C H A P T E R 3

Menu Manager

Contents

You can use the Menu Manager to create and manage the menus in your application. Menus allow the user to view or choose from a list of choices and commands that your application provides.

All Macintosh applications should provide these standard menus: the Apple menu, the File menu, and the Edit menu. If you include an Apple menu as a menu of your application, the Menu Manager automatically adds the Help and Application menus to your application's menu bar; it adds the Keyboard menu if more than one keyboard layout or input method is installed.

Menus are typically stored as resources. This chapter describes the menu-related resources. See the chapter "Introduction to the Macintosh Toolbox" in this book for general information on resources and see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on Resource Manager routines. See *Macintosh Human Interface Guidelines* for additional examples of menus that incorporate many principles of user interface design. *Inside Macintosh: Text* contains further information on localizing your application for worldwide markets.

You can choose to provide help balloons for your application's menus. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for additional details on providing help balloons for your application's menus.

You often present a dialog box to the user as a result of the user's choice of a menu command that requires additional information before you can perform the command. See the chapter "Dialog Manager" later in this book for information on creating dialog boxes in your application.

For additional information on processing events, see the chapter "Event Manager" earlier in this book.

This chapter provides an introduction to menus and the menu bar, and it then describes

- various types of menus your application can use
- standard menus
- how to store menus as resources
- how to create menus
- how to create a menu bar
- how to change characteristics of menu items
- how to add items to a menu

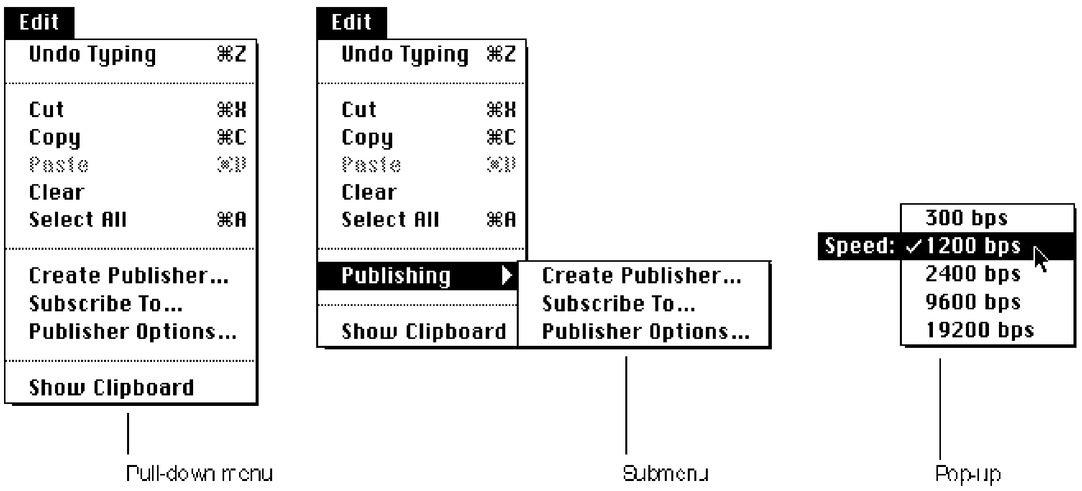
Introduction to Menus

A **menu** is a user interface element you can use in your application to allow the user to view or choose an item from a list of choices and commands that your application provides. Menus can appear in several different forms: pull-down menus, hierarchical menus, and pop-up menus.

A **pull-down menu** is identified by a menu title (a word or an icon) in the menu bar. Your application can use pull-down menus in the menu bar to allow users to choose a command or perform an action on a selected object. A **pop-up menu** is a menu that does not appear in the menu bar, but appears elsewhere on the screen when the user presses the mouse button while the cursor is in a particular place. Pop-up menus are most often accessed from a dialog box. Your application can use pop-up menus to let the user select one choice from a list of many or to set a specific value. A **submenu** refers to a menu that is attached to another menu. A menu to which a submenu is attached is referred to as a **hierarchical menu.**

Figure 3-1 shows examples of a pull-down menu, a submenu, and a pop-up menu.

Figure 3-1 A pull-down menu, a submenu, and a pop-up menu



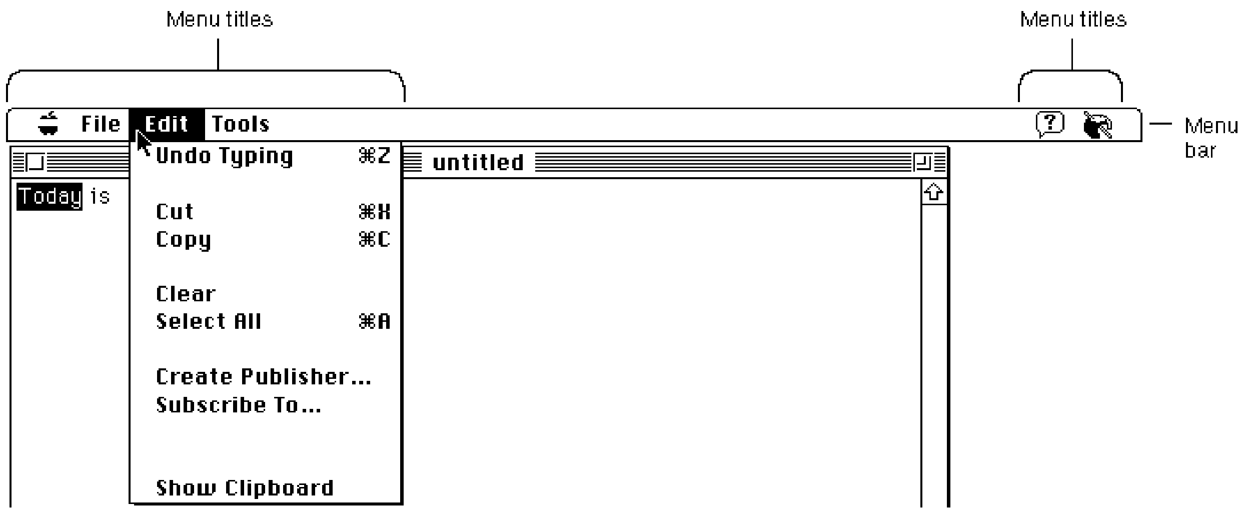
The standard **menu bar** extends across the top of the startup screen and contains the title of each available pull-down menu. Your application's menu bar should always provide at least the Apple menu, the File menu, and the Edit menu. When you insert the Apple menu in your application's menu bar, the Menu Manager automatically adds the Help and Application menus to your application's menu bar. It also adds the Keyboard menu if multiple script systems are installed or if a certain bit is set in the `'itlc'` resource. Your application can include as many other menus as fit on the smallest screen on which your application runs, and you should create only as many items as are essential to your application.

If your application uses a menu bar, you should make it always visible and available for use. If you do not always wish to display the menu bar (for example, if your application allows the user to view a screen presentation), you can give the user the option of viewing the presentation on the entire screen without the menu bar showing. However, you must provide a way, such as a keyboard equivalent for a command, for the user to access the menu bar or to make the menu bar reappear.

Using menus in your application allows the user to explore many possible choices and options without having to choose any particular one. By providing help balloons for your menus, you further allow users to learn about the possible actions or consequences of a particular menu choice without having to choose the menu command to find out what happens.

Figure 3-2 shows the SurfWriter application's menu bar with the Edit menu displayed. This application supports the standard Apple, File, and Edit menus; the Help and Application menus; and in addition supports two other application-specific menus.

Figure 3-2 The SurfWriter application's menu bar with the Edit menu displayed



Each menu has a menu title and one or more menu items associated with it. You should name each menu so that the title describes or relates to the actions the user can perform from that menu. For example, the Edit menu of a typical application contains commands that let the user edit the contents of a document.

Your application can disable any menu. The Menu Manager indicates that a menu is disabled by dimming its menu title. (In Figure 3-2, the Colors menu is disabled.) The Menu Manager dims all menu items of a disabled menu. The user can still pull down and examine the items in a disabled menu, but cannot choose any of the items.

Your application can also disable individual menu items. The Menu Manager dims the appearance of a disabled item and does not highlight it when the user rests the cursor on that item. If the user releases the mouse button while the cursor is over a disabled menu item, the Menu Manager reports that the user did not choose a menu command. (You can determine if this happened, however, by using the `MenuChoice` function.)

In Figure 3-2, the Paste command is disabled; the SurfWriter application disables the Paste command if the Clipboard is empty. SurfWriter also disables the Publisher Options command when the current selection does not contain a publisher or a subscriber. As explained in the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox*, your application should provide help balloons for disabled items that describe what the item normally does and explain why the item is not available at this time.

Note
Although `enabled` and `disabled` are the constants you use in a resource file to display or to dim menus and menu items, you shouldn't use these terms in your help balloons or user documentation. Instead use the terms *menus*, *menu commands*, and *menu items* for those that are enabled, and use the terms *not available* and *dimmed* to distinguish those that have been disabled. u

The Menu Manager highlights an enabled menu item when the cursor is over it. Enabled items do not have a dim appearance and can be chosen by the user.

Your application specifies whether menu items are enabled or disabled when it first defines and creates a menu. You can also disable or enable menu items at any time after creating a menu. You should enable a menu item whenever your application allows the user to choose the action associated with that item, and you should disable an item whenever the user cannot choose that item. For example, if the user selects text and then presses the mouse button while the cursor is in the menu bar, you should enable the Copy command in the Edit menu. You should disable the Copy command in the Edit menu if the user has not selected anything to copy.

Your application can also specify other characteristics of menu items, such as whether the item has a marking character next to its text (for example, a checkmark) or whether the item has a keyboard equivalent (for example, Command-C for the Copy command). "Menu Items" beginning on page 3-12 describes the characteristics of individual menu items in more detail.

The user typically chooses commands by moving the cursor to the menu bar and pressing the mouse button while the cursor is over a menu title. When the user presses the mouse button while the cursor is in the menu bar, your application should call the `MenuSelect` function. The `MenuSelect` function tracks the mouse, displays and removes menus as the user drags the cursor through the menu bar, highlights menu titles as the user drags the cursor over them, displays the menu items associated with a selected menu, highlights enabled menu items as the user drags through a menu, and handles all user activity until the user releases the mouse button.

The user chooses a menu item by releasing the mouse button while the cursor is over a particular enabled menu item. When the user chooses a menu item, the Menu Manager briefly blinks the chosen menu item (to confirm the choice) and then removes the menu from the display. The Menu Manager leaves the title of the chosen menu highlighted to provide feedback to the user.

The `MenuSelect` function returns information that allows your application to determine which menu item was chosen. Your application then typically responds by performing the desired command. When your application completes the requested action, your application should unhighlight the menu title, indicating to the user that the action is complete.

The user can move the cursor out of the menu (or menu bar) at any time; the Menu Manager displays any currently visible menu as long as the mouse button is pressed. (If the cursor is outside of the menu, the Menu Manager removes any highlighting of the menu item.) If the user releases the mouse button outside of a menu, the `MenuSelect` function reports that the user did not choose a menu item, and the Menu Manager removes any currently visible menu. Your application should not take any action if the user does not choose a menu item.

Menu and Menu Bar Definition Routines

The menu definition procedure and menu bar definition function define the general appearance and behavior of menus. The Menu Manager uses these routines to display and perform basic operations on menus and the menu bar.

A **menu definition procedure** performs all the drawing of menu items within a menu. When you define a menu, you specify its menu definition procedure. The Menu Manager uses the specified menu definition procedure to draw the menu items in a menu, determine which item the user chose from a menu, insert scrolling indicators as items in a menu, and calculate the menu's dimensions.

A **menu bar definition function** draws the menu bar and performs most of the drawing activities related to the display of menus when the user moves the cursor between them. Unless you specify otherwise, the Menu Manager uses the standard menu bar definition function to manage your application's menu bar. The Menu Manager uses the standard menu bar definition function to draw the menu bar, clear the menu bar, determine whether the cursor is in the menu bar or any currently displayed menu, calculate the left edges of menu titles, highlight a menu title, invert the entire menu bar, erase the background color of a menu and draw the menu's structure (shadow), and save or restore the bits behind a menu.

Apple provides a standard menu definition procedure and standard menu bar definition function. These definition routines are stored as resources in the System file. The standard menu definition procedure is the `'MDEF'` resource with resource ID 0. The standard menu bar definition function is the `'MBDF'` resource with resource ID 0.

When you define your menus and menu bar, you specify the definition routines that the Menu Manager should use when managing them. You'll usually want to use the standard definition routines for your application. However, if you need a feature not provided by the standard menu definition procedure (for example, if you want to include more graphics in your menus), you can choose to write your own menu definition procedure. See "Writing Your Own Menu Definition Procedure" beginning on page 3-87 for more information. While the Menu Manager does allow you to specify your own menu bar definition function, Apple recommends that you use the standard menu bar definition function.

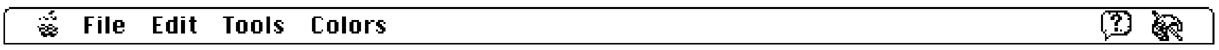
The Menu Bar

Each application has its own menu bar. The menu bar of an application applies to only that application. You usually define a menu bar for your application by providing a **menu bar** (`'MBAR'`) **resource** that lists the order and resource ID of each menu that appears in your menu bar. You define the menu title and the individual characteristics of menu items that appear in a menu by providing a **menu** (`'MENU'`) **resource** for each menu that appears in your menu bar. You use Menu Manager routines to create the menus and menu bar based on these resource definitions.

Your application can change the enabled state of a menu, add menus to or remove menus from its menu bar, or change the characteristics of any menu items. Whenever your application changes the enabled state of a menu or the number of menus in its menu bar, your application must call the `DrawMenuBar` procedure to update the menu bar's appearance.

The menu bar (as defined by the standard menu bar definition function) is white, with a height that is tall enough to display menu titles in the height of the system font and system font size, and with a black lower border that is one pixel tall. The menu bar is as wide as the screen and always appears on the monitor designated by the user as the startup screen. (The user selects a startup screen using the Monitors control panel.) The menu bar appears at the top of the screen, and nothing except the cursor can appear in front of it. Figure 3-3 shows the menu bar of the SurfWriter application.

Figure 3-3 The menu bar of the SurfWriter application



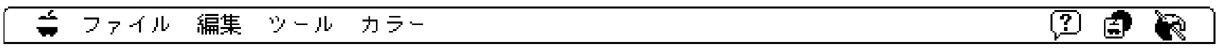
The menu bar helps to indicate the active application. The active application is the one whose menu bar is currently showing and whose icon appears as the menu title of the Application menu.

The titles of menus appear in the menu bar. A menu title is a text string (except for the Apple, Help, Keyboard, and Application menus, the titles of which contain a small icon). Menu titles always appear in the system font and system font size (for Roman scripts, the system font is Chicago and the system font size is 12).

You can insert any number of menu titles in the menu bar; however, less than 10 is usually optimum. Keep in mind that not all users have the same size monitor. Design your menu bar so that all titles can fit in the menu bar of the smallest screen on which your application can run. You should also consider localization issues when designing the number of menus that fit in your menu bar—not all menu titles might fit in the menu bar once the menu titles are translated. For example, English text often grows 50 percent larger when translated to other languages.

Figure 3-4 shows the SurfWriter application's menu bar with menu titles that have been localized for another script system.

Figure 3-4 The SurfWriter application's menu bar localized for another script system



Menus

A menu (as defined by the standard menu definition procedure) is a list of menu items arranged vertically and contained in a rectangle. The rectangle is shaded and can extend vertically for the length of the screen. If a menu has more items than will fit on the screen, the standard menu definition procedure adds a downward-pointing triangular indicator to the last item on the screen, and it automatically scrolls through the additional items when the user moves the cursor past the last menu item currently showing on the screen. When the user begins to scroll through the menu, the standard menu definition procedure adds an upward-pointing triangular indicator to the top item on the screen to indicate that the user can scroll the menu back to its original position.

Each menu can have color information associated with it. If you do not define the colors for your menus in your application's menu color information table, the Menu Manager uses the default colors for your menus and menu bar. The default colors are black text on a white background. In most cases the default colors should meet the needs of your application. "The Menu Color Information Table Record" on page 3-98 and "The Menu Color Information Table Resource" on page 3-155 give information on how you can define colors for your application's menus.

Your application's menus can contain any number of menu items. "Menu Items" (the next section) describes the visual variations that you can use when defining your menu items.

You typically define the order and resource IDs of the menus in your application's menu bar in an `'MBAR'` resource. You should define your `'MBAR'` resource such that the Apple menu is the first menu in the menu bar. You should define the next two menus as the File and Edit menus, followed by any other menus that your application uses. You do not need to define the Keyboard, Help, or Application menus in your `'MBAR'` resource; the Menu Manager automatically adds them to your application's menu bar if your application calls the `GetNewMBar` function and your menu bar includes an Apple menu or if your application inserts the Apple menu into the current menu list using the `InsertMenu` procedure.

You define the menu title and characteristics of each individual menu item in a `'MENU'` resource. "Creating a Menu Resource" on page 3-43 describes the `'MENU'` resource in more detail.

Pop-up menus do not appear in the menu bar but appear elsewhere on the screen. You often use pop-up menus in a dialog box when you want the user to be able to make a selection from a large list of choices. For example, rather than displaying the choices as a number of radio buttons, you can use a pop-up menu to display the choices at the user's convenience.

A hierarchical menu refers to either a pull-down or pop-up menu that has a submenu attached to it. (However, you should avoid attaching a submenu to a pop-up menu whenever possible, as this can make the interface more complex and less intuitive to the user.)

"Creating a Pop-Up Menu" on page 3-56 gives additional information about pop-up menus, and "Creating a Hierarchical Menu" on page 3-53 describes hierarchical menus in more detail.

Menu Items

A **menu item** can contain text or can be a line (a **divider**) separating groups of choices. A divider is always dimmed, and it has no other characteristics associated with it.

Each menu item (other than dividers) can have a number of visual characteristics:

- An icon to the left of the menu item's text. If you define an icon for a menu item, use an icon that gives a symbolic representation of the menu item's meaning or effect. You can specify an icon, a small icon, a reduced icon, or a color icon as the icon for a menu item; however, items with small or reduced icons cannot have submenus and cannot be drawn in a script other than the current system script.
- A checkmark or other marking character to the left of the menu item's text (and to the left of the item's icon, if any). Use such a mark if you need to denote the status of the menu item or the mode it controls. A menu item can have a mark or a submenu, but not both.
- The symbol for the Command key (⌘) and another 1-byte character to the right of the menu item's text (referred to as the *keyboard equivalent* of a command). Use this if your application allows the user to invoke the menu command from the keyboard by pressing the Command key and one or more other keys in combination, just as if the user had chosen the command from the menu. An item that has a keyboard equivalent cannot have a submenu, a small icon, or a reduced icon and cannot be drawn in a script other than the current system script.
- A triangular indicator to the right of the menu item's text to indicate that the item has a submenu. A menu item that has a submenu cannot have a keyboard equivalent, a marking character, a small icon, or a reduced icon and cannot be drawn in a script other than the current system script.
- A font style—either plain or one of various other styles—for the menu item's text. You can set the menu item's style to bold, italic, underline, outline, shadow, or any combination of these.
- The text of the menu item. Choose words for menu items that declare the action that occurs when the user chooses the command (usually verbs, such as Print or Save). You can also use adjectives if the command changes the attribute of a selected object (for example, Bold or Italic). Unless you specify otherwise, the text of menu items appears in the script of the system font and system font size (for Roman scripts, the system font is Chicago and the system font size is 12 points). If you want a menu item's text to appear in a script other than the current system script, you can specify a script code for the text. The Menu Manager draws the item's text in the script identified by the script code if the script for the specified script system is installed. A menu item that is drawn in another script cannot have a submenu, small icon, or reduced icon.
- Three ellipsis points (...) as the last character in the text of the menu item. Use ellipses in the text of menu items to indicate that your application displays a dialog box that requests more information from the user before executing the command. Do not use

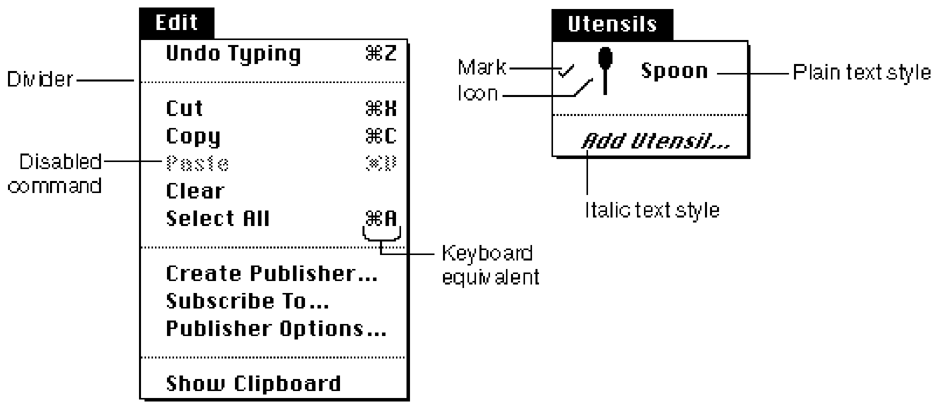
ellipses in menu items that display informational dialog boxes that do not require additional information from the user. In addition, you should not use ellipses if your application displays a confirmation alert after the user chooses a menu command. For example, if the user makes changes to a document, then chooses the Close command, your application can display a confirmation alert box, asking the user whether the document should be saved before closing. This type of command should not contain ellipses in its text.

If your application displays a dialog box requesting more information in response to the choice of a menu command, do include ellipses in the menu item's text. For example, the Open command includes ellipses in its text because the user must provide additional information: the name of the file to open. When you request more information from the user in a dialog box, you should provide an OK button or its equivalent in the dialog box that the user can select to perform the command. The dialog box should also include a Cancel button or its equivalent so that the user can cancel the command. See the chapter "Dialog Manager" in this book for information on creating dialog boxes.

- A dimmed appearance. When your application disables a menu item, the Menu Manager dims the menu item to indicate that the user can't choose it. Note that the Menu Manager dims the entire menu item, including any mark or icon, the menu text, and any keyboard equivalent symbol. Divider lines always have a dimmed appearance, regardless of whether your application enables them or not. When your application disables an entire menu, the Menu Manager dims the menu title and all menu items in that menu.

Figure 3-5 shows two menus with menu items that illustrate many of the characteristics that you can use when defining your menu items.

Figure 3-5 Two menus with various characteristics



When the primary line direction is right to left (as is the case for non-Roman script systems such as Arabic) the Menu Manager reverses the order of elements in menu items. For example, any marking character appears to the far right and any keyboard equivalent appears to the far left of the menu item's text.

On a monitor that is set to display only black and white, the Menu Manager displays dividers as dotted lines. In all other cases, the Menu Manager displays dividers as appropriate, based on the current color table. For example, on a monitor set to display 4-bit color or greater, the Menu Manager typically displays dividers as gray lines.

Your menu can contain as many menu items as you wish. However, only the first 31 menu items can be individually disabled (all menu items past 31 are always enabled if the menu is enabled and always disabled if the menu is disabled). If your menu items exceed the length of the screen, the user must scroll to view the additional items. Keep in mind that the fewer the menu items in a menu, the simpler and clearer the menu is for the user.

Groups of Menu Items

The menu items in a particular menu should be logically related to the title of the menu and grouped to provide greater ease of use and understanding to the user. You should separate groups with dividers.

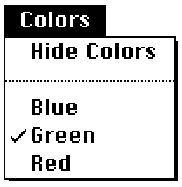
A menu can contain both commands that perform actions and commands that set attributes. You should use a verb or verb phrase to name commands that perform actions (for example, Cut, Copy, Paste). You should use an adjective to name commands that set attributes of a selected object (for example, Bold, Italic, Underline). You should group menu items by their type: verbs (actions) or adjectives (attributes). Create groups within each type according to the guidelines described here.

Group action commands that are logically related but independent; this makes your menus easier to read. For example, the Cut, Copy, Paste, Clear, and Select All commands in the Edit menu are grouped together; the Create Publisher, Subscribe To, and Publisher Options commands are grouped together; and the Show Clipboard command is set off by itself. (Figure 3-5 on page 3-13 shows these commands in the Edit menu of a typical application.)

Group attribute commands that are interdependent. You typically group a set of commands that set attributes into either a mutually exclusive group or an accumulating group.

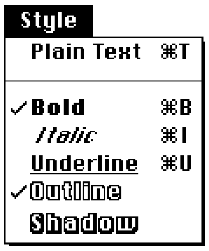
Group a set of attribute commands together if only one attribute in the group can be in effect at any one time (a *mutually exclusive* group). Place a checkmark next to the item that is currently in effect. If the user chooses a different attribute in the group, move the checkmark to the newly chosen attribute. For example, Figure 3-6 shows a Colors menu from the SurfWriter application. The colors listed in the Colors menu form a mutually exclusive group because only one color can be in effect at any one time. In this example, green is the color currently in effect. If the user chooses a different color, such as blue, the SurfWriter application uses the `SetItemMark` procedure to remove the checkmark from the Green command and to place a checkmark next to the Blue command.

Figure 3-6 Menu items in a mutually exclusive group



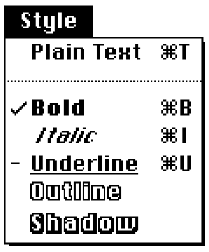
You can also group a set of attribute commands together if a number of the attributes in the group can be in effect at any one time (an *accumulating* group). In an accumulating group, use checkmarks to indicate that multiple attributes are in effect. In this type of group, you also need to provide a command that cancels all the other attributes. For example, a Style menu that lets the user choose any combination of font styles should also include a Plain Text command that cancels all the other attributes. Figure 3-7 shows a Style menu; in this example, the Bold and Outline attributes are both in effect.

Figure 3-7 Menu items in an accumulating group



You can also use a combination of checkmarks and dashes to help indicate the state of the user’s content. For example, in a menu that reflects the state of a selection, place a checkmark next to an item if the attribute applies to the entire selection; place a dash next to an item if the attribute applies to only part of the selection. Figure 3-8 shows a Style menu that indicates that the selection contains more than one style. In this figure, the Bold attribute applies to the entire selection; the Underline attribute applies to only part of the selection.

Figure 3-8 Use of a checkmark and dash in an accumulating group



Your application should adjust its menus appropriately before displaying its menus. For example, you should add checkmarks or dashes to items that are attributes as necessary, based on the state of the user's document and according to the type of window that is in the front. See "Adjusting the Menus of an Application" on page 3-73 for more information.

Another way to show the presence or absence of an attribute is to use a toggled command. Use a toggled command if the attribute has two states and you want to allow the user to move between the two states using a single menu command. For example, your application could provide a Show Borders command when the borders surrounding publishers and subscribers are not showing in a document. When the user chooses the Show Borders command, your application should show the borders and change the menu item to Hide Borders. When the user chooses the Hide Borders command, your application should hide the borders surrounding any publishers or subscribers and change the menu item to Show Borders. Use a toggled command only when the wording of the two versions of the command is not confusing to the user. Choose a verb phrase as the text of a toggled command; the text should clearly indicate the action your application performs when the user chooses the item. See "Changing the Text of an Item" on page 3-59 for further information on providing a toggled command.

Keyboard Equivalents for Menu Commands

A menu command can have a keyboard equivalent. The term **keyboard equivalent** refers to a keyboard combination, such as Command-C (⌘-C) or any other combination of the Command key, another key, and one or more modifier keys, that invokes a corresponding menu command when pressed by the user. For example, if your application supports the New command in the File menu, your application should perform the same action when the user presses Command-N as when the user chooses New from the File menu.

The term **Command-key equivalent** refers specifically to a keyboard equivalent that the user invokes by holding down the Command key and pressing another key (other than a modifier key) at the same time. This generates a keyboard event that specifies a 1-byte character that your application should pass as a parameter to the `MenuKey` function. The `MenuKey` function maps the given 1-byte character to the menu item (if any) with that Command-key equivalent.

The Menu Manager provides support for Command-key equivalents. If you define a Command-key equivalent for a menu item, the standard menu definition procedure draws the Command symbol and the specified 1-byte character to the right of the menu item's text (or to the left of the item's text if the primary line direction is right to left).

You detect a Command-key equivalent of a command by examining the `modifiers` field of the event record for a keyboard event. This allows you to determine whether the Command key was pressed at the same time as the keyboard event. If so, your application typically calls the `MenuKey` function, passing as a parameter the character code that represents the key pressed by the user. The `MenuKey` function determines if the 1-byte character matches any of the keyboard equivalents defined for your menu items; if so, `MenuKey` returns this information to your application. Your application can then perform the associated menu command, if any. See the chapter “Event Manager” in this book for additional information about the `modifiers` field of the event record.

The keyboard layout (`'KCHR'`) resource of some keyboards masks or cancels the effect of the Shift key when the Command key is also pressed. For example, with a U.S. keyboard layout, when a user presses Command-S, the character code in the message field of the event record is $73 (the character code for “s”); when a user presses Command-Shift-S, the character code in the message field of the event record is also $73. However, not all `'KCHR'` resources mask the Shift key in this way.

Furthermore, when your application uses the `MenuKey` function to process Command-key equivalents, `MenuKey` does not distinguish between uppercase and lowercase letters. The `MenuKey` function takes the 1-byte character passed to it and calls the `UpperText` procedure (which provides localizable uppercase conversion of the character). Thus, `MenuKey` translates any lowercase character to uppercase when comparing a keyboard event to keyboard equivalents. If your application must distinguish between lowercase and uppercase characters for keyboard equivalents, you need to provide your own method for handling such keyboard equivalents.

The key you specify for a Command-key equivalent must be a 1-byte character and is usually a letter (although you can specify 1-byte characters other than letters). For consistency and to provide greater support for localizing your application, you should always specify any letters for keyboard equivalents in uppercase when you define your application’s menu commands.

If you wish to provide other types of keyboard equivalents in addition to Command-key equivalents, your application must take additional steps to support them. If your application allows the user to hold down more than one modifier key to invoke a keyboard equivalent, your application must provide in the menu item a visual indication that represents this keyboard combination. In most cases your application must use its own method (other than `MenuKey`) for mapping the keyboard equivalent to the corresponding menu item.

If you specify a key other than a letter for a Command-key equivalent or use more than one modifier key for a keyboard equivalent, you should choose keys and keyboard combinations that can be easily localized for other regions.

If your application uses other keyboard equivalents, you can examine the state of the modifier keys and use the `KeyTranslate` function, if necessary, to help map the keyboard equivalent to a particular menu item. See the chapter “Event Manager” in this book for information on the `KeyTranslate` function, and see the discussion of `'KCHR'` resources in *Inside Macintosh: Text* for information on how various keyboard combinations map to specific character codes.

One command that isn’t listed in a menu but can be invoked from the keyboard is the Command-period (⌘-.) or Cancel command. You detect a Command-period command in a method similar to the method for detecting other keyboard equivalents—you examine the `modifiers` field of a keyboard event to determine whether the Command key was pressed. In this case, however, if the user pressed the period key in addition to the Command key, rather than invoking a menu command your application should cancel the current operation.

You typically define the Command-key equivalents for your application's menu commands when you define the menu commands in a `'MENU'` resource. The Menu Manager displays the Command-key equivalent for a menu command (if it has one) to the right of the menu item's text (or to the left of the item's text for right-to-left script systems).

Apple reserves several keyboard equivalents for common commands. You should use these keyboard equivalents for commands in the File and Edit menus of your application.

Table 3-1 show the keyboard equivalents for standard commands.

Table 3-1 Reserved keyboard equivalents for all systems

Keys	Command	Menu
⌘-A	Select All	Edit
⌘-C	Copy	Edit
⌘-N	New	File
⌘-O	Open…	File
⌘-P	Print…	File
⌘-Q	Quit	File
⌘-S	Save	File
⌘-V	Paste	Edit
⌘-W	Close	File
⌘-X	Cut	Edit
⌘-Z	Undo	Edit

Note

You should use the keyboard equivalents Z, X, C, and V for the editing commands Undo, Cut, Copy, and Paste in order to provide support for editing in desk accessories and dialog boxes. u

Apple also reserves several keyboard equivalents for use with worldwide versions of system software, localized keyboards, and keyboard layouts. Table 3-2 shows these keyboard equivalents. Your application should not use the keyboard equivalents listed in Table 3-2 for its own menu commands.

See *Inside Macintosh: Text* for more discussion of handling keyboard equivalents in other script systems.

The key combinations listed in Table 3-1 and Table 3-2 are reserved across all applications. Even if your application doesn't support one of these menu commands, it shouldn't use these keyboard equivalents for another command. This guideline is for the user's benefit. Reserving these key combinations provides guaranteed, predictable behavior across all applications.

Table 3-2 Reserved keyboard equivalents for worldwide systems

Keys	Action
⌘–Space bar	Rotate through enabled script systems
⌘–Option–Space bar	Rotate through keyboard layouts or input methods within the active script system
⌘–modifier key–Space bar	Reserved
⌘–Right arrow	Change keyboard layout to the current keyboard layout of the Roman script
⌘–Left arrow	Change keyboard layout to the current keyboard layout of the system script

Table 3-3 shows other common keyboard equivalents. These keyboard equivalents are secondary to the standard keyboard equivalents listed in Table 3-1 and Table 3-2. If your application doesn't support one of the functions in Table 3-3, then you can use the equivalent as you wish.

Table 3-3 Other common keyboard equivalents

Keys	Command	Menu
⌘-B	Bold	Style
⌘-F	Find	File
⌘-G	Find Again	File
⌘-I	Italic	Style
⌘-T	Plain Text	Style
⌘-U	Underline	Style

You shouldn't assign keyboard equivalents to infrequently used menu commands. Only add keyboard equivalents for the commands that your users employ most frequently.

Menus Added Automatically by the Menu Manager

In System 7, the Menu Manager may add as many as three additional menus to your application's menu bar: the Help menu, the Keyboard menu, and the Application menu. These menus provide access to system features such as Balloon Help, keyboard layouts, and application switching. All three of these menus have icons as titles and are positioned at the right side of the menu bar. (These menus are sometimes referred to as the *system-handled menus.*)

The Menu Manager automatically inserts these additional menus in your application's current menu list when your application inserts an Apple menu into its menu bar. In this case, the Menu Manager always displays the Application menu, displays the Help menu if space is available, and displays the Keyboard menu if multiple script systems are installed and space is available. The Menu Manager also displays the Keyboard menu if the `smfShowIcon` bit is set in the flags byte of the `'itlc'` resource.

The Help menu icon or both the Help menu icon and the Keyboard menu icon disappear from the menu bar if your application inserts a menu whose title extends into the space occupied by one or both of those icons. This allows your application to reclaim any space in the menu bar that would have been occupied by one or both of those two menu icons, if necessary. However, if your application inserts a menu whose title is long enough to overlap space occupied by the Application menu icon, the overlapping portion of that title disappears behind the Application menu icon. The Application menu icon is always displayed in the menu bar.

Because the Menu Manager inserts the Help, Keyboard, and Application menus into your application's current menu bar, you should not make any assumptions about the last menu (or menus) in your menu bar. Apple also reserves the right to add other system-handled menus to your application's menu bar; for compatibility you should define your menu bar such that there is room for the Help, Keyboard, and Application menus and at least one additional system-handled menu.

Your application does not need to take any action if the user chooses an item from the Keyboard or Application menu; the Menu Manager performs any appropriate actions for these two menus. If the user chooses an item that your application added to the Help menu, your application should perform the corresponding action.

The following sections describe the Help, Keyboard, and Application menus in more detail, and they also describe other menus in a typical application, including the Apple, File, and Edit menus.

The Apple Menu

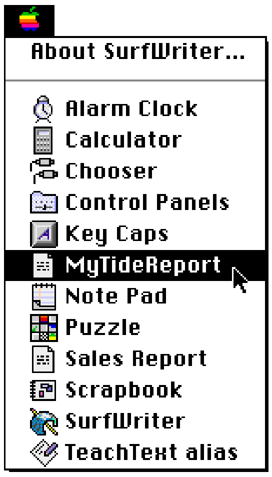
You should define the Apple menu as the first menu in your application. The title of the Apple menu is the Apple icon. The Apple menu of an application typically provides an About command as the first menu item, followed by a divider, which is followed by a list of all desktop objects contained in the Apple Menu Items folder. (The phrase *desktop objects* refers to applications, desk accessories, documents, folders, and any other item that can reside in the Apple Menu Items folder.) The items following the divider in the Apple menu are listed in alphabetical order. Each item below the divider lists a desktop object and the small icon for that object.

Figure 3-9 shows the Apple menu for the SurfWriter application as it might appear on a particular user's system.

To create the items in your application's Apple menu, define the Apple menu title, the characteristics of your application's About command, and the divider following it in a `'MENU'` resource.

To insert the items contained in the Apple Menu Items folder into your application's Apple menu, use the `AppendResMenu` or `InsertResMenu` procedure and specify `'DRVR'` as the resource type to add in the parameter `theType`. If you do this, these procedures automatically add all items in the Apple Menu Items folder in alphabetical order to the specified menu.

Figure 3-9 The Apple menu for the SurfWriter application



Note
The Apple Menu Items folder is available in System 7 and later. In System 6, the `AppendResMenu` and `InsertResMenu` procedures add only the desk accessories in the System file to the specified menu when you specify `'DRVR'` as the resource type to add in the parameter `theType`. u

The user can place any desktop object in the Apple Menu Items folder. When the user places an item in this folder, the system software automatically adds it to the list of items in the Apple menu of all open applications.

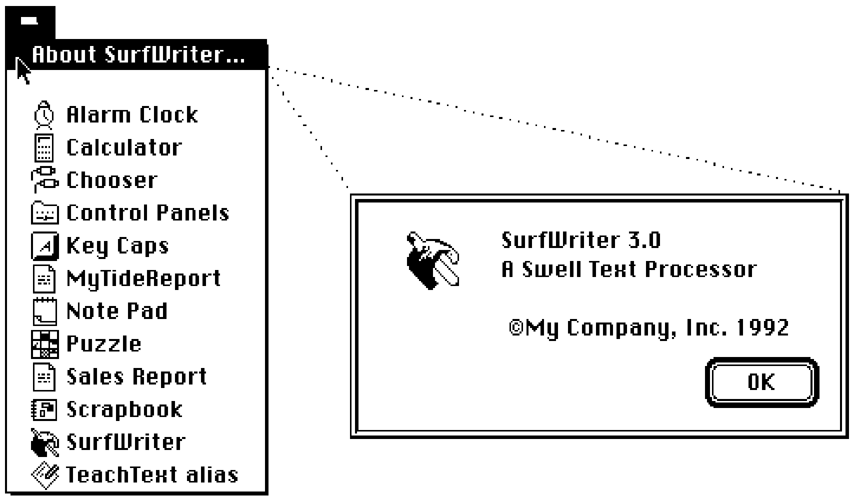
When the user chooses an item other than your application's About command from the Apple menu, your application should call the `OpenDeskAcc` function. The `OpenDeskAcc` function prepares to open the desktop object chosen by the user; for example, if the user chooses the Alarm Clock desk accessory, the `OpenDeskAcc` function prepares to open the Alarm Clock. The `OpenDeskAcc` function schedules the Alarm Clock desk accessory for execution and returns to your application. On your application's next call to `WaitNextEvent`, it receives a suspend event, and then the Alarm Clock desk accessory becomes the foreground process.

If the user chooses a desktop object other than a desk accessory or an application, the `OpenDeskAcc` function also takes the appropriate action. For example, as shown in Figure 3-9, if the user chooses a document called MyTideReport created by the SurfWriter application, the `OpenDeskAcc` function prepares to open the SurfWriter application (if it isn't already open) and schedules the SurfWriter application for execution. The SurfWriter application is instructed to open the MyTideReport document when it becomes the foreground process.

When the user chooses your application's About command, your application can display a dialog box or an alert box that contains your application's name, version number, copyright information, or other information as necessary. Your application should provide an OK button in the dialog box; the user clicks the OK button to close the dialog box.

Figure 3-10 shows the alert box that the SurfWriter application displays when the user chooses the About command from the application's Apple menu.

Figure 3-10 Choosing the About command of the SurfWriter application



If your application provides any application-specific Help commands, place these in the Help menu, not the Apple menu.

The File Menu

The standard File menu contains commands related to managing documents. For example, the user can open, close, save, or print documents from this menu. The user should also be able to quit your application by choosing Quit from the File menu.

Your application should support the menu commands of the standard File menu. If you add other commands to your application's File menu, they should pertain to managing a document.

Figure 3-11 shows the standard File menu for applications.

Figure 3-11 The standard File menu for an application

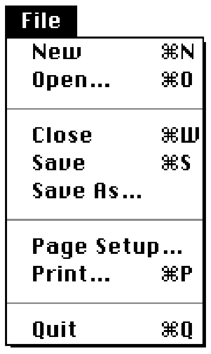


Table 3-4 describes the standard commands in the File menu and the actions your application should take when a user chooses them.

Table 3-4 Actions for standard File menu commands

Command	**Action**
New	Open a new, untitled document.
Open...	Display the Open dialog box using the Standard File Package.
Close	Close the active window (which may be a document window, modeless dialog box, or other type of window). If the active window is a document and the document has been changed since the last save, display a dialog box asking the user if the document should be saved before closing.
Save	Save the active document to a disk, including any changes made to that document since the last time it was saved. If the user chooses Save for a new untitled document (one that the user hasn't named yet), display the Save dialog box using the Standard File Package.
Save As...	Save a copy of the active document under a new name provided by the user. Display the Save dialog box using the Standard File Package. After your application saves the document, the document should remain open and active.
Page Setup...	Display the Page Setup dialog box to let the user specify printing parameters such as the paper size and printing orientation. Your application can provide other printing options as appropriate. Your application should save the user's Page Setup printing preferences for the document when the user saves the document.
Print...	Display the Print job dialog box to let the user specify various parameters, such as print quality and number of copies. Print the document if the user clicks the Print button. The options specified in the Print dialog box apply to only the current printing operation, and your application should not save these settings with the document or restore the settings when the user chooses Print again.
Quit	Quit your application after performing any necessary cleanup operations. If any open documents have been changed since the user last saved them, display the Save dialog box once for each open document that requires saving. If any background or lengthy operation is still in progress, notify the user, giving the user the option to continue and not quit the application.

See *Macintosh Human Interface Guidelines* for additional commands that you can provide in the File menu. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for information on how to perform the actions associated with the commands in the File menu. See the chapter "Standard File Package" in *Inside Macintosh: Files* for information on the standard file dialog boxes. See the chapter "Printing Manager" in *Inside Macintosh: Imaging* for information on displaying the Page Setup and Print job dialog boxes.

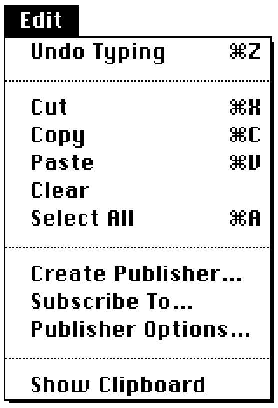
The New, Open, Close, Save, Print, and Quit commands have the keyboard equivalents shown in Figure 3-11 on page 3-22. These keyboard equivalents are reserved for these menu commands; do not assign these keyboard equivalents to any menu command other than the ones shown in Figure 3-11.

The Edit Menu

The standard Edit menu provides commands that let users change or edit the contents of their documents. It also provides commands that allow users to share data within and between documents created by different applications using editions or the Clipboard. All Macintosh applications should support the Undo, Cut, Copy, Paste, and Clear commands. Use these commands to provide standard text-editing abilities in your application.

Figure 3-12 shows the standard Edit menu supported by Macintosh applications.

Figure 3-12 The standard Edit menu for an application



The standard editing commands (Undo, Cut, Copy, Paste, and Clear) in your application's Edit menu should appear in the order shown in Figure 3-12. Whenever possible, you should add an additional word or phrase to clarify what action your application will reverse when the user chooses the Undo command. For example, Figure 3-12 shows an application's Edit menu that uses the phrase Undo Typing when typing was the last action performed by the user. If your application can't undo the last operation, you should change the text of the Undo command to Can't Undo and disable the menu item. See "Changing the Text of an Item" on page 3-59 for an example of how to change the text of a menu item.

You can include other commands in your application's Edit menu if they're related to editing or changing the content of your application's documents. If you add commands to the Edit menu, add them after the standard menu commands. For example, if appropriate, your application should support a Select All command. If your application supports both the Clear and Select All commands, they should appear in the order shown in Figure 3-12.

Table 3-5 describes the standard commands in the Edit menu and the actions your application should take when a user chooses them.

Table 3-5 Actions for standard Edit menu commands

Command	Action
Undo	Reverse the effect of the previous operation. You should add the name of the last operation to the Undo command. For example, change the item to read Undo Typing if the user just finished entering some text in a document. If your application cannot undo the previous operation, disable this menu item and change the phrase to Can't Undo.
Cut	Remove the data in the current selection, if any. Store the cut selection in the scrap (on the Clipboard). This replaces the previous contents of the scrap.
Copy	Copy the data in the current selection, if any. Copy the selection to the scrap (the Clipboard). This replaces the previous contents of the scrap.
Paste	Paste the data from the scrap at the insertion point; this replaces any current selection.
Clear	Remove the data in the highlighted selection; do not copy the data to the scrap (Clipboard).
Select All	Highlight all data in the document.
Create Publisher...	Display the Create Publisher dialog box (using the Edition Manager). Create an edition based on the selected data if the user clicks the Publish button.
Subscribe To...	Display the Subscribe To dialog box (using the Edition Manager). Allow the user to insert data from an edition if the user clicks the Subscribe button.
Publisher Options...	Display the Publisher Options dialog box (using the Edition Manager) and allow the user to set or change options associated with the publisher. Change this menu item to Subscriber Options if the current selection includes a subscriber. When the user chooses the Subscriber Options command, display the Subscriber Options dialog box.
Show Clipboard	Display the contents of the Clipboard in a window. Change this item to Hide Clipboard when the Clipboard window is showing. When the user chooses Hide Clipboard, hide the window displaying the Clipboard contents and change the menu item to Show Clipboard.

The Undo, Cut, Copy, Paste, and Select All commands have the keyboard equivalents shown in Figure 3-12 on page 3-24. These keyboard equivalents are reserved for these menu commands; do not assign these keyboard equivalents to any menu command other than the ones shown in Figure 3-12. See the chapter "Scrap Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on copying data to and from the scrap.

See the chapter "Edition Manager" in *Inside Macintosh: Interapplication Communication* for information on supporting the Create Publisher, Subscribe To, and Publisher Options commands in your application.

The Font Menu

You can provide a Font menu to allow the user to choose text fonts. A font is a complete set of characters created in one typeface and font style. The characters in a font can appear in many different point sizes, but all have the same design elements.

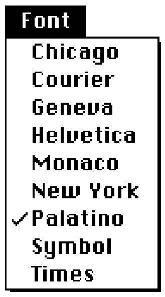
You should list the names of all currently available fonts in your application's Font menu. The currently available fonts are those fonts residing in the Fonts folder of the user's System Folder (or in earlier versions of system software, in the user's System file).

You add fonts to the Font menu using the `AppendResMenu` or `InsertResMenu` procedure. These two procedures add items to the specified menu in alphabetical order.

The user can install a large number of fonts and thereby create a very large Font menu. Therefore, you should never include other items in the Font menu. Use separate menus to accommodate lists of attributes such as style and size choices. You can also provide a Size menu to allow the user to choose a specific point size of a font; the next section describes the Size menu.

Figure 3-13 shows a typical Font menu. Your application should indicate which typeface is in use by adding a checkmark to the left of the name of the current font. In Figure 3-13, the application has placed a checkmark next to Palatino to indicate that Palatino® is the current font. When the user starts entering text at the insertion point, your application should display text in the current font.

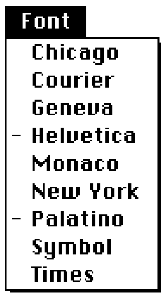
Figure 3-13 A typical Font menu



In the Font menu, you can use dashes to indicate that the selection contains more than one font. (Place a checkmark next to an item if the entire selection contains only one font.) If the current selection contains more than one font, place a dash next to the name of each font that the selection contains. See "Changing the Mark of Menu Items" on page 3-61 for information on adding dashes and checkmarks to a menu item.

Figure 3-14 shows the use of dashes to indicate that a selection contains more than one font. In this figure, part of the selection contains a Helvetica® font and part of the selection contains a Palatino font.

Figure 3-14 A Font menu showing a selection containing more than one font



The `AppendResMenu` and `InsertResMenu` procedures can recognize when an added font resource is associated with a script other than the current system script (non-Roman fonts have font numbers greater than $4000). The Menu Manager displays a font name in its corresponding script if the script system for that font is installed.

You can choose to provide a Size menu and a Style menu in addition to a Font menu. If you do so, these three menus typically appear in the order Font, Size, Style in most applications.

The Size Menu

Your application can provide a Size menu to allow the user to choose sizes for fonts. Font sizes are measured in points. A point is a typographical unit of measure equivalent (on Macintosh computers) to 1/72 of an inch.

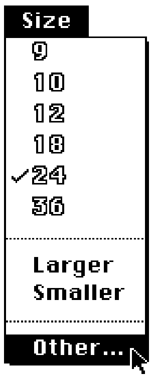
Your application should indicate the current point size by adding a checkmark to the menu item of the current size. You can use dashes if the selection contains more than one point size.

System 7 supports both bitmapped and TrueType fonts. TrueType fonts can be displayed in a wider range of point sizes, for example, 12 points, 51 points, 156 points, 578 points, or greater. Your application should not provide an upper limit for font sizes.

In the Size menu, your application should outline font sizes to indicate which sizes are directly provided by the current font. If the user chooses a TrueType font, outline all sizes of that font in the Size menu. If the user chooses a bitmapped font, outline only those sizes that appear in the Fonts folder. Use plain type for all other font sizes. See the chapter “Font Manager” in *Inside Macintosh: Text* for additional information on supporting fonts in your application.

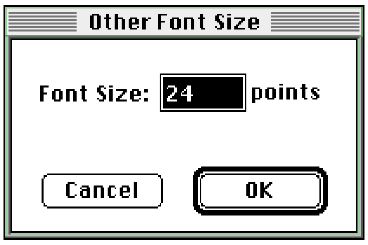
Figure 3-15 shows a typical Size menu of an application.

Figure 3-15 A typical Size menu



Your application should also provide a method that allows users to choose any point size. You can add an Other command to the end of the Size menu for this purpose. When the user chooses this command, display a dialog box that allows the user to choose any available font size. You can include an editable text item in which the user can type the desired font size. Figure 3-16 shows a dialog box an application might display when the user chooses the Other command from the Size menu.

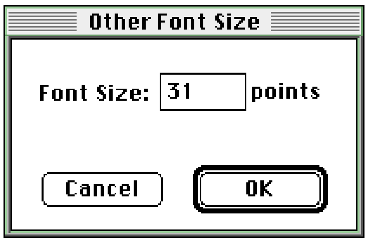
Figure 3-16 A dialog box to select a new point size for a font


Other Font Size

Font Size: 24 points

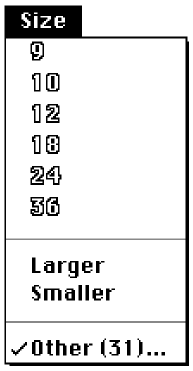
Figure 3-17 shows the Other dialog box after the user has entered a new font size of 31.

Figure 3-17 Entering a new point size for a font



If the user enters a font size not currently in the menu, your application should add a checkmark to the Other menu command and include the font size as part of the text of the Other command. You should show the font size in parentheses after the text Other, as shown in Figure 3-18.

Figure 3-18 The Other command with a font size added to it



If a selection contains more than one nonstandard size, you should include the text Mixed in parentheses following the word Other. In this case leave the editable text field of the Other dialog box blank when the user chooses the Other (Mixed) command.

See “Handling a Size Menu” on page 3-82 for more information on how to respond to the user’s choice of a command from the Size menu. See the chapter “Dialog Manager” for information on creating a dialog box.

The Help Menu

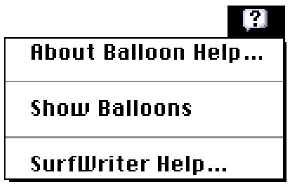
The Help menu is specific to each application, just as the Apple, File, and Edit menus are. The Help menu items defined by the Help Manager are common to all applications and give the user access to Balloon Help.

You can add menu items to your application’s Help menu to give your users access to any online help that your application supplies in addition to help balloons. If you currently provide your users with help information when they choose the About command from the Apple menu, you should instead append a command for your own help to the Help menu. This gives users one consistent place to obtain help information.

When adding your own items to the Help menu, include the name of your application in the command so that users can easily determine which application the help relates to.

Figure 3-19 shows the Help menu for the SurfWriter application. This application appends one item to the end of the standard Help menu: SurfWriter Help. When the user chooses this item, the application provides access to any application-specific help information.

Figure 3-19 The Help menu of the SurfWriter application



You add items to the Help menu by using the `HMGetHelpMenuHandle` function and the `AppendMenu` procedure. Apple reserves the right to change the number of standard items in the Help menu. You should always append any additional items to the end. See "Adding Items to the Help Menu" on page 3-67 for specific examples.

The user turns Balloon Help on or off by choosing Show Balloons or Hide Balloons from the Help menu. The Help Manager automatically enables or disables Balloon Help when the user chooses Show Balloons or Hide Balloons from the Help menu. The setting of help is global and affects all applications.

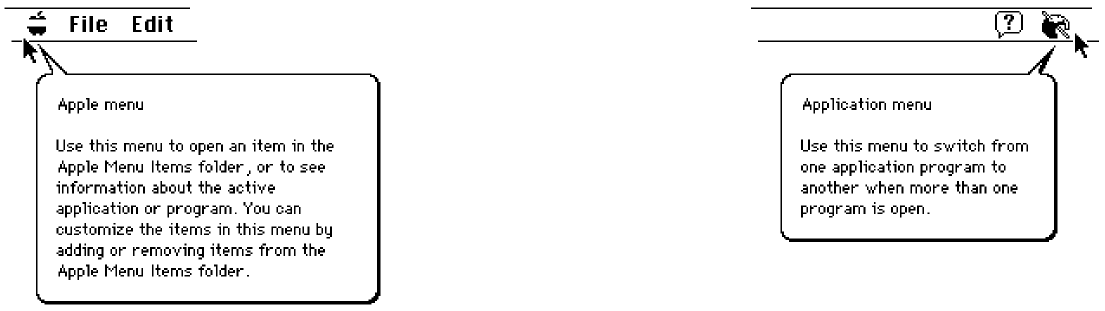
When the user turns on Balloon Help, the Help Manager displays small help balloons as the user moves the cursor over areas such as scroll bars, buttons, menus, or rectangular areas in windows or dialog boxes that have help information associated with them. **Help balloons** are rounded-rectangle windows that contain explanatory information for the user.

The Help Manager provides help balloons for the menu titles of the Apple, Help, Application, and Keyboard menus. The Help Manager also provides help balloons for menu items in the Application and Keyboard menus, for any item from the Apple Menu Items folder in the Apple menu, and for the standard items in the Help menu. The Help Manager provides these help balloons only if your application uses the standard menu definition procedure.

Your application should provide the content of help balloons for all other menu items and menus in your application.

Figure 3-20 shows the default help balloons for the Apple menu title and Application menu title.

Figure 3-20 Default help balloons for the Apple menu and Application menu



Apple menu

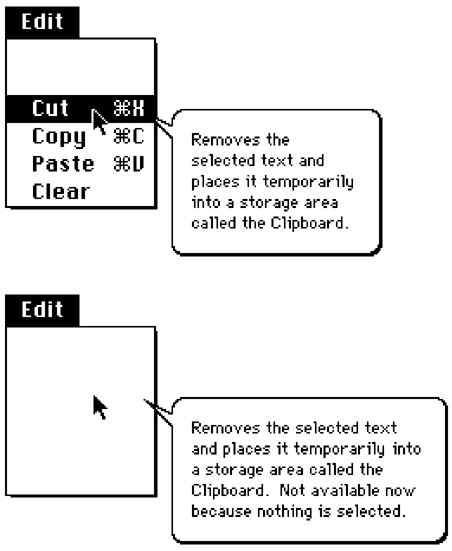
Use this menu to open an item in the Apple Menu Items folder, or to see information about the active application or program. You can customize the items in this menu by adding or removing items from the Apple Menu Items folder.

Application menu

Use this menu to switch from one application program to another when more than one program is open.

Figure 3-21 shows help balloons for an application's Cut command when it is enabled and when it is disabled.

Figure 3-21 Help balloons for different states of the Cut command



Removes the selected text and places it temporarily into a storage area called the Clipboard.

Removes the selected text and places it temporarily into a storage area called the Clipboard. Not available now because nothing is selected.

Your application can provide the content for help balloons for your menus and menu items. You define the help balloons for your application using `'hmmu'` resources.

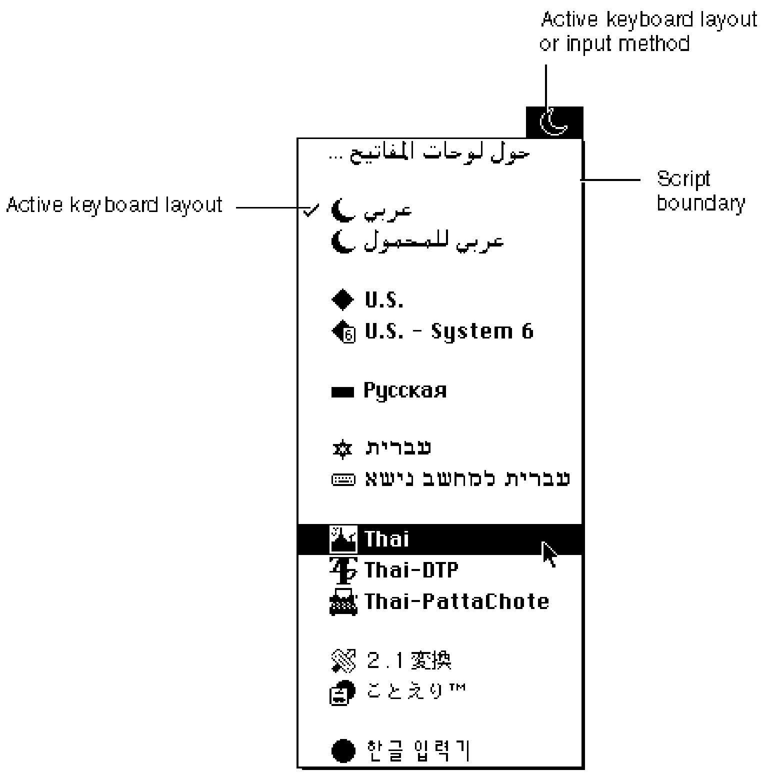
For information on how to define the help balloons for your application's menus in `'hmmu'` resources, see the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox*.

The Keyboard Menu

The Keyboard menu displays a list of all the keyboard layouts and input methods that are available for each enabled script system. Each script system has at least one keyboard layout or input method associated with it. If only the Roman script system and the U.S. keyboard layout are available, the Menu Manager does not add the Keyboard menu (unless the `smfShowIcon` bit is set in the flags byte of the `'itlc'` resource). If the user's system includes an additional script system or includes additional keyboard layouts for the Roman script system and the `smfShowIcon` bit is set in the `'itlc'` resource, the Menu Manager adds the Keyboard menu to your application's menu bar as long as your application's menu bar includes an Apple menu. The Menu Manager adds the Keyboard menu to the right of the Help menu and to the left of the Application menu.

Figure 3-22 shows a Keyboard menu as it might appear on a particular user's system. System software groups the items in the Keyboard menu by their script systems. For example, in Figure 3-22 seven script systems are shown: Arabic, Roman, Cyrillic, Hebrew, Thai, Japanese, and Korean. Two keyboard layouts are available in the user's system for the Arabic script system, two keyboard layouts for the Roman script system, one keyboard layout for the Cyrillic script system, two keyboard layouts for the Hebrew script system, three keyboard layouts for the Thai script system, two input methods for the Japanese script system, and one input method for the Korean script system.

Figure 3-22 Accessing the Keyboard menu from an application



When the user chooses an item from the Keyboard menu, the Menu Manager handles it appropriately. For example, if the user chooses a different keyboard layout in a different script, the Menu Manager changes the current keyboard layout and script system to the item chosen by the user. See *Inside Macintosh: Text* for further information on supporting text and handling text in multiple scripts in your application.

The Application Menu

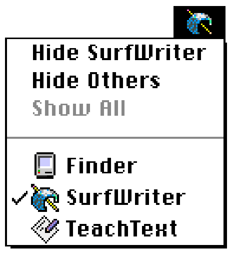
The Application menu is the menu farthest to the right in the menu bar; the Application menu contains the icon of the active application or desk accessory for its menu title.

The Menu Manager automatically appends the Application menu to your application's menu bar if your menu bar includes an Apple menu.

When the user chooses an item from the Application menu, the Menu Manager handles the event as appropriate. For example, if the user chooses the Hide Others command, the Menu Manager hides the windows of all other open applications. If the user chooses another application from the Application menu, the Menu Manager sends your application a suspend event. Your application receives the suspend event the next time it calls `WaitNextEvent`, and your application is switched out after handling the suspend event. (See the chapter "Event Manager" in this book for information about responding to suspend and resume events.)

Figure 3-23 shows the Application menu for the SurfWriter application as it appears when both SurfWriter and TeachText are open and the user is currently interacting with SurfWriter. The checkmark next to the menu item showing SurfWriter's icon indicates that SurfWriter is the active application.

Figure 3-23 SurfWriter's Application menu



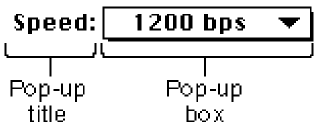
Pop-Up Menus

You can use pop-up menus to present the user with a list of choices in a dialog box or window. Pop-up menus are especially useful in dialog boxes that require the user to select one choice from a list of many or to set a specific value.

In System 7, the standard pop-up menu is implemented by a control definition function. This section explains how the standard pop-up control definition function provides support for pop-up menus. The chapter "Control Manager" in this book explains controls in detail.

A pop-up menu appears as a rectangle with a one-pixel border and a one-pixel drop shadow. Pop-up menus are identified by a downward-pointing triangle that appears in the pop-up box. The title of the pop-up menu appears next to the pop-up box. Figure 3-24 shows a pop-up menu.

Figure 3-24 A pop-up menu

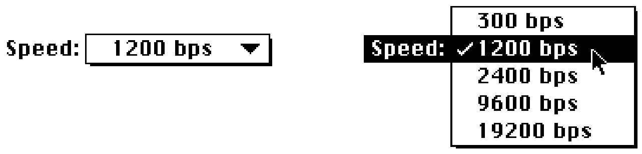


To display a pop-up menu, the user presses the mouse button while the cursor is over the pop-up title or pop-up box. If the pop-up menu is in a dialog box and your application uses the Dialog Manager, the Dialog Manager uses the pop-up control definition function to display the pop-up menu and to handle all user interaction in the pop-up menu. If the pop-up menu is in one of your application's windows, your application needs to determine which control the cursor was in when the user pressed the mouse button. Your application can then use the Control Manager routines to display the pop-up menu and to handle user interaction in the control.

Just like `MenuSelect`, the pop-up control definition function highlights the pop-up menu title and highlights menu items appropriately as the user drags the cursor through the menu items. The pop-up control definition function also highlights the default (current) menu item when the pop-up menu is first displayed and adds the checkmark to the menu item. Once the user releases the mouse button, the pop-up control definition function causes the chosen item (if any) to blink, unhighlights the menu title, changes the text in the pop-up box, and stores the item number of the chosen item as the value of the control. Your application can use the Control Manager function `GetControlValue` to get the menu item chosen by the user.

Figure 3-25 shows a pop-up menu in its closed state (as it appears initially to the user) and its open state (as it appears when the user presses the mouse button while the cursor is in the pop-up menu).

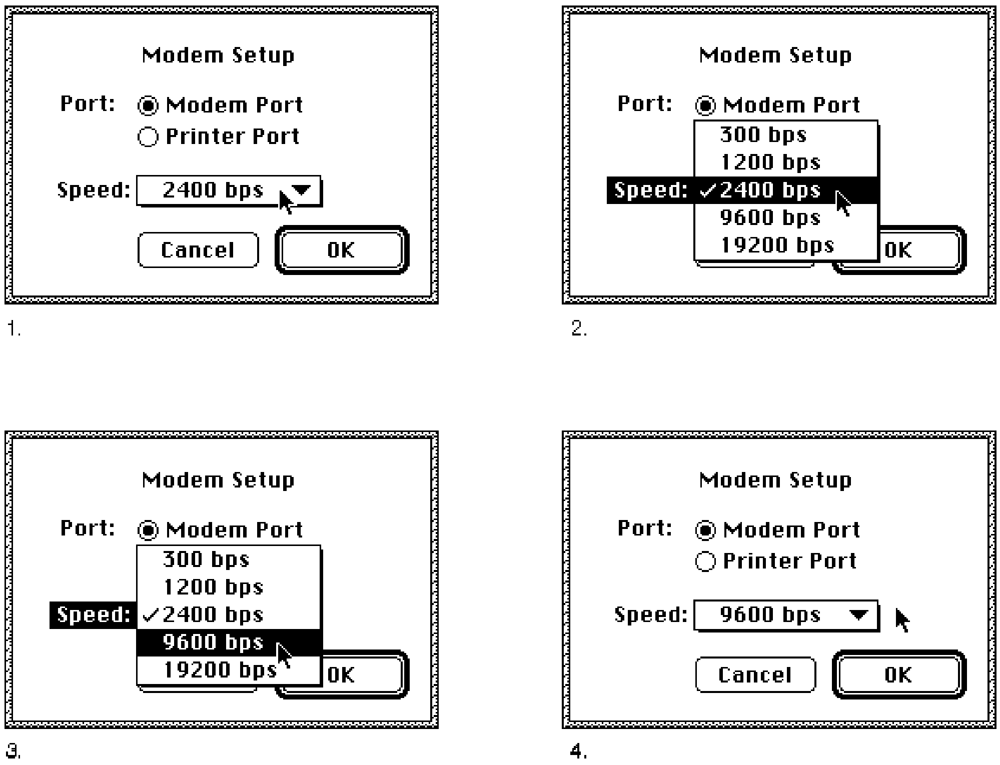
Figure 3-25 A pop-up menu in its closed and open states



If you don't provide a title for a pop-up menu, the current menu item serves as the title. In most cases you should create pop-up menus that have titles. Choose a title that reflects the contents of the menu or indicates the purpose of the menu.

Figure 3-26 shows the process of a user making a selection from a pop-up menu.

Figure 3-26 Making a selection from a pop-up menu

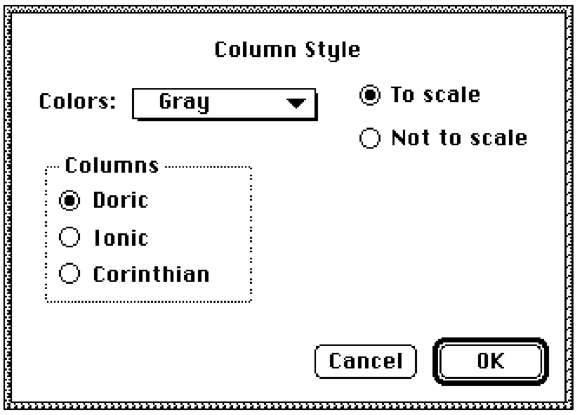


In step 1 in Figure 3-26, the user presses the mouse button while the cursor is over the pop-up box. When this occurs, your application can use the Dialog Manager or Control Manager to call the pop-up control definition function. In step 2, the pop-up control definition function highlights the title of the pop-up menu, removes the downward-pointing triangle from the pop-up box, adds a checkmark to the current item, highlights the current item, and displays the contents of the pop-up menu. In step 3, the pop-up control definition function handles all user interaction, highlighting and unhighlighting menu items, until the user releases the mouse button. When the user releases the mouse button, the pop-up control definition function closes the pop-up menu, unhighlights the pop-up menu title, sets the text of the pop-up box to the item chosen by the user, and stores the item number of the chosen item as the value of the control. Step 4 shows the appearance of the closed pop-up menu after the pop-up control definition function performs these actions.

If your application does not use the standard pop-up control definition function, you can create your own control definition function and you can choose to use the `PopUpMenuSelect` function to help your application handle pop-up menus. In this case, when the user presses the mouse button when the cursor is in a pop-up menu, your application should call the `PopUpMenuSelect` function. Your application must highlight the pop-up title before calling `PopUpMenuSelect` and unhighlight it afterward. The `PopUpMenuSelect` function displays the pop-up menu and highlights menu items appropriately as the user drags the cursor through the menu items. Once the user releases the mouse button, `PopUpMenuSelect` flashes the chosen item, if any, and returns information indicating which menu item was chosen to your application. Your application is responsible for highlighting and unhighlighting the menu title, updating the text in the pop-up box, and storing any changes to the settings of the menu items if you use the `PopUpMenuSelect` function.

Pop-up menus work well when your application needs to present several choices to the user. Note that pop-up menus hide these choices from the user until the user chooses to display the pop-up menu. Use pop-up menus when the user doesn't need to see all the choices all the time. For example, Figure 3-27 shows a dialog box that uses a pop-up menu to allow the user to choose one color from a list of many.

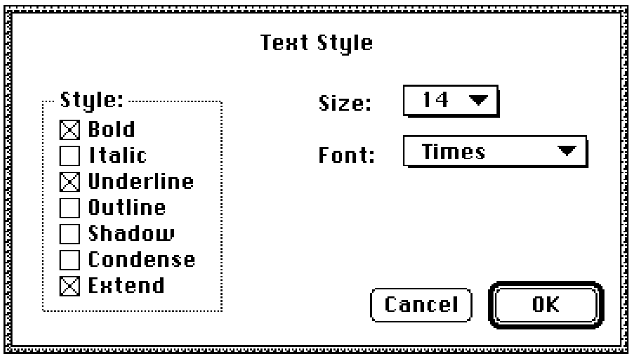
Figure 3-27 Choosing one attribute from a list of many



If you need to show only a few choices, you may find that using checkboxes or radio buttons is more appropriate for your application. For example, in Figure 3-27 the selection of columns is implemented with radio buttons rather than a pop-up menu. Whenever possible, you should show all available choices to the user. Note that in this example the amount of space occupied by the radio buttons is about the same as the amount of space required for a corresponding pop-up menu.

Use pop-up menus to allow the user to choose one option from a set of many choices. Don't use a pop-up menu for multiple-choice lists where the user can make more than one selection. If you do, the text in the menu box will not fully describe the selections in effect. For example, don't use a pop-up menu for font style selections. In a dialog box, font style selections are more appropriately implemented as checkboxes. Figure 3-28 shows a dialog box that uses checkboxes instead of a pop-up menu to allow the user to select more than one font style. The Size and Font choices are implemented as pop-up menus in this example, since the user can choose only one size and one font from a list of many.

Figure 3-28 A dialog box with checkboxes and pop-up menus


Text Style

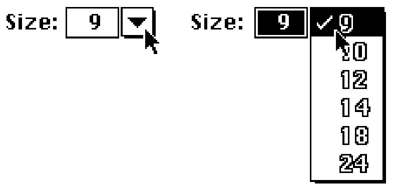
Style:
☒ Bold
☐ Italic
☒ Underline
☐ Outline
☐ Shadow
☐ Condense
☒ Extend

Size: 14
Font: Times

Never use a pop-up menu as a way to provide more commands. Pop-up menus should not contain actions (verbs) but can contain attributes (adjectives) or settings that allow the user to choose one from many. For these reasons, you should not use Command-key equivalents for pop-up menu items.

Your application can also use type-in pop-up menus when appropriate. Use a type-in pop-up menu to give the user a list of choices and to allow the user to type in an additional choice. The standard pop-up control definition function that implements pop-up menus does not provide specific support for type-in menus. You can create your own control definition function to handle type-in pop-up menus. If you do so, your type-in pop-up menu should adhere to the guidelines described here. Figure 3-29 shows a typical type-in pop-up menu in its closed and open states.

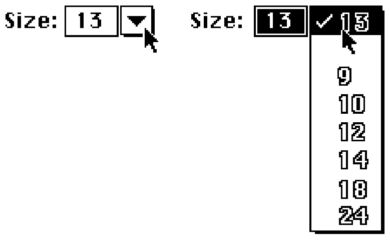
Figure 3-29 A type-in pop-up menu in its closed and open states



Your application is responsible for drawing and highlighting the type-in field of the pop-up menu. Your application does not need to highlight the title of a type-in pop-up menu; your application should highlight the type-in field instead.

If the user types in a value that is already in the menu, make that item the current item. If the user types a value that does not match any of the items in the pop-up menu, add the item to the top of the menu and add a divider below the item to separate it from the rest of the standard items. Figure 3-30 on the next page shows a type-in pop-up menu with a user's choice added to it.

Figure 3-30 A type-in pop-up menu with a user's choice added



A type-in pop-up menu should allow the user to type in a single additional choice. That is, a standard type-in pop-up menu does not accumulate the user's choices in the menu. For example, if the user types in a value of 13, then types in a new choice, such as 43, the menu should appear as shown in Figure 3-30, except that the type-in field and menu item that previously contained 13 is replaced by 43.

A type-in pop-up menu should also allow the user to type in any of the standard values in the menu or choose any of the standard items in the pop-up menu. If the user types in or chooses any of the standard items, you should remove any user-specified item previously added to the menu. For example, as shown in Figure 3-30, the user specified a nonstandard size of 13. If the user then types in or selects 9, your application should return the pop-up menu to its standard state, as shown in Figure 3-29 on page 3-37.

Hierarchical Menus

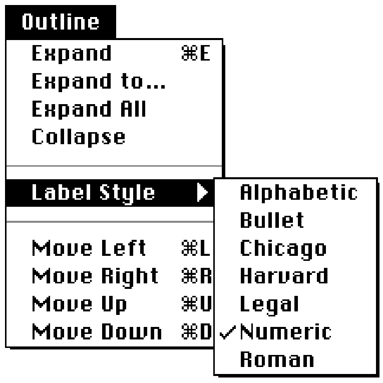
A hierarchical menu is a menu that has a submenu attached to it. Hierarchical menus can be useful when your application needs to offer additional choices to the user without taking up extra space in the menu bar. If you use a hierarchical menu in your application, use it to give the user additional choices or to choose attributes, not to choose additional commands.

In a hierarchical menu, a menu item serves as the title of a submenu; this menu item contains a triangle to identify that the item has a submenu. The triangle appears in the location of the keyboard equivalent. The title of a submenu should represent the choices it contains. Figure 3-31 shows a menu with a submenu whose menu title is Label Style.

When a user drags the cursor through a hierarchical menu and rests the cursor on a menu item that has a submenu, the Menu Manager displays the submenu after a brief delay. The title of the submenu remains highlighted while the user browses through the submenu; the Menu Manager unhighlights the menu title of the submenu when the user releases the mouse button.

Hierarchical menus are useful for providing lists of related items, such as font sizes and font styles. Never use more than one level of hierarchical menus (in other words, don't attach a submenu to another submenu). You can assign keyboard equivalents to the menu items of a submenu; however, if you do so, you make it harder for the user to quickly scan all menus for their keyboard equivalents.

Figure 3-31 A hierarchical menu item and its submenu



About the Menu Manager

The Menu Manager, together with the menu definition procedure and menu bar definition function, provides your application with a convenient way to manage the menus in your application. The Menu Manager uses two data structures, menu records and menu lists, to manage menus. The next two sections describe how the Menu Manager uses these two data structures. “Using the Menu Manager,” which begins on page 3-41, shows how you can use the Menu Manager to

- define a menu using a `'MENU'` resource
- define a menu bar using an `'MBAR'` resource
- install your application’s menu bar
- change the appearance of menu items
- add menu items to a menu
- respond to the user when the user chooses a menu item
- handle the Apple and Help menus
- create a pop-up menu
- create a hierarchical menu
- handle access to menus when your application displays a dialog box
- write your own menu definition procedure

How the Menu Manager Maintains Information About Menus

The Menu Manager maintains information about menus in menu records. Each menu record includes certain information about a specific menu, including

- the menu ID of the menu
- the horizontal and vertical dimensions of the menu (in pixels)
- a handle to the menu definition procedure of the menu
- flags indicating whether each item (for the first 31 items) is enabled or disabled and whether the menu title is enabled or disabled
- the contents of the menu, including the menu title and other data that defines the menu items

You typically specify most of this information in a menu resource, that is, a resource of type `'MENU'`. When you create a menu, the Menu Manager stores this information in a menu record. A **menu record** is a data structure of type `MenuInfo`. You usually never need to access the information in the menu record directly; the Menu Manager automatically updates the menu record when you make any changes to the menu, such as adding a menu item. See “The Menu Record” beginning on page 3-95 if you need to access the fields of the menu record directly.

The Menu Manager identifies every menu by a number referred to as a **menu ID.** You must assign a menu ID to each menu in your application. Each menu in your application must have a menu ID that is unique from that of any other menu in your application. You can use any number greater than 0 for a menu ID of a pull-down or pop-up menu; submenus of an application can use only menu IDs from 1 through 235; submenus of a desk accessory must use menu IDs from 236 through 255.

When you create a menu, the Menu Manager creates a menu record for the menu and returns a handle to that menu record. To refer to a menu, you usually use either the menu’s menu ID or a handle to the menu’s menu record.

To refer to a menu item, use the menu item’s **item number.** Item numbers identify items in menus; items are assigned item numbers starting with 1 for the first menu item in the menu, 2 for the second menu item in the menu, and so on, up to the number of the last menu item in the menu.

How the Menu Manager Maintains Information About an Application’s Menu Bar

A **menu list** contains handles to the menu records of one or more menus (although a menu list can be empty). The end of a menu list can contain handles to the menu records of submenus and pop-up menus; the phrase *submenu portion of the menu list* refers to this portion of the menu list, which contains information about submenus and pop-up menus.

When your application initializes the Menu Manager, the Menu Manager allocates the current menu list, which is initially empty. The contents of the current menu list change as your application adds menus to or removes menus from it.

The **current menu list** contains handles to the menu records of all menus in the current menu bar and the menu records of any submenus or pop-up menus that you have inserted into the current menu list. Your application typically creates a menu list using `GetNewMBar`, and it then sets the current menu list to its newly created menu list using `SetMenuBar`. You can insert other menus in the current menu list using the `GetMenu` function and `InsertMenu` procedure.

The Menu Manager displays the menu bar and the titles of all pull-down menus that are defined in the current menu list when your application calls the `DrawMenuBar` procedure. The Menu Manager displays the menus in the menu bar in the same order that they appear in the current menu list.

The Menu Manager provides routines for adding menus to and removing menus from the current menu list; your application should never access a menu list directly. To refer to a menu list, use the handle returned by `GetNewMBar` or `GetMenuBar`.

The Menu Manager inserts the Help menu, the Keyboard menu if necessary, and the Application menu into your application's menu list if your application calls the `GetNewMBar` function and your menu bar includes an Apple menu; your application then uses `SetMenuBar` to set the current menu list to the newly created menu list. The Menu Manager also inserts these menus into your application's current menu list if your application inserts the Apple menu into the current menu list using the `InsertMenu` procedure. Therefore, you should not make any assumptions about the last menu (or menus) in your application's current menu list.

When your application inserts a submenu into the current menu list, the Menu Manager stores a handle to the menu record of the submenu in the submenu portion of the current menu list. Similarly, when your application inserts a pop-up menu into the current menu list, the Menu Manager stores a handle to the menu record of the pop-up menu in the submenu portion of the current menu list.

Using the Menu Manager

You can define your application's menus and menu bar as resources and use Menu Manager routines to create and manage them. For example, whenever the user presses the mouse button while the cursor is in the menu bar, your application should call the `MenuSelect` function, allowing the user to choose a command from any menu. The `MenuSelect` function handles all user activity until the user releases the mouse button. The `MenuSelect` function displays and removes menus as the user drags the cursor through the menu bar, and it highlights enabled menu items as the user drags through a menu.

You should provide help balloons for each menu title and menu item of your application. You store information and text for help balloons in resources. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for complete and specific information on how to provide help balloons for the menus of your application. The BalloonWriter application, available from APDA, can also help you create help balloons for the menus of your application.

Your application needs to initialize QuickDraw, the Font Manager, and the Window Manager before using the Menu Manager. Your application can accomplish this using the `InitGraf`, `InitFonts`, and `InitWindows` procedures. To initialize the Menu Manager, use the `InitMenus` procedure.

If your application uses pop-up menus, use the `Gestalt` function with the `gestaltPopUpAttr` selector to determine if the control definition function for pop-up menus is available. See *Inside Macintosh: Operating System Utilities* for information about the `Gestalt` function.

To create the pull-down menus in your application's menu bar, you need to

- create descriptions of each pull-down menu in `'MENU'` resources
- create an `'MBAR'` resource that lists the order and resource ID of each menu
- use the `GetNewMBar` function and `SetMenuBar` procedure to set up your menu bar and use the `DrawMenuBar` procedure to draw your menu bar

The next section, "Creating a Menu," explains these steps in detail.

After creating your application's menu bar, you can enable or disable your menu items, add marks such as checkmarks or dashes to menu items, or add items to any of your menus as needed. See "Enabling and Disabling Menu Items" on page 3-58, "Changing the Mark of Menu Items" on page 3-61, and "Adding Items to a Menu" beginning on page 3-64 for information on these topics.

"Handling User Choice of a Menu Command," beginning on page 3-70, shows how to handle mouse-down events in the menu bar, adjust the menus of your application, and determine if the user chose a keyboard equivalent of a command.

"Responding When the User Chooses a Menu Item," beginning on page 3-78, describes how your application should respond once the user chooses an item and also shows how to handle the user's choice of a command from the Apple and Help menus.

If your application displays dialog boxes, see "Accessing Menus From a Dialog Box" beginning on page 3-84.

Finally, if your application needs to create submenus or pop-up menus, see "Creating a Hierarchical Menu" on page 3-53 and "Creating a Pop-Up Menu" on page 3-56.

Creating a Menu

You use various Menu Manager routines to set up the menus and the menu bar for your application. You can use any of these methods to create pull-down menus for your application:

- You can create descriptions of your application's menus in `'MENU'` resources and describe your application's menu bar in an `'MBAR'` resource. You use the `GetNewMBar` function to read in descriptions of your menu bar and menus and create a new menu list, use the `SetMenuBar` procedure to set the current menu list to your application's menu list, and use the `DrawMenuBar` procedure to update the menu bar.
- You can create descriptions of your application's menus in `'MENU'` resources, read them in using `GetMenu`, add them to the current menu list using `InsertMenu`, and update the menu bar using `DrawMenuBar`.

- You can use NewMenu to create new empty menus; use AppendMenu, InsertMenuItem, InsertResMenu, or AppendResMenu to fill the menus with menu items; add the menus to the current menu list using InsertMenu; and update the menu bar using DrawMenuBar.

Whenever possible you should define your menus in menu ('MENU') resources and your menu bar in a menu bar ('MBAR') resource to make your application easier to localize.

To create a hierarchical menu, you need to create descriptions of the submenu and the menu to which the submenu is attached. Usually you create the description of both menus in 'MENU' resources. You typically read in the description of the hierarchical menu using GetNewMBar (if you also provide an 'MBAR' resource). To read in the description of the submenu and insert it in the current menu list, use the GetMenu function and InsertMenu procedure.

To create a pop-up menu, create descriptions of the pop-up menu and its menu items, create a control that uses the pop-up control definition function, and associate the control with a window or dialog box. You can display and manage the pop-up menu using the Dialog Manager or Control Manager routines.

Once the Menu Manager creates a menu for your application, if necessary you can add additional menu items to the menu using AppendMenu, InsertMenuItem, InsertResMenu, or AppendResMenu. You can use various Menu Manager routines to change the appearance of menu items.

The next sections describe how to create 'MENU' and 'MBAR' resources. "Creating a Hierarchical Menu" on page 3-53 describes how to create a menu that has a submenu, and "Creating a Pop-Up Menu" on page 3-56 describes how to create pop-up menus.

Creating a Menu Resource

Usually you should define your menus in menu ('MENU') resources so that you can easily localize the menu titles and menu items for other languages, cultures, or regions. A 'MENU' resource defines the menu title of a menu and the characteristics of menu items in a menu. Listing 3-1 shows a sample 'MENU' resource in Rez format for an application's Apple menu. (Rez is a resource compiler available with MPW. You can also define menus using a resource utility, such as ResEdit, available from APDA.)

Listing 3-1 Rez input for a 'MENU' resource for the Apple menu

```
#define mApple 128

resource 'MENU' (mApple, preload) { /*resource ID, preload resource*/
      mApple,                       /*menu ID*/
      textMenuProc,                 /*uses standard menu definition */
                                    /* procedure*/
      0b1111111111111111111111111111101,  /*enable About item, */
                                          /* disable divider, */
                                          /* enable all other items*/
      enabled,                      /*enable menu title*/
       apple,                       /*menu title*/
      {
                                    /*first menu item*/
            "About SurfWriter…",       /*text of menu item */
                                       /* (includes ellipsis)*/
                                       /*item characteristics follow*/
                  noicon,              /*icon number (if any) or */
                                       /* script code (if any)*/
                  nokey,               /*keyboard equivalent (if any) */
                                       /* or submenu (if any) or */
                                       /* small or reduced icon (if any)*/
                  nomark,              /*marking character (if any) or */
                                       /* menu ID of submenu (if any)*/
                  plain;               /*style of menu item text*/
                                    /*second menu item*/
            "-",                       /*item text (divider)*/
                  noicon, nokey, nomark, plain  /*item characteristics*/
      }
};
```

You should also define help balloons for each of your application's menu items and each menu title when you create your menus. (Figure 3-21 on page 3-31 shows help balloons for an application's Cut command.) You define the help balloons for your application's menus in `'hmmu'` resources. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for examples of how to create `'hmmu'` resources.

Listing 3-1 defines the resource ID of the Apple menu as 128. You can use any number equal to or greater than 128 as a resource ID for a menu. By convention, many applications use 128 as the resource ID of the first menu in the application's menu bar (the Apple menu) and use sequential numbers for the resource IDs of following menus.

Listing 3-1 also defines the menu ID of the Apple menu as 128. Once your application creates the menu, the Menu Manager uses the defined menu ID to refer to this menu. The number you define for the menu ID of a menu does not have to match the resource ID of the menu, but it is usually more convenient to use the same number. You can use any number greater than 0 for the menu ID of a pull-down or pop-up menu; submenus of an application can only use menu IDs from 1 through 235; submenus of a desk accessory must use menu IDs from 236 through 255.

The listing specifies that this menu uses the standard menu definition procedure. If you choose to create your own menu definition procedure, list its resource ID instead of the `textMenuProc` constant.

After the resource ID of the menu definition procedure is a 32-bit number (expressed as a 31-bit field followed by a `Boolean` field), where bits 1–31 indicate if the corresponding menu item is disabled or enabled, and bit 0 indicates whether the menu is enabled or disabled.

The listing specifies in the 31-bit field that the first menu item should be enabled, that the second menu item should be disabled, and that the following menu items (item numbers 3 through 31) should be enabled when the menu is first created. After creating a menu, your application can enable or disable menu items using the `EnableItem` or `DisableItem` procedure. If a menu contains more than 31 items, the Menu Manager automatically enables all items following the 31st item when the menu is enabled. Your application cannot disable any individual items following the 31st item. However, you can disable all items, including items after the 31st item, by disabling the entire menu.

Listing 3-1 specifies that the menu title should be enabled when it is first created. Your application can also disable or enable the menu title using the `DisableItem` or `EnableItem` procedure. When you disable a menu using the `DisableItem` procedure, the Menu Manager disables all menu items in the menu (including any items following the 31st item) and dims the title of the menu.

The resource listing identifies the title of the menu using the constant `apple`. If you specify the `apple` constant as the title, the Menu Manager uses a small Apple icon as the title of the menu. The Menu Manager uses a color Apple icon if the monitor is set to display colors. The listing then defines the characteristics of each menu item in the menu. For each menu item, you need to define the text and any other characteristics of the menu item. For example, Listing 3-1 defines the first item in the Apple menu as the About command; note that the text of this menu item specifies three ellipsis points (...). Specify three ellipsis points following the text of a menu command if your application displays a dialog box requesting information from the user before performing the command. In general, you should not use ellipses if your application displays a confirmation alert after the user chooses a menu command; the About command is an exception to this guideline.

Listing 3-1 defines other characteristics of the About command—it doesn't have an icon to the left of the menu item text, it doesn't have a keyboard equivalent, it doesn't have any mark to the left of the menu item text, and the font style of the menu item text is plain.

By specifying various combinations of values in the icon field and keyboard equivalent field, you can define an icon (normal, small, reduced, or color), a keyboard equivalent, a submenu, or the script code of a menu item. Note that some characteristics are mutually exclusive (for example, an item can have a keyboard equivalent or submenu, but not both), as described in the following paragraphs. Table 3-6 on page 3-46 summarizes how the Menu Manager interprets these item characteristics.

Table 3-6 Specifying submenus, script codes, reduced icons, small icons, and color icons of a menu item in a menu resource

Keyboard equivalent field	Icon field	Marking character field	Description
$1B		Menu ID of submenu	Indicates the item has a submenu. The marking character field specifies the menu ID of the submenu.
$1C	Script code of item text		Indicates the item text uses the script defined by the script code specified in the icon field.
$1D	Icon number of `'ICON'` resource		Indicates the item has an icon defined by an `'ICON'` resource and that it should be reduced to fit in a 16-by-16 bit rectangle.
$1E	Icon number of `'SICN'` resource		Indicates the item has an icon defined by an `'SICN'` resource.
$00 or >$20	Icon number of `'ICON'` or `'cicn'` resource		Indicates the item has an icon defined by an `'ICON'` or a `'cicn'` resource. (A value greater than $20 in the keyboard equivalent field specifies the item's keyboard equivalent.)

To assign an icon to a menu item, specify an icon number in place of the `noicon` constant. The icon number you specify should be a number from 1 through 255 (or from 1 through 254 for small icons and reduced icons); add 256 to your icon number and use the result for the resource ID of the color icon (`'cicn'`) resource, icon (`'ICON'`) resource, or small icon (`'SICN'`) resource that describes the icons for the menu item. You must define the icon for a menu item in a `'cicn'`, an `'ICON'`, or an `'SICN'` resource; the Menu Manager uses only these types of resources for icons you define for your menu items. The Menu Manager first looks for a `'cicn'` resource with the calculated resource ID and uses that icon if it finds it. If it doesn't find a `'cicn'` resource (or if the computer doesn't have Color QuickDraw) and the keyboard equivalent field specifies $1E, the Menu Manager looks for an `'SICN'` resource with the calculated resource ID. Otherwise, the Menu Manager looks for an `'ICON'` resource and plots it in a 32-by-32 bit rectangle, unless the keyboard equivalent field contains $1D. If the keyboard equivalent field contains $1D, the Menu Manager reduces the icon to fit in a 16-by-16 bit rectangle.

If you provide an `'ICON'` resource and specify the `nokey` constant or a value greater than $20 as the keyboard equivalent, the Menu Manager enlarges the rectangle of the entire menu item to fit the 32-by-32 bit `'ICON'` resource. If you specify a value of $1D as the keyboard equivalent of the menu item, the Menu Manager reduces the `'ICON'` resource to fit in a 16-by-16 bit rectangle. If you provide an `'SICN'` resource and specify a value of $1E as the keyboard equivalent of a menu item, the Menu Manager plots the small icon in a 16-by-16 bit rectangle. If you provide a `'cicn'` resource, the Menu Manager automatically enlarges the enclosing rectangle of the menu item according to the rectangle specified in the `'cicn'` resource. (For the Apple and Application menus, the Menu Manager automatically reduces the icon to fit within the enclosing rectangle of a menu item or uses the appropriate icon from the application's icon family, such as an `'ics8'` resource, if one is available.) See the chapter "Finder Interface" in this book for details on how to create icons for your application.

To assign a keyboard equivalent to a menu item, specify the 1-byte character that the user types in addition to the Command key in place of the `nokey` constant in your resource definition for the menu item. If your application attaches a submenu to a menu item, then specify the `hierarchicalMenu` constant in place of the `nokey` constant. A menu item can have either a keyboard equivalent or submenu defined for it, but not both. To indicate that a menu item has an icon that is defined in an `'SICN'` resource, specify $1E in place of the `nokey` constant. To indicate that a menu item has an icon that is defined in an `'ICON'` resource and that the Menu Manager should reduce this icon to a 16-by-16 bit rectangle, specify $1D in place of the `nokey` constant. Menu items that have small icons or reduced icons cannot have keyboard equivalents.

To set the script code of a menu item's text, specify $1C in place of the `nokey` constant and define the desired script code in place of the `noicon` constant. If an item contains $1C in its keyboard equivalent field and a script code in its icon field, the Menu Manager draws the item's text in the script identified by the script code value if the corresponding script system is installed. If you do not specify a script code for a menu item, the Menu Manager displays the menu item's text in the system font of the current system script. For Roman scripts, the system font is Chicago and the system font size is 12.

To assign a mark that appears to the left of the menu item text and to the left of any icon, specify the marking character in place of the `nomark` constant in your resource definition. If the menu item has a submenu, then specify the menu ID of the submenu in place of the `nomark` constant. Submenus of an application must use menu IDs from 1 through 235; submenus of a desk accessory must use menu IDs from 236 through 255. Note that defining the menu ID of a submenu in a `'MENU'` resource does not attach the submenu to its menu. You must use the `GetMenu` function and `InsertMenu` procedure to do this. "Creating a Hierarchical Menu," which begins on page 3-53, gives information on attaching a submenu to its menu.

To assign a font style to a menu item, in your `'MENU'` resource use the constants `bold`, `italic`, `plain`, `outline`, and `shadow` to get their corresponding styles.

Listing 3-1 defines the second menu item as a divider. When you use a hyphen as the first character in the string that defines the text of a menu item, the Menu Manager creates a divider that extends across the entire width of the menu item. You cannot assign any other characteristics to a divider.

The `'MENU'` resource for the Apple menu does not list any other menu items. Use the `AppendResMenu` procedure to add the desktop items to the Apple menu after your application creates the menu. See "Adding Items to the Apple Menu" on page 3-68 for more information.

Once you create a menu, you can append additional items to it using the `AppendMenu`, `InsertMenuItem`, `InsertResMenu`, or `AppendResMenu` procedure. You can also change the characteristics of individual menu items using Menu Manager routines. See "Changing the Appearance of Items in a Menu" on page 3-57 for more information.

Figure 3-12 on page 3-24 shows a typical Edit menu for an application. Listing 3-2 shows a `'MENU'` resource for this Edit menu.

Listing 3-2 Rez input for a `'MENU'` resource for an Edit menu

```
#define mEdit 130
resource 'MENU' (mEdit, preload) {     /*resource ID, preload resource*/
   mEdit,                              /*menu ID*/
   textMenuProc,                       /*uses standard menu definition */
                                       /* procedure*/
   0b0000000000000000001001000000000,  /*enable/disable first 31 menu */
                                       /* items as appropriate*/
   enabled,                            /*enable title*/
   "Edit",                             /*text of menu title*/
    {                                  /*menu items*/
   "Undo",  noicon, "Z",   nomark, plain; /*keyboard equivalent Command-Z*/
   "-",     noicon, nokey, nomark, plain;
   "Cut",   noicon, "X",   nomark, plain; /*keyboard equivalent Command-X*/
   "Copy",  noicon, "C",   nomark, plain; /*keyboard equivalent Command-C*/
   "Paste", noicon, "V",   nomark, plain; /*keyboard equivalent Command-V*/
   "Clear", noicon, nokey, nomark, plain;
   "Select All",
            noicon, "A",   nomark, plain; /*keyboard equivalent Command-A*/
   "-",     noicon, nokey, nomark, plain;
   "Create Publisher…",
            noicon, nokey, nomark, plain;
   "Subscribe To…",
            noicon, nokey, nomark, plain;
   "Publisher Options…",
            noicon, nokey, nomark, plain;
   "-",     noicon, nokey, nomark, plain;
   "Show Clipboard",
            noicon, nokey, nomark, plain
      }
};
```

Listing 3-2 defines the resource ID of the Edit menu as 130, defines the menu ID of the Edit menu as 130, and specifies that this menu uses the standard menu definition procedure. The listing defines the initial enabled state of the first 31 menu items and also specifies that the menu title should be enabled when it is first created.

The resource listing defines the title of the menu, Edit. It then defines the characteristics of each menu item in the menu. For each menu item, you need to specify the text of the menu item and any other characteristics of the menu item. For example, Listing 3-2 defines the first item in the Edit menu as the Undo command with these characteristics: there is no icon to the left of the menu item text, the menu item has a keyboard equivalent of Command-Z, it does not have any mark to the left of the menu item text, and the style of the menu item text is plain. The listing defines the second menu item as a divider line. It defines the Cut, Copy, and Paste commands; specifies keyboard equivalents for each of them; and defines the rest of the items in the menu.

Listing 3-3 shows another example of a resource description of a menu, the File menu of a typical application.

Listing 3-3 Rez input for a `'MENU'` resource for a File menu

```
resource 'MENU' (mFile, preload) {
      mFile, textMenuProc,
      0b0000000000000000000010000000000,
      enabled,
      "File",
      {
         "New",         noicon, "N",   nomark, plain;
         "Open…",       noicon, "O",   nomark, plain;
         "-",           noicon, nokey, nomark, plain;
         "Close",       noicon, "W",   nomark, plain;
         "Save",        noicon, "S",   nomark, plain;
         "Save As…",    noicon, nokey, nomark, plain;
         "-",           noicon, nokey, nomark, plain;
         "Page Setup…", noicon, nokey, nomark, plain;
         "Print…",      noicon, "P",   nomark, plain;
         "-",           noicon, nokey, nomark, plain;
         "Quit",        noicon, "Q",   nomark, plain
      }
};
```

Creating a Menu Bar Resource

You typically define your application’s menu bar using a menu bar (`'MBAR'`) resource. Listing 3-4 shows an `'MBAR'` resource, in Rez format, for a sample application.

Listing 3-4 Rez input for an `'MBAR'` resource

```
#define rMenuBar      128

#define mApple        128
#define mFile         129
#define mEdit         130
```

```
resource 'MBAR' (rMenuBar, preload) {/*resource ID, preload*/
   /*menus appear in the order listed here*/
   { mApple, mFile, mEdit };        /*resource IDs for menus in */
                                    /* this menu bar*/
};
```

Listing 3-4 defines the `'MBAR'` resource with resource ID 128. This `'MBAR'` resource defines the order and resource IDs of the menus contained in it; it defines its first three menus as the menus with resource IDs 128, 129, and 130. The Menu Manager uses the assigned resource IDs to read in the menus when it creates a menu bar from an `'MBAR'` resource.

Setting Up Your Application's Menu Bar

To create a menu list as defined in an `'MBAR'` resource, use the `GetNewMBar` function. For each menu defined by the `'MBAR'` resource, the `GetNewMBar` function creates a menu record for the menu, creates each menu according to its resource definition in its corresponding `'MENU'` resource, and inserts each menu into the new menu list. The `GetNewMBar` function returns a handle to the created menu list. For example, this code creates a menu list for the menu bar defined by the `'MBAR'` resource with resource ID 128 (defined by the constant `rMenuBar`):

```
CONST
   rMenuBar = 128;
VAR
   menuBar:    Handle;

 menuBar := GetNewMBar(rMenuBar); {read menus and menu bar }
                                  { descriptions,create & return }
                                  { a handle to a new menu list}
```

Use the `SetMenuBar` procedure to set the current menu list to the menu list created by your application and the `DrawMenuBar` procedure to update the menu bar's appearance. For example, Listing 3-5 uses these two routines to set up the application's menu bar.

Listing 3-5 Setting up an application's menus and menu bar

```
PROCEDURE MyMakeMenus;
VAR
   menuBar:                Handle;
 BEGIN
{first use the GetNewMBar function to read menus in & create a }
{ new menu list. If you define an Apple menu, the Menu Manager }
{ inserts the Help and Application menus (and Keyboard menu if }
{ necessary) into the newly created menu list}
```

```
   menuBar := GetNewMBar(rMenuBar);
   IF menuBar = NIL THEN
      EXIT(MyMakeMenus);
   SetMenuBar(menuBar); {insert menus into the current menu list}
   DisposHandle(menuBar);
                        {add desktop items in Apple Menu Items }
                        { folder to Apple menu}
   AppendResMenu(GetMenuHandle(mApple), 'DRVR');
   MyAdjustMenus;       {adjust items and enabled state of menus}
   DrawMenuBar;         {draw the menu bar}
 END;
```

The code in Listing 3-5 creates the application's menu bar by reading in the definition from the 'MBAR' resource with resource ID 128, and it uses SetMenuBar to set the current menu list to the newly created menu list. The code then adds the desktop items in the Apple Menu Items folder to the Apple menu using the AppendResMenu procedure.

You can use the GetMenuHandle function to get a handle to the menu record of any menu in the current menu list. You supply the menu ID of the desired menu as a parameter to GetMenuHandle, and GetMenuHandle returns a handle to the menu's menu record. Most Menu Manager routines require either a menu ID or a handle to a menu record as a parameter.

After creating the menu bar and adding any other menus or items as necessary, the code calls the MyAdjustMenus procedure to adjust the application's menus—for example, this procedure sets the enabled and disabled states of menu items in accordance with the current state of the application. (Listing 3-19 on page 3-74 shows the application-defined MyAdjustMenus procedure used in Listing 3-5.) After adjusting the menus, the code in Listing 3-5 uses DrawMenuBar to draw the menus in the menu bar according to their current enabled state and as they are defined in the current menu list.

Usually you'll define the menus of your application and its menu bar using 'MENU' resources and an 'MBAR' resource and using the GetNewMBar function to read the resource definitions. However, you can choose to read in a 'MENU' resource using the GetMenu function or to create a new empty menu using NewMenu. You can then insert a menu into the current menu list using the InsertMenu procedure. See "Creating Menus" on page 3-105 and "Adding Menus to and Removing Menus From the Current Menu List" on page 3-108 for information on forming your menus using these routines.

If your application uses a submenu, you need to use the GetMenu function and InsertMenu procedure to make these menus available to your application. See "Creating a Hierarchical Menu" on page 3-53 for information on creating submenus. If your application uses a pop-up menu, you can use the pop-up control definition function and Dialog Manager or Control Manager routines to create and display the pop-up menu. See "Creating a Pop-Up Menu" on page 3-56 for information on creating pop-up menus.

The Menu Manager creates and initializes your application's menu color information table when your application calls `GetNewMBar`. You can add entries to your application's menu color information table if you want to use colors other than the default colors in your menus and menu bar. You can add entries to this table by providing menu color information table (`'mctb'`) resources or by using the `SetMCEntries` procedure. Usually you should use the default colors to help maintain a consistent user interface.

If you add menu color entries to your application's menu color information table and your application uses more than one menu bar, you need to save a copy of your application's menu color information table before changing menu bars. Use the `GetMCInfo` function before calling `GetNewMBar` and call `SetMCInfo` afterward to restore the menu color information table. Listing 3-6 shows a routine that saves and then restores the menu color information table when creating a new menu bar.

Listing 3-6 Saving and restoring menu color information

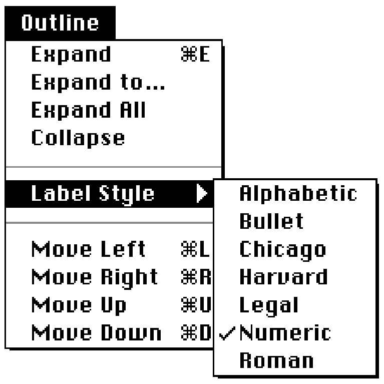
```
PROCEDURE MyChangeMenuBarAndSaveColorInfo;
CONST
   rMenuBar2 = 129;
VAR
   menu:             MenuHandle;
   menuBar:          Handle;
   currentMCTable:   MCTableHandle;
   newMCTable:       MCTableHandle;
 BEGIN
   currentMCTable := GetMCInfo;     {save menu color info table}
   IF currentMCTable = NIL THEN
      EXIT(MyChangeMenuBarAndSaveColorInfo);
   menuBar := GetNewMBar(rMenuBar2);{read menus in & create new menu list}
   IF menuBar = NIL THEN
      EXIT(MyChangeMenuBarAndSaveColorInfo);
   newMCTable := GetMCInfo;         {get new menu color info table}
   IF newMCTable = NIL THEN
      EXIT(MyChangeMenuBarAndSaveColorInfo);
   SetMCInfo(currentMCTable);       {restore previous menu color info table}
   SetMenuBar(menuBar);             {insert menus into the current menu list}
   DisposHandle(menuBar);
   AppendResMenu(GetMenuHandle(m2Apple), 'DRVR'); {add desktop items from }
                                    { Apple Menu Items folder to Apple menu}
   MyAdjustMenus;                         {adjust menu items}
   DrawMenuBar;                           {draw the menu bar}
 END;
```

Creating a Hierarchical Menu

A hierarchical menu is a menu that has a submenu attached to one or more of its menu items. Submenus can be useful when your application needs to offer additional choices to the user without taking up extra space in the menu bar. If you use a submenu in your application, use it to give the user additional choices or to choose attributes, not additional commands.

A menu item of a pull-down menu is the title of the attached submenu. A menu item that has a triangle facing right in the location of the keyboard equivalent identifies that a submenu is attached to the menu item. The title of a submenu should represent the choices it contains. Figure 3-32 shows a menu with a submenu whose menu title is Label Style.

Figure 3-32 A menu item with a submenu



When a user drags the cursor through a menu and rests it on a menu item with a submenu attached to it, the Menu Manager displays the submenu after a brief delay. The title of the submenu remains highlighted while the user browses through the submenu; the Menu Manager unhighlights the menu title of the submenu when the user releases the mouse button.

Your application is responsible for placing any marks next to the current choice or attribute of the submenu. For example, in Figure 3-32 the application placed the checkmark next to the Numeric menu item to indicate the current choice. If the user makes a new choice from the menu, your application should update the menu items accordingly.

You can specify that a particular menu item has a submenu by identifying this characteristic (using the `hierarchicalMenu` constant) when you define the menu item in its `'MENU'` resource. You cannot assign keyboard equivalents to a menu item that has a submenu. (You can define keyboard equivalents for the menu items in the submenu, but this is not recommended.) You identify the menu ID of the submenu in place of the marking character in the menu item's resource description. Thus, a menu item that has a submenu cannot have a marking character and cannot have a keyboard equivalent.

To insert a submenu into the current menu list, you must use the `InsertMenu` procedure. The `GetNewMBar` function does not read in the resource descriptions of any submenus.

Listing 3-7 shows the `'MENU'` resource for an application-defined menu called Outline. The Outline menu contains a number of menu items, including the Label Style menu item. The description of this menu item contains the constant `hierarchicalMenu`, which indicates that the item has a submenu. This menu item description also contains the menu ID of the submenu (defined by the `mSubMenu` constant). The menu ID of a submenu of an application must be from 1 through 235; the menu ID of a submenu of a desk accessory must be from 236 through 255.

The submenu is defined by the menu with the menu ID specified by the Label Style menu item. You define the menu items of a submenu in the same way as you would a pull-down menu (except you shouldn't define keyboard equivalents for items in a submenu, and you shouldn't attach a submenu to another submenu).

Listing 3-7 Rez input for a description of a hierarchical menu with a submenu

```
#define mOutline 135
#define mSubMenu 181

resource 'MENU' (mOutline, preload) {
      mOutline ,                          /*menu ID*/
      textMenuProc,
      0b0000000000000000000000000010000,
      enabled,
      "Outline",                          /*menu title*/
      {                                   /*menu items*/
         "Expand",            noicon, "E",   nomark, plain;
         "Expand To…",        noicon, nokey, nomark, plain;
         "Expand All",        noicon, nokey, nomark, plain;
         "Collapse",          noicon, nokey, nomark, plain;
         "-",                 noicon, nokey, nomark, plain;
         /*the Label Style item has a submenu with menu ID mSubMenu*/
         "Label Style",       noicon, hierarchicalMenu, mSubMenu, plain;
         "-",                 noicon, nokey, nomark, plain;
         "Move Left",         noicon, "L",   nomark, plain;
         "Move Right",        noicon, "R",   nomark, plain;
         "Move Up",           noicon, "U",   nomark, plain;
         "Move Down",         noicon, "D",   nomark, plain
      }
};
```

```
resource 'MENU' (mSubMenu , preload) {
      mSubMenu ,                          /*menu ID*/
      textMenuProc,
      0b0000000000000000000000001111111,
      enabled,
      "Label Style",                      /*menu title (ignored--defined */
                                          /* by parent menu item text)*/
      {                                   /*menu items*/
         "Alphabetic",        noicon, nokey, nomark, plain;
         "Bullet",            noicon, nokey, nomark, plain;
         "Chicago",           noicon, nokey, nomark, plain;
         "Harvard",           noicon, nokey, nomark, plain;
         "Legal",             noicon, nokey, nomark, plain;
         "Numeric",           noicon, nokey, nomark, plain;
         "Roman",             noicon, nokey, nomark, plain
      }
};
```

When you use `GetNewMBar` to read in menu descriptions and create a new menu list, `GetNewMBar` records the menu ID of any submenu in the menu record but does not read in the description of the submenu. To read a description of a submenu, use the `GetMenu` function. To actually insert a submenu into the current menu list, you must use the `InsertMenu` procedure.

When needed, your application can use the `GetMenu` function to read a description of the characteristics of a menu from a `'MENU'` resource. The `GetMenu` function creates a menu record for the menu, allocating space for the menu record in your application's heap. The `GetMenu` function creates the menu and menu items (and fills in the menu record) according to its `'MENU'` resource. The `GetMenu` function does not insert the menu into a menu list. When you're ready to add it to the current menu list, use the `InsertMenu` procedure.

Listing 3-8 uses the `GetMenu` function to read the description of a submenu and uses the `InsertMenu` procedure to insert the menu into the current menu list.

Listing 3-8 Creating a hierarchical menu

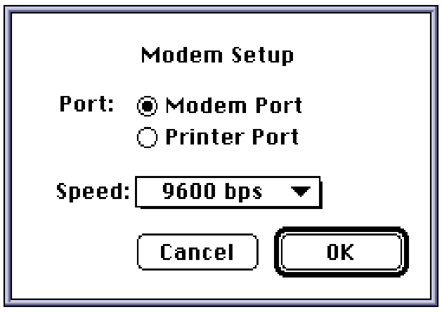
```
PROCEDURE MyMakeSubMenu (subMenuResID: Integer);
VAR
   subMenu: MenuHandle;
BEGIN
   subMenu := GetMenu(subMenuResID);
   InsertMenu(subMenu, -1);
END;
```

To insert a submenu into the current menu list using the `InsertMenu` procedure, specify –1 in the second parameter to insert the menu into the submenu portion of the menu list. As the user traverses menu items, if a menu item has a submenu the `MenuSelect` function looks in the submenu portion of the menu list for the submenu; it searches for a menu with a defined menu ID that matches the menu ID specified by the hierarchical menu item. If it finds a menu with a matching menu ID, it attaches the submenu to the menu item and allows the user to browse through the submenu.

Creating a Pop-Up Menu

In System 7, pop-up menus are implemented as controls. You define the menu items of a pop-up menu in the same way as in other menus (using a `'MENU'` resource), and you define specific features of the pop-up menu itself (such as the location of the pop-up menu) in a control that uses the standard pop-up control definition function. Pop-up menus provide the user with a simple way to select from among a list of choices without having to move up to the menu bar. They are particularly useful in a dialog box that requires the user to specify a number of settings or values. Figure 3-33 shows an example of a pop-up menu in a dialog box.

Figure 3-33 A pop-up menu in a dialog box



To create a pop-up menu, create a control that uses the pop-up control definition function, define the pop-up menu and its menu items, and associate the control with a window or dialog box. You can use Dialog Manager or Control Manager routines to display pop-up menus.

For example, if you define a modal dialog box that contains a pop-up control and use the Dialog Manager to display and help handle events in the dialog box, the Dialog Manager automatically uses the pop-up control definition function to draw the control and also to handle user interaction when the user presses the mouse button while the cursor is over a pop-up control.

If your application defines a control in one of your application's windows, you can use `TrackControl` and other Control Manager routines to handle the pop-up menu.

The pop-up control definition function draws a box around the pop-up box, draws the drop shadow, inserts the text into the pop-up box, draws the downward-pointing triangle, and draws the pop-up title. When a dialog box contains a control that uses the pop-up control definition function and the user presses the mouse button while the cursor is in the pop-up control, the pop-up control definition function highlights the pop-up menu title, displays the pop-up menu, and handles all user interaction until the user releases the mouse button. When the user releases the mouse button, the pop-up control definition function closes the pop-up box, draws the user's choice in the pop-up box (or restores the previous item if the user did not make a new choice), stores the user's choice as the value of the control, and unhighlights the pop-up menu title. Your application can use the Control Manager function `GetControlValue` to get the value of the control and to determine the currently selected item in the pop-up menu.

To create a pop-up control, create a control and specify that the control uses the pop-up control definition function by specifying the `popupMenuProc` constant:

```
CONST popupMenuProc = 1008;      {pop-up menu control}
```

If you specify `popupMenuProc` (plus any appropriate variation code) as the `procID` field of the resource description of a control, when your application creates the control (by using the Dialog Manager or by using `GetNewControl`) the Control Manager creates the pop-up control, which includes the pop-up title and the pop-up box with a one-pixel drop shadow. The appearance of the pop-up title and the values in the menu are controlled by other values stored in the resource (or other parameters passed to `NewControl`). See the chapter "Control Manager" in this book for information on the `NewControl` function.

If your application does not use the standard pop-up control definition function, you can create your own control definition function to implement pop-up menus. In this case you can use the `PopUpMenuSelect` function to draw the pop-up menu and track the cursor within the menu. Your application is responsible for highlighting the title of the pop-up menu before calling `PopUpMenuSelect` and unhighlighting the title afterward (to duplicate the behavior of menu titles in the menu bar). Your application must also set the mark of the items in the pop-up menu as appropriate if you use the `PopUpMenuSelect` function.

For more information on creating controls, see the chapter "Control Manager" in this book. For listings that define the dialog box shown in Figure 3-33, see the chapter "Dialog Manager" in this book.

Changing the Appearance of Items in a Menu

You can change the appearance of an item in a menu using Menu Manager routines. For example, you can change the font style, text, or other characteristics of menu items. You can also enable or disable a menu item.

Most of the Menu Manager routines that get or set characteristics of a particular menu item require three parameters:

- a handle to the menu record of the menu containing the desired menu item
- the number of the menu item
- a variable that either specifies the data to set or identifies where to return information about that item

Enabling and Disabling Menu Items

Using the `EnableItem` and `DisableItem` procedures, you can enable and disable specific menu items or an entire menu. You pass as parameters to these two procedures a handle to the menu record that identifies the desired menu and either an item number that identifies the particular menu item to enable or disable or a value of 0 to indicate that the entire menu should be enabled or disabled.

Your application should always enable and disable any menu items as appropriate—according to the user's content—before calling `MenuSelect` or `MenuKey`. For example, you should enable the Paste command when the scrap contains data that the user can paste. (Listing 3-19 on page 3-74 shows code that adjusts an application's menus.)

When you disable or enable an entire menu, call `DrawMenuBar` to update the menu bar. The `DrawMenuBar` procedure draws the menus in the menu bar according to their current enabled state and as they are defined in the current menu list.

If you disable an entire menu, the Menu Manager dims the menu title at your application's next call to `DrawMenuBar` and dims all items in the menu when it displays the menu. If you enable an entire menu, the Menu Manager enables only the menu title and any items that you did not previously disable individually; the Menu Manager does not enable any item that your application previously disabled by calling `DisableItem` with that menu item's item number. For example, if all items in your application's Edit menu are enabled, you can disable the Cut and Copy commands individually using `DisableItem`. If you choose to disable the entire menu by passing 0 as the menu item parameter to `DisableItem`, the menu and all its items are disabled. If you then enable the entire menu by passing 0 as the menu item parameter to `EnableItem`, the menu and its items are enabled, except for the Cut and Copy commands, which remain disabled. In this case, to enable the Cut and Copy commands, you must enable each one individually using `EnableItem`.

You can use `DisableItem` to disable items that aren't appropriate at a given time. For example, you can disable the Cut and Copy commands when the user has not selected anything to cut or copy and disable the Paste command when the scrap is empty.

This code enables the File menu, disables the Cut and Copy commands in the Edit menu, and disables the application-defined menu Colors.

```
VAR
   menu: MenuHandle;

   menu := GetMenuHandle(mFile); {get a handle to the File menu}
   EnableItem(menu, 0);          {enable File menu and any items }
                                 { not individually disabled}
   DrawMenuBar;                  {update menu bar's appearance}

   menu := GetMenuHandle(mEdit); {get a handle to the Edit menu}
   DisableItem(menu, iCut);      {disable the Cut command}
   DisableItem(menu, iCopy);     {disable the Copy command}
   menu := GetMenuHandle(mColors);{get a handle to Colors menu}
   DisableItem(menu, 0);         {disable Colors menu & all }
                                 { items in it}
   DrawMenuBar;                  {update menu bar's appearance}
```

If you disable or enable an entire menu, call `DrawMenuBar` when you need to update the menu bar's appearance. If you do not need to update the menu bar immediately, you can use the `InvalMenuBar` procedure instead of `DrawMenuBar`, thus reducing flickering in the menu bar. Rather than drawing the menu bar twice as in the previous example, you can use `InvalMenuBar` instead of `DrawMenuBar`, causing the Event Manager to redraw the menu bar the next time it scans for update events. The `InvalMenuBar` procedure is available in System 7 and later. See page 3-114 for additional details on the `InvalMenuBar` procedure.

Changing the Text of an Item

You can get or set the text of a menu item using Menu Manager routines.

To get the text of a menu item, use the `GetMenuItemText` procedure. For example, you can use the `GetMenuItemText` procedure to get the text of a menu item that you added to a menu using `InsertResMenu` or `AppendResMenu`.

To set the text of a menu item, use the `SetMenuItemText` procedure. You can use the `SetMenuItemText` procedure as a convenient way to change the text of a menu command that allows the user to toggle between two states. For example, if your application has a menu command that allows the user to either show or hide the Clipboard window, depending on whether the window is currently showing, you can change the text of the menu item at the appropriate time using the `SetMenuItemText` procedure.

Listing 3-9 changes the text of a menu item from Hide Clipboard to Show Clipboard or vice versa, based on the state of an application-defined global variable (`gToggleState`) that holds the state information.

Listing 3-9 Changing the text of a menu item

```
PROCEDURE MyToggleHideShow;
VAR
   myMenu:     MenuHandle;
   item:       Integer;
   itemString: Str255;
BEGIN
   myMenu := GetMenuHandle(mEdit);
   item := iToggleHideShow;
   IF gToggleState = kShow THEN
   BEGIN
      GetIndString(itemString, kMyStrings, kShowClipboard);
      gToggleState := kHide;
   END
   ELSE
   BEGIN
      GetIndString(itemString, kMyStrings, kHideClipboard);
      gToggleState := kShow;
   END;
   SetMenuItemText(myMenu, item, itemString);
END;
```

Note that if you use the `SetMenuItemText` procedure, you should define the text of the menu item in a string resource or string list resource (for example, using an `'STR '` or `'STR#'` resource). This makes your application easier to localize.

Changing the Font Style of Menu Items

You can change or get the font style of a menu item using the `SetItemStyle` or `GetItemStyle` procedure. To set the style of a menu item, specify a handle to the menu record of the menu containing the menu item whose style you want to set, specify the number of the menu item to set, and specify the desired style.

You specify the style using values from the set defined by the `Style` data type:

```
TYPE
      StyleItem = (bold, italic, underline, outline, shadow,
                   condense, extend);
      Style = SET OF StyleItem;
```

You can set the style of a menu item to zero, one, or more than one of the styles defined by the `StyleItem` data type. You can set the style of a menu item to the empty set to obtain the plain font style.

Listing 3-10 shows code that sets the style of menu items listed in an application's Style menu.

Listing 3-10 Setting the font style of menu items

```
VAR
   menu:       MenuHandle;
   itemStyle:  Style;

   menu := GetMenuHandle(mStyle); {get a handle to the Style menu}
   itemStyle := [italic];
   SetItemStyle(menu, iItalic, itemStyle);{set to italic style}
   itemStyle := [bold];
   SetItemStyle(menu, iBold, itemStyle);{set item to bold style}
   itemStyle := [bold, Italic];
   SetItemStyle(menu, iBoldItal, itemStyle);{bold & italic style}
   itemStyle := [];
   SetItemStyle(menu, iPlain, itemStyle);{set item to plain style}
```

To get the style of a menu item, you can use the `GetItemStyle` procedure.

Changing the Mark of Menu Items

You can change or get the mark of a menu item using the `SetItemMark` or `GetItemMark` procedure. To set the mark of a menu item to a checkmark, you can use either the `CheckItem` or the `SetItemMark` procedure.

To set the mark of a menu item, specify a handle to the menu record of the menu containing the item whose mark you want to set, specify the number of the menu item to set, and specify the mark to use as the marking character of the menu item.

You typically use checkmarks and dashes in menus that contain commands that set attributes and that you have grouped in accumulating groups. For example, you use a combination of checkmarks and dashes in the Style menu to indicate whether the selection contains more than one style. Figure 3-8 on page 3-15 shows an example of using checkmarks and dashes in a menu. “Groups of Menu Items” beginning on page 3-14 gives guidelines for determining how to group your menu items.

You specify the mark of the menu item by passing a character as one of the parameters to the `SetItemMark` procedure. You should use only the standard marking characters, such as the checkmark, diamond, or dash, in your menu items; avoid using other marks that might confuse the user. You can use the constants listed here to specify that the item has no mark or to set the marking character to a checkmark or diamond:

```
CONST noMark      = 0;     {no marking character}
      checkMark   = $12;   {checkmark}
      diamondMark = $13;   {diamond symbol}
```

As another example of the use of marks in menus, Listing 3-11 shows code that sets the mark of items in an application-defined Directory menu. It sets the marking character of the menu item of the last directory accessed to a checkmark, sets the marking character of the second-last directory accessed to the diamond mark, and removes the mark from the third-last directory accessed.

Listing 3-11 Adding marks to and removing marks from menu items

```
VAR
   menu:       MenuHandle;
   itemMark:   Char;
                                  {get handle to Directory menu}
   menu := GetMenuHandle(mDirectory);

   itemMark := CHR(checkMark);
   SetItemMark(menu, gLastDir, itemMark); {set mark to checkmark}

   itemMark := CHR(diamondMark);
   SetItemMark(menu, gOldLastDir, itemMark); {set mark to diamond}

   itemMark := CHR(noMark);
   SetItemMark(menu, gSecondLastDir, itemMark);{remove any mark}
```

You can also set the mark of a menu item to a checkmark using the `CheckItem` procedure:

```
VAR
   menu:       MenuHandle;
                              {get handle to Directory menu}
   menu := GetMenuHandle(mDirectory);
   CheckItem(menu, gLastDir, TRUE);       {set to checkmark}
   CheckItem(menu, gSecondLastDir, FALSE);{remove checkmark or }
                                          { any other mark}
```

Changing the Icon or Script Code of Menu Items

You can change or get the icon of a menu item using the `SetItemIcon` or `GetItemIcon` procedure. You can also use these procedures to get or set the script code of a menu item's text.

To set the script code of a menu item using the `SetItemIcon` procedure, you need to

- specify a handle to the menu record of the menu containing the item whose script code you want to set
- specify the number of the menu item to set
- specify the script code

To set a menu item's script code, you must also define the keyboard equivalent field of the item to $1C. If an item contains $1C in its keyboard equivalent field and a script code in its icon field, the Menu Manager draws the item in the script identified by the script code value if the corresponding script system is installed.

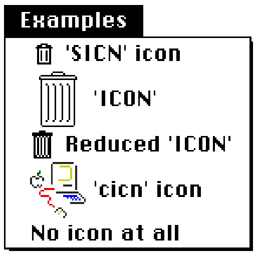
To set the icon of a menu item using the `SetItemIcon` procedure, you need to

- specify a handle to the menu record of the menu containing the item whose icon you want to set
- specify the number of the menu item to set
- specify the icon number (the Menu Manager uses the icon number to generate the resource ID of the icon)

The icon number that you specify to `SetItemIcon` must be a value from 1 through 255 for color icons or icons, from 1 through 254 for small icons and reduced icons, or 0 to specify that the item doesn't have an icon. The Menu Manager adds 256 to the number you specify and uses this calculated number as the icon's resource ID. For example, if you specify the icon number as 5, the Menu Manager uses the Resource Manager to find the icon with resource ID 261. The Menu Manager first looks for an icon resource of type `'cicn'`; if it can't find one with the calculated resource ID number (or if the computer doesn't have Color QuickDraw), it looks for a resource of type `'SICN'` if the keyboard equivalent field contains $1E; otherwise, it looks for an `'ICON'` resource.

Use either an `'ICON'` or `'SICN'` resource if you want to provide only a black-and-white icon. In addition, provide a `'cicn'` resource if you want the Menu Manager to use a color icon when Color QuickDraw is available. Figure 3-34 shows examples of icons in a menu item generated from icon resources: an `'SICN'` resource, an `'ICON'` resource, an `'ICON'` resource reduced to fit in a 16-by-16 bit rectangle, and a `'cicn'` resource.

Figure 3-34 Icons in menu items



The Menu Manager automatically fits the icon in the menu item according to your specifications. If the Menu Manager uses a `'cicn'` resource, it automatically enlarges the enclosing rectangle of the menu item according to the rectangle specified in the `'cicn'` resource. If the Menu Manager uses an `'ICON'` resource and the item specifies the `nokey` constant as the keyboard equivalent, the Menu Manager enlarges the rectangle of the menu item to fit the 32-by-32 bit `'ICON'` resource. You can request that the Menu Manager reduce an `'ICON'` resource to the size of a 16-by-16 bit small icon by specifying a value of $1D as the item's keyboard equivalent. To request that the Menu Manager use an `'SICN'` resource instead of an `'ICON'` resource, specify a value of $1E as the item's keyboard equivalent.

This code sets the icon of a menu item to a specified icon.

```
VAR
   menu:       MenuHandle;
   itemIcon:   Byte;

   itemIcon := 5;
   menu := GetMenuHandle(mWeather);
   {set the icon for this item in the Weather menu}
   SetItemIcon(menu, iBeachWeather, itemIcon);
```

Listing 3-12 shows the Rez description of three menu items, each of which contains icons. The first menu item has an icon with resource ID 261 (5 plus 256) and is defined by a resource type of either `'cicn'` or `'ICON'`. The second menu item has an icon with resource ID 262 (6 plus 256) and is identified by either a `'cicn'` resource or an `'ICON'` resource; however, in this case, the value of $1D requests the Menu Manager to reduce the `'ICON'` resource to a small icon. The third menu item has an icon with resource ID 263 (7 plus 256) and is defined by either a `'cicn'` resource or an `'SICN'` resource.

Listing 3-12 Specifying icons for menu items

```
#define mWeather 138
resource 'MENU' (mWeather, preload) {
      mWeather,
      textMenuProc,
      0b0000000000000000001011101100111,
      enabled, “Weather",
      {
         "Beach Weather",  /*item has icon or color icon */
                           /* with icon number 5*/
            5, nokey, nomark, plain;
         "Ski Weather",    /*item has reduced icon or color */
                           /* icon with icon number 6*/
            6, $1D, nomark, plain;
         "Kite-Flying Weather",/*item has small icon or */
                               /* color icon with icon number 7*/
            7, $1E, nomark, plain
      }
   };
```

See the chapter “Finder Interface” in this book for details on how to create icons.

Adding Items to a Menu

Usually you define a menu and all its items in a `'MENU'` resource. Occasionally you might need to add items to a menu after you’ve created it. After creating a menu (using `NewMenu`, `GetMenu`, or `GetNewMBar`), you can add items to it using the `AppendMenu`, `InsertMenuItem`, `AppendResMenu`, or `InsertResMenu` procedure.

You can use `AppendResMenu` or `InsertResMenu` to add items that consist of resource names to a menu. For example, you can use the `AppendResMenu` procedure to add fonts to your application’s Font menu or to add all of the desktop items from the Apple Menu Items folder to your application’s Apple menu. These are common instances when you’ll need to add items not already defined in a `'MENU'` resource to a menu. See “Adding Fonts to a Menu” on page 3-69 and “Adding Items to the Apple Menu” on page 3-68 for information on adding names of resources to menus.

If you add items to your application’s Help menu, you’ll need to use `AppendMenu` or `InsertMenuItem` to add the additional items. This section discusses how to add items using the `AppendMenu` and `InsertMenuItem` procedures, and “Adding Items to the Help Menu” on page 3-67 shows a specific example of adding items to the Help menu.

If you need to add items other than the names of resources to a previously created menu, you can use the `AppendMenu` or `InsertMenuItem` procedure. You can use `AppendMenu` to add items to the end of a menu; note that you can add items to only the end of the menu when using `AppendMenu`. Use `InsertMenuItem` to add items after any given item in a menu. When you add items to a menu using `AppendMenu` or `InsertMenuItem`, you can specify the same characteristics for menu items that are available to you when defining `'MENU'` resources.

You specify a handle to the menu record of the menu to which you want to add the item or items, and you specify a string describing the items to add as parameters to the `AppendMenu` or `InsertMenuItem` procedure. The string you pass to these procedures should consist of the text and any characteristics of the menu items. You can specify a hyphen as the menu item text to create a divider line. You can also use various metacharacters in the text string to separate menu items and to specify certain characteristics of the menu items. The metacharacters aren't displayed in the menu.

Here is a list of the metacharacters you can use in the text string that you specify to the `AppendMenu` or `InsertMenuItem` procedure:

Metacharacter	Description
`;` or Return	Separates menu items.
`^`	When followed by an icon number, defines the icon for the item. If the keyboard equivalent field contains $1C, this number is interpreted as a script code.
`!`	When followed by a character, defines the mark for the item. If the keyboard equivalent field contains $1B, this value is interpreted as the menu ID of a submenu of this menu item.
`<`	When followed by one or more of the characters B, I, U, O, and S, defines the character style of the item to Bold, Italic, Underline, Outline, or Shadow, respectively.
`/`	When followed by a character, defines the keyboard equivalent for the item. When followed by $1B, specifies that this menu item has a submenu. To specify that the menu item has a script code, small icon, or reduced icon, use the `SetItemCmd` procedure to set the keyboard equivalent field to $1C, $1D, or $1E, respectively.
`(`	Defines the menu item as disabled.

You can specify any, all, or none of these metacharacters in the text string to define the characteristics of a menu item. Note that the metacharacters that you specify aren't displayed in the menu item. (To use any of these metacharacters in the text of a menu item, first use `AppendMenu` or `InsertMenuItem`, specifying at least one character as the item's text. Then use the `SetMenuItemText` procedure to set the item's text to the desired string.)

Note

If you add menu items using the `AppendMenu` or `InsertMenuItem` procedure, you should define the text and any marks or keyboard equivalents in resources for easier localization. u

Listing 3-13 shows a string list (`'STR#'`) resource that stores the text of the menu items used in the next examples.

Listing 3-13 Rez input for text of menu items

```
resource 'STR#' (300, "Text for appended menu items") {
   {
   /*[1]*/
   "Just Text";
   /*[2]*/
   "Pick a Color…";
   /*[3]*/
   "(^2!=Everything<B/E";
   }
);
```

Here's code that uses the `AppendMenu` procedure to append a menu item with no specific characteristics other than its text to the menu identified by the menu handle in the `myMenu` variable. The text for the menu item is "Just Text" as stored in the `'STR#'` resource with resource ID 300.

```
VAR
   myMenu:     MenuHandle;
   itemString: Str255;

   myMenu := GetMenuHandle(mLibrary);
   GetIndString(itemString, 300, 1);
   AppendMenu(myMenu, itemString);
```

To insert an item after a given menu item, use the `InsertMenuItem` procedure. For example, this code inserts the menu item Pick a Color after the menu item with the item number specified by the `iRed` constant. The text for the new menu item consists of the string "Pick a Color…" as stored in the `'STR#'` resource with resource ID 300.

```
VAR
   myMenu:     MenuHandle;
   itemString: Str255;

   myMenu := GetMenuHandle(mColors);
   GetIndString(itemString, 300, 2);
   InsertMenuItem(myMenu, itemString, iRed);
```

If you do not explicitly specify a value for an item characteristic in the text string that you pass to `AppendMenu` or `InsertMenuItem`, the procedure assigns the default value for that characteristic. The Menu Manager defines the default item characteristics as no icon, no marking character, no keyboard equivalent, and plain text style. `AppendMenu` and `InsertMenuItem` enable the added menu items unless you specify otherwise.

This code appends a menu item with the text "Everything" to the menu identified by the menu handle in the `myMenu` variable. The text and other characteristics of this menu item are stored in the `'STR#'` resource shown in Listing 3-13. It also specifies that this menu item is disabled, has an icon with resource ID 258 (2 + 256), and has the "=" character as a marking character; the style of the text is Bold; and the menu item has a keyboard equivalent of Command-E.

```
VAR
   myMenu:     MenuHandle;
   itemString: Str255;

   myMenu := GetMenuHandle(mLibrary);
   GetIndString(itemString, 300, 3);
   AppendMenu(myMenu, itemString);
```

This code appends multiple items to the Edit menu using `AppendMenu`:

```
VAR
   myMenu:     MenuHandle;

   myMenu := GetMenuHandle(mEdit);
   AppendMenu(myMenu, 'Undo/Z;-;Cut/X;Copy/C;Paste/V');
```

The `InsertMenuItem` procedure differs from `AppendMenu` in how it handles the given text string when the text string specifies multiple items. The `InsertMenuItem` procedure inserts the items in the reverse of their order in the text string. For example, this code inserts menu items into the Edit menu using `InsertMenuItem` and is equivalent to the previous example:

```
VAR
   myMenu:     MenuHandle;

   myMenu := GetMenuHandle(mEdit);
   InsertMenuItem(myMenu, 'Paste/V';Copy/C;Cut/X;-;Undo/Z',0);
```

Once you've added a menu item to a menu, you can change any of its characteristics using Menu Manager routines, as described in "Changing the Appearance of Items in a Menu" on page 3-57.

Adding Items to the Help Menu

You add items to the Help menu by using the `HMGetHelpMenuHandle` function and either the `AppendMenu` or `InsertMenuItem` procedure.

The `HMGetHelpMenuHandle` function returns a copy of the handle to the menu record of your application's Help menu. Do not use the `GetMenuHandle` function to get a handle to the Help menu, because `GetMenuHandle` returns a handle to the global Help menu, not the Help menu that is specific to your application. Once you have a handle to the Help menu that is specific to your application, you can add items to it using the `AppendMenu` procedure or other Menu Manager routines. For example, Listing 3-14 adds the menu item displayed in Figure 3-19 on page 3-30.

Listing 3-14 Adding an item to the Help menu

```
PROCEDURE MyAddHelpItem;
VAR
   myMenu:     MenuHandle;
   myErr:      OSErr;
   itemString: Str255;
BEGIN
   myErr := HMGetHelpMenuHandle(myMenu);
   IF myErr = noErr THEN
      IF myMenu <> NIL THEN
      BEGIN
         {get the string (with index kSurfHelp) from the 'STR#' }
         { resource with resource ID kMyStrings}
         GetIndString(itemString, kMyStrings, kSurfHelp);
         AppendMenu(myMenu, itemString);
      END;
END;
```

When you add items to the Help menu, the Help Manager automatically adds a divider between the end of the standard Help menu items and your items.

Be sure to use an `'hmnu'` resource and specify the `kHMHelpMenuID` constant as the resource ID to provide help balloons for items you've added to the Help menu. (The Help Manager automatically creates the help balloons for the Help menu title and the standard Help menu items.) See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for specific information on the `'hmnu'` resource.

The Help Manager automatically processes the event when a user chooses any of the standard menu items in the Help menu. The Help Manager automatically enables and disables help when the user chooses Show Balloons or Hide Balloons from the Help menu. The setting of Balloon Help is global and affects all applications. See "Handling the Help Menu" on page 3-81 for information on responding to the user when the user chooses one of your appended items.

Adding Items to the Apple Menu

To insert the items contained in the Apple Menu Items folder into your application's Apple menu, use the `AppendResMenu` or `InsertResMenu` procedure and specify `'DRVR'` as the resource type. Doing so causes this procedure to automatically add all items in the Apple Menu Items folder to the specified menu.

The user can place any desktop object in the Apple Menu Items folder. When the user places an item in this folder, the system software automatically adds it to the list of items in the Apple menu of all open applications.

After inserting the Apple menu into your application's menu bar (by using GetNewMBar or GetMenu and InsertMenu), your application can add items to it. Listing 3-15 shows code that uses GetMenuHandle to get a handle to the application's Apple menu. The code then uses the AppendResMenu procedure, specifying that AppendResMenu should add all desktop items in the Apple Menu Items folder to the application's Apple menu.

Listing 3-15 Adding menu items to the Apple menu

```
VAR
   myMenu:            MenuHandle;

   myMenu := GetMenuHandle(mApple);
   IF myMenu <> NIL THEN
      AppendResMenu(myMenu, 'DRVR');{add desktop items in the }
                                    { Apple Menu Items folder }
                                    { to the Apple menu}
```

Listing 3-16 on page 3-70 shows a complete sample that sets up an application's menu bar, adds items to the Apple menu, adds items to the Font menu, and then updates the appearance of the menu bar.

Adding Fonts to a Menu

If your application provides a Font menu, you typically include the description of the menu in a 'MENU' resource, include a description of your menu bar in an 'MBAR' resource, and use GetNewMBar to create your menu bar and all menus in the menu bar. Once you've created the menu, you can use AppendResMenu to add the names of all font resources in the Fonts folder of the System Folder (or in system software versions earlier than 7.1, in the System file) as menu items in your application's Font menu. (You can also use InsertResMenu to insert the fonts into your menu.)

Listing 3-16 on the next page shows how to add names of font resources in the Fonts folder to an application's Font menu. The AppendResMenu procedure adds all resources of the specified type to a given menu. If you specify the resource type 'FONT' or 'FOND', the Menu Manager adds all resources of type 'FOND' and 'FONT' to the menu. ('NFNT' and 'sfnt' resources are specified through 'FOND' resources.)

The AppendResMenu and InsertResMenu procedures perform special processing for any font resources they find that have font numbers greater than $4000. The Menu Manager automatically sets the keyboard equivalent field of the menu item to $1C and stores the script code in the icon field of the menu item for any such 'FOND' resource. The Menu Manager displays a font name in its corresponding script if the script system for that font is installed.

Listing 3-16 Adding font names to a menu

```
PROCEDURE MyMakeAllMenus;
VAR
   menu:               MenuHandle;
   menuBar:            Handle;
 BEGIN
                       {read menus in & create new menu list}
   menuBar := GetNewMBar(rMenuBar);
   IF menuBar = NIL THEN
      EXIT(MyMakeAllMenus);
   SetMenuBar(menuBar); {insert menus into the current menu list}
   DisposHandle(menuBar);
   myMenu := GetMenuHandle(mApple);
   IF myMenu <> NIL THEN                  {add desktop items in }
      AppendResMenu(myMenu, 'DRVR');      { Apple Menu Items }
                                          { folder to Apple menu}
   myMenu := GetMenuHandle(mFont);
   IF myMenu <> NIL THEN
      AppendResMenu(myMenu, 'FONT');      {add font names to the }
         { Font menu--this adds all bitmapped and TrueType fonts }
         { in the Fonts folder to the Font menu}
   MyAddHelpItem;      {add app-specific item to Help menu}
   MyAdjustMenus;      {adjust menu items}
   DrawMenuBar;        {draw the menu bar}
 END;
```

Your application should indicate the current font to the user by placing the appropriate mark next to the text of the menu item that lists the font name. ("Changing the Mark of Menu Items" on page 3-61 explains how to add marks to and remove marks from menu items; Figure 3-13 on page 3-26 and Figure 3-14 on page 3-27 show examples of typical Font menus.)

If your application allows the user to change the font style or font size of text, you should provide separate Size and Style menus. See "Handling a Size Menu" beginning on page 3-82 for information on providing a Size menu in your application.

Handling User Choice of a Menu Command

If the user presses the mouse button while the cursor is in the menu bar, your application should first adjust its menus (enable or disable menu items and add marks to or remove marks from any items as appropriate to the user's context) and then call the `MenuSelect` function to allow the user to choose a menu command. The `MenuSelect` function handles all user interaction until the user releases the mouse button and returns a value as its function result that indicates which (if any) menu item the user chose.

For a command with a keyboard equivalent, your application should allow the user to choose the command by pressing the keys that correspond to the keyboard equivalent of that menu command. If the user presses the Command key and another key, your application should adjust its menus and then call the `MenuKey` function to map this combination to a keyboard equivalent. The `MenuKey` function returns as its function result a value that indicates the corresponding menu and menu item of the keyboard equivalent.

When the user chooses a menu command, your application should perform the action associated with that command. The `MenuSelect` and `MenuKey` functions highlight the menu title of the menu containing the chosen menu command. After your application performs any operation associated with the menu command chosen by the user, your application should unhighlight the menu title by using the `HiliteMenu` procedure.

However, if in response to a menu command your application displays a window that contains editable text (such as a modal dialog box), you should unhighlight the menu title immediately so that the user can access the Edit menu or other appropriate menus. In other words, any time the user can use a menu, make sure that the menu title is not highlighted.

When the user chooses a menu command that involves an operation that takes a long time, display the animated wristwatch cursor or display a status dialog box to give the user feedback that the operation is in progress.

If you want the users of your application to be able to record their actions (such as menu commands, text input, or any sequence of actions) for later playback, your application should send itself Apple events whenever a user performs a significant action. To do this for menu commands, your application typically sends itself an Apple event to perform the action associated with the chosen menu command. For example, when a user chooses the New command from the File menu, your application can choose to send itself a Create Element event. Your application then creates the new document in response to this event. For information on sending Apple events in response to menu commands, see *Inside Macintosh: Interapplication Communication*.

The next sections show how your application can

- determine if the user pressed the mouse button while the cursor was in the menu bar
- adjust its menus—enabling and disabling commands according to the current state of the document—before displaying menus or before responding to the user's choice of a keyboard equivalent of a command
- determine if the user chose the keyboard equivalent of a menu command
- respond to the user when the user chooses a menu command

The next sections also show how your application should respond when the user chooses an item from the Apple or Help menu.

Handling Mouse-Down Events in the Menu Bar

You can determine when the user has pressed the mouse button while the cursor is in the menu bar by examining the event record for a mouse-down event. You can use the Window Manager function `FindWindow` to map the mouse location at the time of the mouse-down event to a corresponding area of the screen. If the cursor was in the menu bar, your application should call the `MenuSelect` function, allowing the user to choose a menu command.

Listing 3-17 shows an application-defined procedure, `DoEvent`, that determines whether a mouse-down event occurred and, if so, calls another application-defined procedure to handle the mouse-down event. (For a complete discussion of how to handle events, see the “Event Manager” chapter in this book.)

Listing 3-17 Determining whether a mouse-down event occurred

```
PROCEDURE DoEvent (event: EventRecord);
BEGIN
   CASE event.what OF
      mouseDown:                {handle mouse-down event}
         DoMouseDown(event);
      {handle other events appropriately}
   END; {of CASE}
END;
```

Listing 3-18 shows an application-defined procedure, `DoMouseDown`, that handles mouse-down events. The `DoMouseDown` procedure determines where the cursor was when the mouse button was pressed and then responds appropriately.

Listing 3-18 Determining when the cursor is in the menu bar

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:       Integer;
   thisWindow: WindowPtr;
BEGIN
   part := FindWindow(event.where, thisWindow);
   CASE part OF
      inMenuBar:{mouse down in menu bar, respond appropriately}
         BEGIN
            {adjust marks and enabled state of menu items}
            MyAdjustMenus;
            {let user choose a menu command if desired}
            DoMenuCommand(MenuSelect(event.where));
         END;
      {handle other mouse-down events appropriately}
   END; {of CASE}
END;
```

You can use the FindWindow function to map the mouse location at the time the user pressed the mouse button to general areas of the screen. If the mouse location is in the menu bar, the FindWindow function returns the constant inMenuBar. In Listing 3-18, if the mouse location associated with the mouse-down event is in the menu bar, the DoMouseDown procedure first calls another application-defined procedure, MyAdjustMenus, to adjust the menus. Listing 3-19 shows the MyAdjustMenus procedure.

The DoMouseDown procedure then calls an application-defined procedure, DoMenuCommand. The DoMouseDown procedure passes as a parameter to the DoMenuCommand procedure the value returned from the MenuSelect function.

The MenuSelect function displays menus and handles all user interaction until the user releases the mouse button. The MenuSelect function returns a long integer indicating whether the user chose a menu command, and if so, it indicates which menu and which command the user chose.

Listing 3-24 on page 3-79 shows the DoMenuCommand procedure.

Adjusting the Menus of an Application

Your application should always adjust its menus before calling MenuSelect or MenuKey. For example, you should enable and disable any menu items as necessary and add checkmarks or dashes to items that are attributes. When you adjust your application’s menus, you should enable and disable menu items according to the type of window that is in the front. For example, when a document window is the frontmost window, you should enable items as appropriate for that document window. When a modeless dialog box or modal dialog box is the frontmost window, enable those items as appropriate to that particular dialog box. Listing 3-19 shows an application-defined routine, MyAdjustMenus, that adjusts the menus of the SurfWriter application appropriately.

The MyAdjustMenus procedure first determines what kind of window is in front and then adjusts the application’s menus appropriately. The application-defined MyGetWindowType procedure returns a value that indicates whether the window is a document window, a dialog window, or a window belonging to a desk accessory. It also returns the constant kNil if there isn’t a front window. (See the chapter “Window Manager” in this book for a listing of the MyGetWindowType procedure.) The MyAdjustMenus procedure calls other application-defined routines to adjust the menus as appropriate for the given window type.

Listing 3-19 Adjusting an application's menus

```
PROCEDURE MyAdjustMenus;
VAR
   window:              WindowPtr;
   windowType:          Integer;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   CASE windowType OF

   kMyDocWindow:
   BEGIN {document window is in front, adjust items 
appropriately}
      MyAdjustFileMenuForDocWindow;
      MyAdjustEditMenuForDocWindow;
      {adjust other menus as needed}
   END;  {of adjusting menus for a document window}

   kMyDialogWindow:
      {adjust menus accordingly for any dialog box}
      MyAdjustMenusForDialogs;

   kDAWindow:{adjust menus accordingly for a DA window}
      MyAdjustMenusForDA;
   kNil:{adjust menus accordingly when there isn't a front 
window}
      MyAdjustMenusNoWindows;
   END; {of CASE}
   DrawMenuBar;
END;
```

Listing 3-20 shows the application-defined procedure `MyAdjustFileMenuForDocWindow`. This procedure enables and disables the File menu for the application's document window, according to the state of the document. For example, this application always allows the user to create a new document or open a file, so the code enables the New and Open menu items. The code also enables the Close, Save As, Page Setup, Print, and Quit menu items. If the user has modified the file since last saving it, the code enables the Save command; otherwise, it disables the Save command.

Listing 3-20 Adjusting the File menu for a document window

```
PROCEDURE MyAdjustFileMenuForDocWindow;
VAR
   window:     WindowPtr;
   menu:       MenuHandle;
   myData:     MyDocRecHnd;
BEGIN
   window := FrontWindow;
   menu := GetMenuHandle(mFile); {get a handle to the File menu}
   IF menu = NIL THEN            {add your own error handling}
      EXIT (MyAdjustFileMenuForDocWindow);
   EnableItem(menu, iNew);
   EnableItem(menu, iOpen);
   EnableItem(menu, iClose);
   myData := MyDocRecHnd(GetWRefCon(window));
   IF myData^^.windowDirty THEN
      EnableItem(menu, iSave)
   ELSE
      DisableItem(menu, iSave);
   EnableItem(menu, iSaveAs);
   EnableItem(menu, iPageSetup);
   EnableItem(menu, iPrint);
   EnableItem(menu, iQuit);
END;
```

Listing 3-21 shows the application-defined `MyAdjustEditMenuForDocWindow` procedure.

Listing 3-21 Adjusting the Edit menu for a document window

```
PROCEDURE MyAdjustEditMenuForDocWindow;
VAR
   window:              WindowPtr;
   menu:                MenuHandle;
   selection, undo:     Boolean;
   isSubscriber:        Boolean;
   undoText:            Str255;
   offset:              LongInt;
BEGIN
   window := FrontWindow;
   menu := GetMenuHandle(mEdit); {get a handle to the Edit menu}
   IF menu = NIL THEN            {add your own error handling}
      EXIT (MyAdjustEditMenuForDocWindow);
   undo := MyIsLastActionUndoable(undoText);
   IF undo THEN   {if action can be undone}
   BEGIN
```

```
      SetMenuItemText(menu, iUndo, undoText);
      EnableItem(menu, iUndo);
   END

   ELSE           {if action can't be undone}
   BEGIN
      SetMenuItemText(menu, iUndo, gCantUndo);
      DisableItem(menu, iUndo);
   END;
   selection := MySelection(window);
   IF selection THEN
   BEGIN    {enable editing items if there's a selection}
      EnableItem(menu, iCut);
      EnableItem(menu, iCopy);
      EnableItem(menu, iCreatePublisher);
   END
   ELSE
   BEGIN    {disable editing items if there isn't a selection}
      DisableItem(menu, iCut);
      DisableItem(menu, iCopy);
      DisableItem(menu, iCreatePublisher);
   END;
   IF GetScrap(NIL, 'TEXT', offset) > 0 THEN
      EnableItem(menu, iPaste)   {enable if something to paste}
   ELSE
      DisableItem(menu, iPaste); {disable if nothing to paste}
   EnableItem(menu, iSelectAll);
   EnableItem(menu, iSubscribeTo);
   IF MySelectionContainsSubscriberOrPublisher(isSubcriber) THEN
   BEGIN {selection contains a single subscriber or publisher}
      IF isSubscriber THEN {selection contains a subscriber}
         SetMenuItemText(menu, iPubSubOptions, gSubOptText)
      ELSE                 {selection contains a publisher}
         SetMenuItemText(menu, iPubSubOptions, gPubOptText);
      EnableItem(menu, iPubSubOptions);
   END
   ELSE  {selection contains either no subscribers or publishers }
         { or contains at least one subscriber and one publisher}
      DisableItem(menu, iPubSubOptions);
   IF (gPubCount > 0) OR (gSubCount > 0) THEN
      EnableItem(menu, iShowHideBorders)
   ELSE
      DisableItem(menu, iShowHideBorders);
END;
```

The procedure in Listing 3-21 adjusts the items in the Edit menu as appropriate for a document window of the application. The code enables the Undo command if the application can undo the last command, enables the Cut and Copy commands if there's a selection that can be cut or copied, enables the Paste command if there's text data in the scrap, and enables the menu items relating to publishers and subscribers appropriately, according to whether the current selection contains a publisher or subscriber. The application-defined `MySelectionContainsSubscriberOrPublisher` function returns `TRUE` if the current selection contains a single subscriber or a single publisher and returns `FALSE` otherwise. If the `MySelectionContainsSubscriberOrPublisher` function returns `TRUE`, the code sets the text for the Publisher Options (or Subscriber Options) command and enables the menu item. If the function returns `FALSE`, the code disables the Publisher Options (or Subscriber Options) command.

Determining if the User Chose a Keyboard Equivalent

Keyboard equivalents of commands allow the user to invoke a menu command from the keyboard. You can determine if the user chose the keyboard equivalent of a menu command by examining the event record for a key-down event. If the user pressed the Command key in combination with another 1-byte character, you can determine if this combination maps to a Command-key equivalent by using the `MenuKey` function.

If your application supports keyboard equivalents that use other modifier keys in addition to the Command key, your application should examine the `modifiers` field and take any appropriate action; depending on the modifier keys you use, your application may or may not be able to use `MenuKey` to map the key to the menu command.

Listing 3-22 shows an application-defined procedure, `DoEvent`, that determines whether a key-down event occurred and, if so, calls an application-defined routine to handle the key-down event.

Listing 3-22 Determining when a key is pressed

```
PROCEDURE DoEvent (event: EventRecord);
BEGIN
   CASE event.what OF
      keyDown, autoKey:       {handle keyboard events}
         DoKeyDown(event);
      {handle other events appropriately}
   END; {of CASE}
END;
```

If your application determines that the user pressed a key, you need to determine whether the user chose the keyboard equivalent of a menu command. You can do this by examining the `modifiers` field of the event record describing the key-down event. If the Command key was also pressed, then your application should call the `MenuKey` function. The `MenuKey` function scans the current menu list for a menu item that has a matching keyboard equivalent and returns the menu and menu item, if any. Although you should not define the same keyboard equivalent for more than one command, the MenuKey function scans the menus from right to left, scanning the items from top to bottom, and returns the first matching keyboard equivalent that it finds.

If your application uses other keyboard equivalents in addition to Command-key equivalents, you can examine the state of the modifier keys and use the Event Manager function KeyTranslate, if necessary, to help map the keyboard equivalent to a particular menu item. See the discussion of 'KCHR' resources in *Inside Macintosh: Text* for information on how various keyboard combinations map to specific character codes.

Listing 3-23 shows an application's DoKeyDown procedure that handles key-down events and determines if a keyboard equivalent was pressed.

Listing 3-23 Checking a key-down event for a keyboard equivalent

```
PROCEDURE DoKeyDown (event: EventRecord);
VAR
   key:        Char;
BEGIN
   key := CHR(BAnd(event.message, charCodeMask));
   IF BAnd(event.modifiers, cmdKey) <> 0 THEN
   BEGIN       {Command key down}
      IF event.what = keyDown THEN
      BEGIN                {first enable/disable/check }
         MyAdjustMenus;    { menu items properly}
         DoMenuCommand(MenuKey(key));{handle the menu command}
      END;
   END
   ELSE
      MyHandleKeyDown(event);
END;
```

Listing 3-23 extracts the pressed key from the message field of the event record and then examines the modifiers field to determine if the Command key was also pressed. If so, the application first adjusts its menus and then calls an application-defined procedure, DoMenuCommand. The DoKeyDown procedure passes as a parameter to the DoMenuCommand procedure the value returned from the MenuKey function.

Listing 3-24 shows the DoMenuCommand procedure.

Responding When the User Chooses a Menu Item

Your application can use the MenuSelect function to determine when the user chooses a menu command, and your application can use the MenuKey function to determine when the user presses the keyboard equivalent for a menu command. Both MenuSelect and MenuKey return a long integer value that indicates which menu and menu item the user chose.

The MenuSelect and MenuKey functions return the menu ID of the menu in the high word and the menu item number in the low word of their function result. If the user did not choose a menu command or if the user pressed a keyboard combination that does not map to any keyboard equivalent in your application's menus, the functions return 0 in the high word and the value of the low word is undefined. The MenuSelect function also returns 0 in the high word when the user selects an item in the Application or Keyboard menu. The MenuSelect function (and MenuKey function, if the command has a keyboard equivalent) returns the kHMHelpMenuID constant in the high word and the menu item in the low word when the user selects an item that your application appended to the Help menu.

Listing 3-24 shows an application-defined procedure, DoMenuCommand. This procedure takes the appropriate action based on which menu command the user chose.

The DoMenuCommand procedure is called by the application after the application determines that either the user pressed the mouse button while the cursor was in the menu bar (in which case the application calls MenuSelect to allow the user to choose a command) or the user pressed the Command key and another key (in which case the application calls the MenuKey function). In either case, the application passes the function result returned by MenuSelect or MenuKey as a parameter to the DoMenuCommand procedure.

Listing 3-24 Responding to the user's choice of a menu command

```
PROCEDURE DoMenuCommand (menuResult: LongInt);
VAR
   menuID, menuItem: Integer;
BEGIN
   menuID := HiWord(menuResult);    {get menu ID of menu}
   menuItem := LoWord(menuResult);  {get menu item number}
   CASE menuID OF
      mApple:
         MyHandleAppleCommand(menuItem);
      mFile:
         MyHandleFileCommand(menuItem);
      mEdit:
         MyHandleEditCommand(menuItem);
      mFont:
         MyHandleFontCommand(menuItem);
      mSize:
         MyHandleSizeCommand(menuItem);
      kHMHelpMenuID:
         MyHandleHelpCommand(menuItem);
      mOutline:
         MyHandleOutlineCommand(menuItem);
      mSubMenu: {user chose item from submenu}
         MyHandleSubLabelStyleCommand(menuItem);
   END;  {end of CASE menuID}
   HiliteMenu(0); {unhighlight what MenuSelect or MenuKey hilited}
END;
```

The DoMenuCommand procedure calls other application-defined routines to perform the requested action. After performing the action associated with the chosen menu item, your application should use the HiliteMenu procedure to unhighlight the menu title to indicate that the requested action is complete.

Handling the Apple Menu

When the user chooses an item from the Apple menu, the MenuSelect function returns the menu ID of your application's Apple menu in the high word and returns the chosen menu item in the low word of its function result.

If your application provides an About command as the first menu item in the Apple menu and the user chose this item, you should display your application's About box. Otherwise your application should use the GetMenuItemText procedure to get the menu item text and then call the OpenDeskAcc function, passing the text of the chosen menu item as a parameter.

Listing 3-25 shows an application-defined procedure, MyHandleAppleCommand, that the application calls in response to the user's choice of an item from the Apple menu.

Listing 3-25 Responding to the user's choice of an item from the Apple menu

```
PROCEDURE MyHandleAppleCommand (menuItem: Integer);
VAR
   itemName: Str255;
   daRefNum: Integer;
BEGIN
   CASE menuItem OF
      iAbout:           {bring up alert for About}
         DisplayMyAboutBox;
      OTHERWISE
      BEGIN {all non-About items in this menu are desktop items, }
            { for example, DA's, other apps, documents, etc.}
         GetMenuItemText(GetMenuHandle(mApple), menuItem,
                         itemName);
         daRefNum := OpenDeskAcc(itemName);
      END;
    END; {of CASE}
END;
```

When the user chooses an item other than your application's About command from the Apple menu, your application should call the OpenDeskAcc function. The OpenDeskAcc function prepares to open the desktop object chosen by the user; for example, if the user chose a document created by the TeachText application, the OpenDeskAcc function schedules the TeachText application for execution (or prepares to open it if it isn't already open) and returns to your application. On your application's next call to WaitNextEvent, your application receives a suspend event, and then the Process Manager makes TeachText the foreground application and instructs TeachText to open the chosen document.

Handling the Help Menu

Both the MenuSelect and MenuKey functions return the kHMHelpMenuID constant (–16490) in the high word when the user chooses an appended item from the Help menu. The item number of the appended menu item is returned in the low word of the function result.

The DoMenuCommand procedure shown in Listing 3-24 determines which menu command was chosen by the user. If the user chose a command from the Help menu, the DoMenuCommand procedure calls the application-defined procedure MyHandleHelpCommand. Listing 3-26 shows the application-defined procedure MyHandleHelpCommand. This procedure illustrates how the SurfWriter application responds to the user's choice of an item from the application's Help menu. Note that you should use the HMGetHelpMenuHandle function, not the GetMenuHandle function, to get a handle to your application's Help menu.

Listing 3-26 Responding to the user's choice of a command from the Help menu

```
PROCEDURE MyHandleHelpCommand (menuItem: Integer);
VAR
   myHelpMenuHdl:             MenuHandle;
   origHelpItems, numItems:   Integer;
   myErr:                     OSErr;
BEGIN
   {get handle to your application's Help menu}
   myErr := HMGetHelpMenuHandle(myHelpMenuHdl);
   IF myErr <> noErr THEN
      EXIT(MyHandleHelpCommand);
   {count the number of items in the Help menu}
   numItems := CountMItems(myHelpMenuHdl);
   origHelpItems := numItems - kNumMyHelpItems;
   IF menuItem > origHelpItems THEN
   BEGIN {user chose an item added by this application}
      {adjust this application's global variables that hold item }
      { numbers of the menu items that this application appended}
      gMyHelpItem1 := origHelpItems +1;
      gMyHelpItem2 := origHelpItems +2;
      MyHelp(menuItem);
   END;
END;
```

Apple reserves the right to change the number of standard items in the Help menu. To determine the number of items in the Help menu, call the `CountMItems` function.

Handling a Size Menu

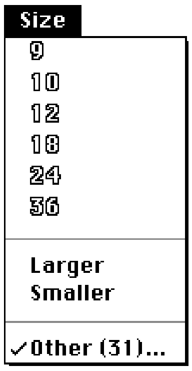
Your application can provide a Size menu to let the user choose various sizes of a font. Your Size menu should also provide the user with a method for specifying a size that isn't currently listed in the menu. For example, you can choose to provide an Other command that displays a dialog box allowing the user to choose a different font size. If the user chooses a font size not already in the menu, add a checkmark to the Other menu command and add the chosen size in parentheses to the text of the Other command.

Your application should outline font sizes to indicate which sizes are directly provided by the current font. For bitmapped fonts, outline only those sizes that actually exist in the Fonts folder. For TrueType fonts, outline all sizes that the TrueType font supports.

Your application should indicate the current font size to the user by placing a checkmark next to the text of the menu item that lists the current font size. If the current selection contains more than one font size, place a dash next to the name of each font size that the selection contains. ("Changing the Mark of Menu Items" on page 3-61 explains how to add marks to and remove marks from menu items.)

Figure 3-35 shows a Size menu as it appears after the user chooses a new font size of 31 by using the Other command. In Figure 3-35 the sizes 9, 10, 12, 18, 24, and 36 are the standard sizes provided by the application. Your application should place a checkmark next to the Other command to indicate that the current font size is a size other than a standard size. If the selection contains only one nonstandard size, include the size of the font in parentheses following the text Other. In Figure 3-35 the current selection contains a nonstandard size of 31, so the application places the checkmark next to the Other command and includes 31 in parentheses following the Other text. If the selection contains multiple nonstandard sizes, include the text Mixed in parentheses following the word Other. If the selection contains one or more standard sizes and only one nonstandard size, place a dash next to each standard size that the selection contains and place a dash next to the Other command with the nonstandard size included in paretheses in the text of the Other command.

Figure 3-35 A Size menu with user-specified size added



When the user chooses the Other command, you should display the current font size in a dialog box and allow the user to choose a new size. Figure 3-16 on page 3-28 shows a sample dialog box an application might display in response to the user's choice of the Other command.

You should always specify the text of the Other command in the plain font style (as shown in Figure 3-35) and never outlined, regardless of whether the current font is a TrueType font that supports that size or a bitmapped font that exists at that size in the Fonts folder.

Listing 3-27 shows an application-defined procedure that handles the user's choice of an item in the Size menu shown in Figure 3-35.

Listing 3-27 Handling the Size menu

```
PROCEDURE MyHandleSizeCommand (menuItem: Integer);
VAR
   numItems:      Integer;
   addItem:       Boolean;
   itemString:    Str255;
   itemStyle:     Style;
   sizeChosen:    LongInt;
BEGIN
   numItems := CountMItems(GetMenuHandle(mSize));
   IF menuItem = numItems THEN
   BEGIN                         {user chose Other command}
         {display a dialog box to allow the user to choose any }
         { size. If the user-specified size is not in the menu, }
         { add a checkmark to the Other command and add the }
         { new font size to the text of the Other command}
      MyDisplayOtherBox(sizeChosen);
   END
   ELSE
   BEGIN
      IF (menuItem = (numItems -2)) OR
         (menuItem = (numItems -3)) THEN
         DoMakeLargerOrSmaller(menuItem, sizeChosen)
      ELSE
      BEGIN             {user chose size displayed in the menu}
         {remove checkmark or dashes from menu items showing }
         { previous size}
         MyRemoveMarksFromSizeMenu;
         {add checkmark to menu item of new current size}
         CheckItem(GetMenuHandle(mSize), menuItem, TRUE);
         sizeChosen := MyItemToSize(menuItem);
      END;
   END;
   {update the document's state or the user's selection as needed}
   MyResizeSelection(sizeChosen);
END;
```

If the user chooses an item from the Size menu, the `MyHandleSizeCommand` procedure first counts the current number of items in the menu. If the user chooses the last item in the menu (the Other command), the procedure displays a dialog box like the one shown in Figure 3-16 on page 3-28 to let the user choose a size other than the ones currently shown in the menu. The application-defined function `MyDisplayOtherBox` also adds a checkmark to the Other command if the user chose a new size, adds the new size to the text of the Other command, and returns the chosen size in the `sizeChosen` variable.

If the user chose the Larger or Smaller command from the Size menu, the code calls an application-defined routine, `DoMakeLargerOrSmaller`, to perform the requested action. The `DoMakeLargerOrSmaller` procedure also adds a checkmark and adds the new size to the text of the Other command if the new size does not match any size in the menu. The procedure returns the chosen size in the `sizeChosen` variable.

If the user chose any size currently displayed in the menu, the `MyHandleSizeCommand` procedure adjusts the marking character of the menu items appropriately. The code removes the checkmark from the previous menu item and adds a checkmark to the menu item representing the new size chosen by the user. The code uses an application-defined function, `MyItemToSize`, to map the item number of the chosen menu item to a given size and returns this size in the `sizeChosen` variable.

The code then uses the application-defined procedure `MyResizeSelection` to update the document's state and resize the user's selection, if any, to the chosen size.

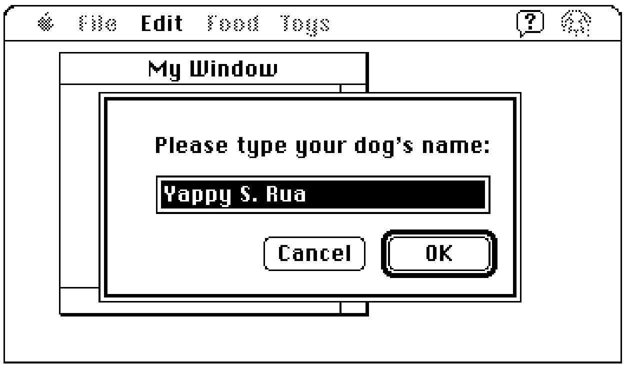
Accessing Menus From a Dialog Box

In System 7, the Menu Manager or your application can allow the user to access selected menus in the menu bar while interacting with an alert box or a modal dialog box. This allows users to make menu selections while your application is displaying an alert box or a modal dialog box. For example, a user might want to turn on Balloon Help for assistance in figuring out how to respond to an alert box. Similarly, if the modal dialog box contains several editable text fields, the user might find it simpler to copy text from one text field and paste it into another. Figure 3-36 shows a modal dialog box with an editable text field. Note that only the Edit and Help menus are enabled and all other menus are disabled. This gives the user access to editing commands and also to Balloon Help.

Note

In System 6, user access to menus in the menu bar is prohibited from an alert box or a modal dialog box unless your application specifically allows it. For example, in System 6, your application must provide a filter procedure to replace the standard filter procedure if you want to support the keyboard equivalents of the standard Edit menu commands in a modal dialog box. In System 7, you can let the Menu Manager enable these commands for you. u

Figure 3-36 Menu access from a modal dialog box



When your application displays a modeless or movable modal dialog box, your application should adjust its menus as appropriate for that dialog box. For example, when a movable modal dialog box is the frontmost window, your application should enable the Apple menu, enable the Edit menu if your dialog box contains an editable text item, enable or disable any other menus as needed, and disable any items it added to the Help menu if the user can't perform those actions while the dialog box is displayed.

When your application displays an alert box, system software automatically disables all of your application's menus except for the Help menu (in which all items are disabled except for the Show Balloons / Hide Balloons command).

When your application displays a modal dialog box, your application should also enable and disable its menus as appropriate. For example, you should enable the Edit menu if your dialog box contains an editable text item and disable any items it added to the Help menu if the user can't perform those actions while the dialog box is displayed. If your application handles access to the menu bar from a modal dialog box, it should disable the Apple menu or the first item in the Apple menu.

If your application does not specifically handle access to the menu bar from an alert box or a modal dialog box, in some cases the Menu Manager automatically disables the appropriate menus for you, as described in the following paragraphs.

When your application displays an alert box or a modal dialog box (that is, a window of type `dBoxProc`), the Menu Manager (in conjunction with the Dialog Manager) always appropriately adjusts the system-handled menus and performs these actions:

1. Disables all menu items in the Help menu except the Show Balloons (or Hide Balloons) command, which it enables.
2. Disables all menu items in the Application menu.
3. Enables the Keyboard menu if it appears in the menu bar, except for the About Keyboards command, which it disables.

In addition, if your application then calls the `ModalDialog` procedure, the Menu Manager (in conjunction with the Dialog Manager) performs two other actions:

4. Disables all of your application's menus.
5. Enables commands with the standard keyboard equivalents Command-X, Command-C, and Command-V if the modal dialog box contains a visible and active editable text field. The user can then use either the menu commands or their keyboard equivalents to cut, copy, and paste text. (The menu item having keyboard equivalent Command-X must be one of the first five menu items.)

When the user dismisses the modal dialog box, the Menu Manager restores all menus to the state they were in prior to the appearance of the modal dialog box.

In some cases actions 4 and 5 do not occur when you call `ModalDialog`. The enabling and disabling described in steps 4 and 5 do not occur if any of these conditions is true:

- Your application does not have an Apple menu.
- Your application has an Apple menu, but the menu is disabled when the modal dialog box is displayed.
- Your application has an Apple menu, but the first item in that menu is disabled when the dialog box is displayed.

Note

If your application already handles access to the menu bar from a modal dialog box and you do not want the automatic menu enabling and disabling provided by System 7 to occur, you should ensure that one or more of those conditions is true when you display a modal dialog box. u

When your application displays alert boxes or modal dialog boxes with no editable text items, your application can allow system software to handle menu bar access. In all other cases, your application should handle its own menu bar access.

System software always leaves the Help, Keyboard, and Application menus and their commands available when you display movable modal dialog boxes and modeless dialog boxes. For these types of dialog boxes, you must disable menus as appropriate and handle menu bar access as appropriate given their contents.

When your application displays a movable modal dialog box (a window of type `movableDBoxProc`), your application does not need to adjust the system-handled menus but should disable all its other menus except the Apple menu and—if your movable modal dialog box contains editable text items—the Edit menu. Leave the Apple menu enabled so that the user can use it to open other applications, and leave the Edit menu enabled so that the user can use the Cut, Copy, and Paste commands within the editable text item. (You can also leave your Undo and Clear commands enabled; otherwise, disable all other commands in the Edit menu.)

When your application removes a movable modal dialog box, modeless dialog box, or modal dialog box with editable text items, your application must restore to their previous states any menus that it disabled prior to displaying the dialog box. See the chapter "Dialog Manager" in this book for additional information on dialog boxes.

Writing Your Own Menu Definition Procedure

The Menu Manager uses the menu definition procedure and menu bar definition function to display and perform basic operations on menus and the menu bar. The menu definition procedure performs all the drawing of menu items within a menu and performs all the actions that might differ between one type of menu and another. The menu bar definition function draws the menu bar and performs most of the drawing activities related to the display of menus when the user moves the cursor between menus.

Apple provides a standard menu bar definition function, stored as a resource in the System file. The standard menu bar definition procedure is the `'MBDF'` resource with resource ID 0. When you create your menus and menu bar, by default the Menu Manager uses the standard menu bar definition function to manage them. Although the Menu Manager lets you provide your own menu bar definition function, Apple recommends that you always use the standard menu bar definition function.

The Menu Manager uses the standard menu bar definition function to

- draw the menu bar
- clear the menu bar
- determine if the cursor is in the menu bar or any currently displayed menus
- calculate the left edges of menu titles
- highlight a menu title
- invert the entire menu bar
- erase the background color of a menu and draw the menu's structure (shadow)
- save or restore the bits behind a menu

Apple provides a standard menu definition procedure, stored as a resource in the System file. The standard menu definition procedure is the `'MDEF'` resource with resource ID 0. The standard menu definition procedure handles three types of menus: pull-down, pop-up, and hierarchical; it also implements scrolling in menus. When you define your menus, you specify the menu definition procedure that the Menu Manager should use when managing them. You'll usually want to use the standard definition procedure for your application. However, if you need a feature not provided by the standard menu definition procedure (for example, if you want to include more graphics in your menus), you can write your own menu definition procedure.

The Menu Manager uses the standard menu definition procedure to

- calculate a menu's dimensions
- draw the menu items in a menu
- highlight and unhighlight menu items as the user moves the cursor between them
- determine which item the user chose from a menu

If you provide your own menu definition procedure, it should also perform these tasks. Your menu definition procedure should also support scrolling in menus and color in menus and provide support for Balloon Help.

If you provide your own menu definition procedure, store it in a resource of type `'MDEF'` and include its resource ID in the description of each menu that uses your own menu definition procedure. If you create a menu using `GetMenu` (or `GetNewMBar`), the Menu Manager reads the menu definition procedure into memory and stores a handle to it in the `menuProc` field of the menu's menu record.

When your application uses `GetMenu` (or `GetNewMBar`) to create a new menu that uses your menu definition procedure, the Menu Manager creates a menu record for the menu and fills in the `menuID`, `menuProc`, `enableFlags`, and `menuData` fields according to the menu's resource description. The Menu Manager also reads in the data for each menu item and stores it as variable data at the end of the menu record. The menu definition procedure is responsible for interpreting the contents of the data. For example, the standard menu definition procedure interprets this data as described in "The Menu Resource" beginning on page 3-151. After reading in a resource description of a menu, the Menu Manager requests the menu definition procedure to calculate the size of the menu and to store these values in the `menuWidth` and `menuHeight` fields of the menu's menu record.

Note that when drawing a menu, the Menu Manager first requests your menu definition procedure to calculate the dimensions (the menu rectangle) of the menu. Next the Menu Manager requests the menu bar definition function to draw the structure (shadow) of the menu and erase the contents of the menu to its background color. Then the Menu Manager requests your menu definition procedure to draw the items in the menu. As the user moves the cursor into and out of menu items, the Menu Manager requests your menu definition procedure to highlight and unhighlight items appropriately. Your menu definition procedure should also determine when to add scrolling indicators to a menu and scroll the menu appropriately when the cursor is in a scrolling item. Your menu definition is responsible for showing and removing any help balloons associated with a menu item.

When the Menu Manager requests your menu definition procedure to perform an action on a menu, it provides your procedure with a handle to its menu record. This allows your procedure to access the data in the menu record and to use any data in the variable data portion of the menu record to appropriately handle the menu items. However, your menu definition procedure should not assume that the A5 register is properly set up, so your procedure can't refer to any of the QuickDraw global variables.

The Menu Manager passes a value to your menu definition procedure in the `message` parameter that indicates the action your menu definition procedure should perform. The Menu Manager always passes a handle to the menu record of the menu that the operation should affect in the parameter `theMenu`. Depending on the requested action, the Menu Manager passes additional information in other parameters.

Listing 3-28 shows how you might declare a menu definition procedure.

Listing 3-28 A sample menu definition procedure

```
PROCEDURE MyMDEF (message: Integer; theMenu: MenuHandle;
                  VAR menuRect: Rect; hitPt: Point;
                  VAR whichItem: Integer);

{any support routines used by the main program of your MDEF }
{ go here}

BEGIN
   CASE message OF
      mDrawMsg:
         MyDrawMenu(theMenu, menuRect);
      mChooseMsg:
         MyChooseItem(theMenu, menuRect, hitPt, whichItem);
      mSizeMsg:
         MySizeTheMenu(theMenu);
      mPopUpMsg:
         MyCalcMenuRectForOpenPopUpBox(theMenu, hitPt, menuRect);
   END;
END;
```

The next sections describe in more detail how your menu definition procedure should respond when it receives the `mDrawMsg`, `mChooseMsg`, or `mSizeMsg` constant in the `message` parameter. For a complete description of the menu definition procedure and the parameters passed to your procedure by the Menu Manager, see "The Menu Definition Procedure" beginning on page 3-148.

Calculating the Dimensions of a Menu

Whenever the Menu Manager creates a menu or needs to calculate the size of a menu that is managed by your menu definition procedure, the Menu Manager calls your procedure and specifies the `mSizeMsg` constant in the `message` parameter, requesting that your procedure calculate the size of the menu.

Listing 3-29 on page 3-90 shows an application-defined support routine, `MySizeTheMenu`, used by the application's menu definition procedure. After calculating the height and width of the menu's rectangle, the menu definition procedure stores the values in the `menuWidth` and `menuHeight` fields of the menu's menu record.

Listing 3-29 Calculating the size of a menu

```
PROCEDURE MySizeTheMenu(theMenu: MenuHandle);
VAR
   itemDataPtr:   Ptr;
   numItems:      Integer;
BEGIN
   HLock(Handle(theMenu));
   WITH theMenu^^ DO
   BEGIN    {menuData points to title of menu and additional item data}
      itemDataPtr := @menuData;
      {skip past the menu title}
      itemDataPtr := POINTER(ORD4(itemDataPtr)+ itemDataPtr^ +1);
   END;
   numItems := CountMItems(theMenu);
   {calculate the height of the menu--each item's height can vary }
   { according to whether the item has an icon or a script code defined. }
   { The height of the menu should not exceed the height of the }
   { screen minus the menu bar height. }
   { Store the height in the menu's menu record}
   theMenu^^.menuHeight := MyCalcMenuHeight(itemDataPtr, numItems);

   {calculate the width of the menu (the width of the longest item): }
   { for each item calculate the width as }
   { width = iconWidth + markWidth + textWidth + subMenuWidth }
   {         + cmdKeyComboWidth }
   { If an item doesn't have a characteristic, use 0 as the width of }
   { that characteristic. }
   { To calculate the width of item's text, must consider script code and }
   { width of the font. }
   { The width of the menu should not exceed the right or left }
   { boundaries of the screen. }
   { Store the width in the menu's menu record}
   theMenu^^.menuWidth := MyCalcMenuWidth(itemDataPtr, numItems);
   HUnLock(Handle(theMenu));
END;
```

Drawing Menu Items in a Menu

Whenever the user presses the mouse button while the cursor is in the menu title of a menu managed by your menu definition procedure, the Menu Manager calls the menu bar definition function to highlight the menu title, draw the structure of the menu, and erase the contents of the menu to its background color. The Menu Manager then calls your menu definition procedure and specifies the `mDrawMsg` constant in the `message` parameter, requesting that your procedure draw the menu items. When your menu definition procedure receives this constant, it should draw the menu items of the menu specified by the parameter `theMenu` inside the rectangle specified by the `menuRect` parameter. The Menu Manager sets the current graphics port to the Window Manager port before calling your menu definition procedure. Your menu definition procedure can determine how to draw the menu items by examining the data in the menu record.

If your menu definition procedure supports color menus, your procedure should check the application's menu color information table for the colors to use to draw each item. If the application's menu color information table contains a color entry for an item, draw the item using that color. If the table does not contain an item entry for a particular item, use the default item color defined in the menu title entry. If a menu title entry doesn't exist, use the default item color defined in the menu bar entry. If the menu bar entry doesn't exist, draw the item using black on white.

If your menu definition procedure supports scrolling menus, it should insert scrolling indicators if necessary when drawing the menu items.

Listing 3-30 shows an application-defined support routine, `MyDrawMenu`, used by the application's menu definition procedure. The `MyDrawMenu` procedure draws each item in the menu, according to the item's defined characteristics. Disabled items should be drawn using the colors returned by the `GetGray` function. Pass the RGB color of the item's background in the `bkgnd` parameter to the `GetGray` function; pass the RGB color of the item's enabled text in the `fgnd` parameter. The `GetGray` function returns `TRUE` if there's an available color between the two specified colors and returns in the `fgnd` parameter the color in which you should draw the item.

Listing 3-30 Drawing menu items

```
PROCEDURE MyDrawMenu(theMenu: MenuHandle; menuRect: Rect);
VAR
   numItems:      Integer;
   itemRect:      Rect;
   item:          Integer;
   currentOffset: LongInt;
   nextOffset:    LongInt;
BEGIN
   numItems := CountMItems(theMenu);
   currentOffset := 0;
   nextOffset := 0;
   FOR item := 1 TO numItems DO
   BEGIN
   {calculate the enclosing rectangle for this item}
      itemRect := MyCalcItemRect(item, menuRect, currentOffset, nextOffset);
   {draw the item--index into the item-specific data from the menu record }
   { to get the characteristics of this menu item and draw the item }
   { according to its defined characteristics. For example, draw the item's }
   { text in its defined style & font of its defined script, draw any icon, }
   { mark, submenu indication, or keyboard equivalent, and draw each }
   { characteristic of the item according to its color entry in the menu's }
   { menu color information table. }
   { Draw disabled items in gray--use the GetGray function to return the }
   { appropriate color. Also draw dividers using the gray color }
   { returned by GetGray}
      MyDrawTheItem(item, itemRect, menuRect, currentOffset);
   END;
   {if your menu supports scrolling, insert scrolling indicators if needed}
   MyInsertScrollingArrows(menuRect);
END;
```

Determining Whether the Cursor Is in an Enabled Menu Item

Whenever the user drags the cursor into or out of a menu item of a displayed menu managed by your menu definition procedure, the Menu Manager calls your procedure and specifies the `mChooseMsg` constant in the `message` parameter, requesting that your procedure determine whether the cursor is in a menu item and that your procedure highlight or unhighlight the menu item as appropriate. When your menu definition procedure receives this constant, it should use the menu rectangle specified in the `menuRect` parameter, the mouse location specified in the `hitPt` parameter, and the item number specified in the `whichItem` parameter to determine the proper action to take.

To see whether the user chose an enabled item, your menu definition procedure should determine whether the specified mouse location is inside the rectangle specified by the `menuRect` parameter, and, if so, it should check whether the menu is enabled. If the menu is enabled, your menu definition procedure should determine whether the mouse location specified in the `hitPt` parameter is in an enabled menu item.

If the mouse location is in an enabled menu item, your menu definition procedure should unhighlight the item specified by the `whichItem` parameter, highlight the new item, and return the new item number in `whichItem`.

If the mouse location isn’t in an enabled menu item, your menu definition procedure should unhighlight the item specified by the `whichItem` parameter and return 0 in the `whichItem` parameter.

When your menu definition procedure draws a menu item in its highlighted state in a color menu, it should reverse the background color and the item color and then draw the menu item. When your menu definition procedure needs to return a menu item to its normal (unhighlighted) state, it should reset the background color and item color of that menu item and draw the menu item.

If your menu definition procedure supports scrolling menus, it should scroll the menu when the user moves the cursor into the area of the indicator, or when the cursor is directly above or below the menu. If the user can scroll the menu up (by dragging the cursor past the last item to view more items), place a downward-pointing triangular indicator in place of the last item in the menu. If the user can scroll the menu down (by dragging the cursor past the first item to view the items originally at the top of the menu), place an upward-pointing triangular indicator in place of the first item in the menu.

For all menus, your menu definition procedure should set the global variable `MenuDisable` appropriately each time a new item is highlighted. Set `MenuDisable` to the menu ID and item number of the last menu item chosen, whether or not it's disabled. The `MenuChoice` function uses the value in `MenuDisable` to determine if a chosen menu item is disabled.

Listing 3-31 shows an application-defined support routine, `MyChooseItem`, used by the application's menu definition procedure. This routine determines which item, if any, the point specified by the `hitPt` parameter is in. If the item is in an enabled menu item that is different from the previous item, the `MyChooseItem` procedure unhighlights the old item and highlights the new item. However, the `MyChooseItem` procedure does not highlight the new item if the item is in a divider or disabled item.

The procedure also removes any help balloons as appropriate and, if Balloon Help is turned on, displays any help balloon of the new item (for any item other than a divider or scrolling indicator). The `MyChooseItem` procedure returns the item number of the new item in the `whichItem` parameter or returns 0 if no item is chosen. Although not shown in the listing, if the item is a disabled item, the procedure returns 0 in the `whichItem` parameter and sets the `MenuDisable` global variable to the menu ID and item number of the disabled item.

Listing 3-31 Choosing menu items

```
PROCEDURE MyChooseItem (theMenu: MenuHandle; menuRect: Rect; hitPt: Point;
                        VAR whichItem: Integer);
VAR
   oldWhichItem:        Integer;
   MenuChoicePtr:       ^LongInt;
   numItems, item, max: Integer;
   itemChosen:          Integer;
   inScroll:            Integer;
   currentOffset:       LongInt;
   nextOffset:          LongInt;
```

```
BEGIN
   oldWhichItem := whichItem;
   whichItem := 0;
   itemChosen := 0;
   MenuChoicePtr := POINTER(kLowMemMenuDisable);
   numItems := CountMItems(theMenu);
   {find out whether the hitPt is in an item's rectangle, and if so, }
   { determine which item}
   item := 1;
   max := numItems + 1;
   currentOffset := 0;
   nextOffset := 0;
   REPEAT
      itemRect := MyCalcItemRect(item, menuRect, currentOffset, nextOffset);
      IF PtInRect(hitPt, itemRect) THEN {hitPt is in this item}
         itemChosen := item;
      item := item + 1;
   UNTIL (item = MAX) OR (itemChosen <> 0);
   IF itemChosen = 0 THEN
   BEGIN {the mouse isn't in any item of this menu;unhighlight previous item}
      MyNotInMenu(menuRect, oldWhichItem);
   END
   ELSE
   BEGIN {the mouse is in this menu item. }
         { First see if a previous item was highlighted}
      IF ((oldWhichItem <> 0) AND (oldWhichItem <> itemChosen)) THEN
      BEGIN
         {a previous item was highlighted--unhighlight it}
         itemRect := MyCalcOldItemRect(oldWhichItem, menuRect);
         IF HMGetBalloons THEN {if Balloon Help is on then }
            HMRemoveBalloon;{ remove any balloon that might be showing}
         MyHighlightItem(itemRect, oldWhichItem, FALSE);
      END;
      IF HMGetBalloons and MyIsItemDivider(itemChosen) THEN
         {Balloon Help is on and item is divider}
         HMRemoveBalloon;{remove any balloon that might be showing}
      IF MyIsItemEnabled(itemChosen) THEN
      BEGIN
         {the item is enabled, so highlight the item the cursor is in}
         itemRect := MyCalcNewItemRect(itemChosen, menuRect, currentOffset);
         {the highlighting routine must also support scrolling correctly }
         { (if the cursor is in a scrolling item, don't highlight the item)}
         inScroll := MyIsScrollItem(itemChosen);
```

```
          MyHighlightItem(itemRect, itemChosen, inScroll);
          IF HMGetBalloons AND inScroll THEN
             HMRemoveBalloon   {remove any balloon that might be showing}
          ELSE
          BEGIN {display help balloon for this item, if any}
             IF HMGetBalloons THEN


             BEGIN
                IF StillDown THEN {mouse button is still down in this item}
                   {this routine sets up the needed parameters and then }
                   { calls HMShowMenuBalloon}
                   MyShowMenuBalloon(itemChosen, itemRect);
             END;
          END;
       END;
    END;
END;
```

Menu Manager Reference

This section describes the data structures and routines of the Menu Manager. It also describes various resources, including the resources you can use to create your menus and menu bar, the `'MBAR'` and `'MENU'` resources.

Data Structures

This section describes the menu record, menu list, and menu color information table. The Menu Manager maintains information about the menus in your application in menu records. The Menu Manager maintains information about all the menus in a menu bar in a data structure called the *menu list*.

The Menu Manager stores color information about your application's menus in a menu color information table. You can add entries to your application's menu color information table if you want to use colors other than the default colors for your menu bar or menus. You can add entries to this table by using the `SetMCEntries` procedure or by providing `'mctb'` resources.

The Menu Record

A menu record contains information about a single menu. Your application should never manipulate or access the fields of a menu record; instead your application should use

Menu Manager routines to create and manage the menus in your application. To refer to a menu, use a handle to the menu's menu record.

The `MenuInfo` data type defines the menu record. The `MenuHandle` data type is a handle to a menu record.

```
TYPE  MenuPtr     = ^MenuInfo;   {pointer to a menu record}
      MenuHandle  = ^MenuPtr;    {handle to a menu record}
```

Here is the structure of a menu record:

```
TYPE  MenuInfo =                    {menu record}
      RECORD
         menuID:      Integer;   {number that identifies the menu}
         menuWidth:   Integer;   {width (in pixels) of the menu}
         menuHeight:  Integer;   {height (in pixels) of the menu}
         menuProc:    Handle;    {menu definition procedure}
         enableFlags: LongInt;   {indicates whether menu and }
                                 { menu items are enabled}
         menuData:    Str255;    {title of menu}
         {itemDefinitions}       {variable-length data that }
                                 { defines the menu items}
      END;
```

Field descriptions

`menuID`	A number that identifies the menu. Each menu in your application must have a unique menu ID. Your application specifies the menu ID when you create the menu. Thereafter you can use the menu ID and the `GetMenuHandle` function to get a handle to the menu's menu record. When you define hierarchical menus, you must use a number from 1 through 235 for the menu ID of a submenu of an application; use a number from 236 through 255 for the submenu of a desk accessory.
`menuWidth`	The horizontal dimensions of the menu, in pixels.
`menuHeight`	The vertical dimensions of the menu, in pixels.
`menuProc`	A handle to the menu definition procedure of the menu. The Menu Manager uses this menu definition procedure to draw the menu.
`enableFlags`	A value that represents the enabled state of the menu title and the first 31 items in the menu. All menu items greater than 31 are enabled by default and can be disabled only by disabling the entire menu.
`menuData`	A string that defines the title of the menu. Although the `menuData` field is defined by the data type `Str255` in the `MenuInfo` data structure, the Menu Manager allocates only the storage necessary for the title: the number of characters in the title of the string plus 1.
`itemDefinitions`	Variable-length data that defines the characteristics of each menu item in the menu. If the menu uses the standard menu definition procedure, this data can be conceptually defined in this manner:

```
itemData: ARRAY[1..X] OF
   itemString:  String; {text of menu item}
   itemIcon:    Byte;   {icon number minus 256}
```

```
      itemCmd:     Char;   {keyboard equivalent or }
                           { value ($1B) indicating }
                           { item has a submenu, or }
                           { ($1C) if item has }
                           { a script code, or }
                           { ($1D) if item's 'ICON' }
                           { should be reduced, or }
                           { ($1E) if item has an }
                           { 'SICN' icon}
      itemMark:    Char;   {marking character or }
                           { menu ID of submenu}
      itemStyle:   Style;  {style of menu text}
   endMarker:      Byte;   {contains 0 if no }
                           { more menu items}
```

The menu definition procedure maintains the information about the menu items. You typically define your menu items in `'MENU'` resources, and the Menu Manager stores information describing your items in the menu's menu record.

Your application should not directly change the values of any fields in a menu record. Use Menu Manager routines to change the characteristics of menu items or to make other changes to a menu.

The Menu List

The menu list contains information about the menus in a menu bar, about submenus, and about pop-up menus. A menu list contains handles to the menu records of zero, one, or more menus and contains other information that the Menu Manager uses to manage menus.

The `InitMenus` procedure creates the current menu list of an application. The current menu list contains handles to the menu records of all menus currently in the menu bar and handles to the menu records of any submenus or pop-up menus inserted into the menu list by your application. The menu bar shows the titles, in order, of all menus (other than submenus or pop-up menus) in the menu list.

The initial menu list created by `InitMenus` does not contain handles to any menus. The Menu Manager dynamically allocates storage in a menu list as menus are added to and deleted from the menu list.

Your application should not directly change or access the information in a menu list. You should use Menu Manager routines to create a menu list and to add menus to or remove menus from the current menu list.

You typically define your application's menu bar in an `'MBAR'` resource and create a menu list using the `GetNewMBar` function. The `GetNewMBar` function returns a handle to a menu list. You can set the current menu list to the menu list returned by `GetNewMBar` using the `SetMenuBar` procedure.

The structure of the menu list is private to the Menu Manager. For conceptual purposes, however, its general structure is defined here.

```
TYPE  DynamicMenuList =
      RECORD
         lastMenu:   Integer;    {offset to last pull-down menu}
         lastRight:  Integer;    {pixel location of right edge }
                                 { of rightmost menu in menu bar}
         mbResID:    Integer;    {upper 13 bits are the resource ID of menu }
                                 { bar defn function, low 3 bits the variant}
         menu:       ARRAY[1..X] {variable array with one record for }
                     OF MenuRec; { each menu}
         lastHMenu:  Integer;    {offset to last submenu or pop-up menu}
         menuTitleSave:          {handle to bits behind inverted menu title}
                     pixMapHandle;
         hMenu:      ARRAY[1..Y] {variable array with one record for }
                     OF HMenuRec;{ each submenu or pop-up menu}
      END;
```

The Menu Manager dynamically allocates the records that contain handles to the menu records of menus in the menu bar, submenus, and pop-up menus. These records can be defined conceptually as the `MenuRec` and `HMenuRec` data types. The Menu Manager uses a data structure similar to that of the `MenuRec` data type to store information about pull-down menus in the menu list.

```
TYPE  MenuRec =
      RECORD
         menuOH:     MenuHandle; {handle to menu's menu record}
         menuLeft:   Integer;    {pixel location of left edge }
                                 { of this menu}
      END;
```

The Menu Manager stores information about submenus and pop-up menus at the end of a menu list in a data structure similar to that of the `HMenuRec` data type.

```
TYPE  HMenuRec =
      RECORD
         menuHOH:    MenuHandle; {handle to menu's menu record}
         reserved:   Integer;    {reserved}
      END;
```

The Menu Color Information Table Record

Your application's **menu color information table** defines the standard color for the menu bar, titles of menus, text and characteristics of menu items, and background color of a displayed menu. If you do not add any entries to this table, the Menu Manager draws your menus using the default colors, black on white. You can add colors to your menus by adding entries to your application's menu color information table by using Menu Manager routines or by defining these entries in an 'mctb' resource. Note that the menu color information table uses a format that is different from the standard color table format.

The Menu Manager maintains information about an application's menu color information table as an array of menu color entry records.

```
TYPE  MCTable = ARRAY[0..0] OF MCEntry;         {menu color table}
      MCTablePtr = ^MCTable;      {pointer to a menu color table}
      MCTableHandle = ^MCTablePtr;{handle to a menu color table}
```

A menu color entry is defined by the `MCEntry` data type.

```
TYPE  MCEntry =                   {menu color entry}
      RECORD
         mctID:      Integer;     {menu ID or 0 for menu bar}
         mctItem:    Integer;     {menu item number or 0 for }
                                  { menu title}
         mctRGB1:    RGBColor;    {usage depends on mctID and }
                                  { mctItem}
         mctRGB2:    RGBColor;    {usage depends on mctID and }
                                  { mctItem}
         mctRGB3:    RGBColor;    {usage depends on mctID and }
                                  { mctItem}
         mctRGB4:    RGBColor;    {usage depends on mctID and }
                                  { mctItem}
         mctreserved:Integer;     {reserved}
      END;

      MCEntryPtr = ^MCEntry;      {pointer to a menu color entry}
```

The first two fields of a menu color entry record, `mctID` and `mctItem`, define whether the entry is a menu bar entry, a menu title entry, or a menu item entry. The following four fields specify color information for whatever type of entry the `mctID` and `mctItem` fields describe. The value of the `mctID` field in the last entry in a menu color information table is –99, and the rest of the fields of the last entry are reserved. The Menu Manager automatically creates the last entry in a menu color information table; your application should not use the value –99 as the menu ID of a menu if you wish to add a menu color entry for it.

The Menu Manager creates your application's menu color information table the first time your application calls `InitMenus` or `InitProcMenu`. It creates the menu color information table as initially empty except for the last entry, which indicates the end of the table.

Table 3-7 shows how the Menu Manager interprets the `mctID` and `mctItem` fields for each type of menu color entry in a menu color information table.

Table 3-7 Color information for menu entries

	ID	Item	RGB1	RGB2	RGB3	RGB4
Menu bar	0	0	Default menu title color	Default background color of menus	Default item color	Default bar color
Menu title	*N*<>0	0	Menu title color	Bar color	Default item color	Background color of menu
Menu item	*N*<>0	*M*<>0	Mark color	Item text color	Keyboard equivalent color	Background color of menu
Last entry	–99	Reserved	Reserved	Reserved	Reserved	Reserved

A **menu bar entry** is defined by a menu color entry record that contains 0 in both the `mctID` and `mctItem` fields. You can define only one menu bar entry in a menu color information table. If you don't provide a menu bar entry for your application's menu color information table, the Menu Manager uses the standard menu bar colors (black text on a white background), and it uses the standard colors for the other menu elements. You can provide a menu bar entry to specify default colors for the menu title, the background of a displayed menu, the items in a menu, and the menu bar. The color information fields for a menu bar entry are interpreted as follows:

- `mctRGB1` specifies the default color for menu titles. If a menu doesn't have a menu title entry, the Menu Manager uses the value in this field as the color of the menu title.
- `mctRGB2` specifies the default color for the background of a displayed menu. If a menu doesn't have a menu title entry, the Menu Manager uses the value in this field as the color of the menu's background when it is displayed.
- `mctRGB3` specifies the default color for the items in a displayed menu. If a menu item doesn't have a menu item entry or a default color defined in a menu title entry, the Menu Manager uses the value in this field as the color of the menu item.
- `mctRGB4` specifies the default color for the menu bar. If a menu doesn't have a menu bar entry (and doesn't have any menu title entries), the Menu Manager uses the standard colors for the menu bar.

A **menu title entry** is defined by a menu color entry record that contains a menu ID in the `mctID` field and 0 in the `mctItem` field. You can define only one menu title entry for each menu. If you don't provide a menu title entry for a menu in your application's menu color information table, the Menu Manager uses the colors defined by the menu bar entry. If a menu bar entry doesn't exist, the Menu Manager uses the standard colors (black on white). You can provide a menu title entry to specify a color for the title and background of a specific menu and a default color for its items. The color information fields for a menu title entry are interpreted as follows:

- `mctRGB1` specifies the color for the menu title of the specified menu. If a menu doesn't have a menu title entry, the Menu Manager uses the default value defined in the menu bar entry.
- `mctRGB2` specifies the default color for the menu bar. If a menu color information table doesn't have a menu bar entry, the Menu Manager uses the value in this field as the color of the menu bar. If a menu bar entry already exists, the Menu Manager replaces the value in the `mctRGB2` field of the menu title entry with the value defined in the `mctRGB4` field of the menu bar entry.
- `mctRGB3` specifies the default color for the items in the menu. If a menu item doesn't have a menu item entry or a default color defined in a menu bar entry, the Menu Manager uses the value in this field as the color of the menu item.
- `mctRGB4` specifies the color for the background of the menu.

A **menu item entry** is defined by a menu color entry record that contains a menu ID in the `mctID` field and an item number in the `mctItem` field. You can define only one menu item entry for each menu item. If you don't provide a menu item entry for an item in your application's menu color information table, the Menu Manager uses the colors defined by the menu title entry (or by the menu bar entry if the menu containing the item doesn't have a menu title entry). If neither a menu title entry nor a menu bar entry exists, the Menu Manager draws the mark, text, and keyboard equivalent in black. You can provide a menu item entry to specify a color for the mark, text, and keyboard equivalent of a specific menu item. The color information fields for a menu item entry are interpreted as follows:

- `mctRGB1` specifies the color for the mark of the menu item. If a menu item doesn't have a menu item entry, the Menu Manager uses the default value defined in the menu title entry or the menu bar entry.
- `mctRGB2` specifies the color for the text of the menu item. If a menu item doesn't have a menu item entry, the Menu Manager uses the default value defined in the menu title entry or the menu bar entry. The Menu Manager also draws a black-and-white icon of a menu item using the same color as defined by the `mctRGB2` field. (Use a `'cicn'` resource to provide a menu item with a color icon.)
- `mctRGB3` specifies the color for the keyboard equivalent of the menu item. If a menu item doesn't have a menu item entry, the Menu Manager uses the default value defined in the menu title entry or the menu bar entry.
- `mctRGB4` specifies the color for the background of the menu. If the menu color information table doesn't have a menu title entry for the menu this item is in, or doesn't have a menu bar entry, the Menu Manager uses the value in this field as the background color of the menu. If a menu title entry already exists, the Menu Manager replaces the value in the `mctRGB4` field of the menu item entry with the value defined in the `mctRGB4` field of the menu title entry (or with the `mctRGB2` field of the menu bar entry).

You can use the `GetMCInfo` function to get a copy of your application's menu color information table and the `SetMCEntries` procedure to set entries of your application's menu color information table, or you can provide `'mctb'` resources that define the color entries for your menus.

The `GetMenu`, `GetNewMBar`, and `ClearMenuBar` routines can also modify the entries in the menu color information table. The `GetMenu` function looks for an `'mctb'` resource with a resource ID equal to the value in the `menuID` parameter. If it finds one, it adds the entries to the application's menu color information table.

The `GetNewMBar` function builds a new menu color information table when it creates the new menu list. If you want to save the current menu color information table, call `GetMCInfo` before calling `GetNewMBar`.

The `ClearMenuBar` procedure reinitializes both the current menu list and the menu color information table.

Menu Manager Routines

The Menu Manager includes routines for creating menus, changing the characteristics of menu items, and handling user choice of menu commands. The Menu Manager also provides routines for adding items to and deleting items from menus, counting the number of items in a menu, getting a handle to a menu's menu record, disposing of menus, calculating the dimensions of a menu, highlighting the menu bar, and managing entries in your application's menu color information table.

Some Menu Manager routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, `GetMenuHandle` is also available as `GetMHandle`. Table 3-8 provides a mapping between the previous name of a routine and its new equivalent name.

Table 3-8 Mapping between new and previous names of Menu Manager routines

New name	Previous name
`AppendResMenu`	`AddResMenu`
`DeleteMCEntries`	`DelMCEntries`
`DeleteMenuItem`	`DelMenuItem`
`DisposeMCInfo`	`DispMCInfo`
`GetMenuHandle`	`GetMHandle`
`GetMenuItemText`	`GetItem`
`InsertMenuItem`	`InsMenuItem`
`SetMenuItemText`	`SetItem`

Initializing the Menu Manager

You can use the `InitMenus` procedure to initialize the Menu Manager.

You can use the `InitProcMenu` procedure to set the current menu list so that it uses a custom menu bar definition function if necessary.

InitMenus

The `InitMenus` procedure allocates space for your application's current menu list in your application's heap. Your application needs to call `InitMenus` only once to initialize the Menu Manager and the current menu list for your application.

```
PROCEDURE InitMenus;
```

DESCRIPTION

The `InitMenus` procedure creates the current menu list with no menus, submenus, or pop-up menus. `InitMenus` also creates your application's menu color information table. After allocating the menu color information table, `InitMenus` looks for an `'mctb'` resource with resource ID 0. You can provide an `'mctb'` resource with a resource ID of 0 as one of your application's resources if you want to use colors other than the default colors for your application's menu bar and menus. If `InitMenus` finds and successfully loads an `'mctb'` resource, it adds the information contained in that resource to the menu color information table (using `SetMCEntries`).

The `InitMenus` procedure also draws an empty menu bar.

SPECIAL CONSIDERATIONS

Your application must initalize QuickDraw, the Font Manager, and the Window Manager (using the `InitGraf`, `InitFonts`, and `InitWindows` procedures) before initializing the Menu Manager.

SEE ALSO

To set up the menus for your application's menu bar, use `GetNewMBar` and `SetMenuBar`, described on page 3-111 and page 3-112, respectively. You can also add menus to the current menu list using the `InsertMenu` procedure, described on page 3-108.

To remove all menus from the current menu list, use the `ClearMenuBar` procedure, described on page 3-110.

If your application uses its own menu bar definition function, use the `InitProcMenu` procedure to set the `mbResID` field of the current menu list to the resource ID of your custom `'MBDF'` resource.

See "The Menu Color Information Table Resource" on page 3-155 for a description of the `'mctb'` resource.

See the chapter "Window Manager" in this book for a description of the `InitWindows` procedure. See *Inside Macintosh: Imaging* and *Inside Macintosh: Text* for descriptions of the `InitGraf` and `InitFonts` procedures.

InitProcMenu

Apple recommends that you use the standard menu bar definition function. However, if your application provides its own menu bar definition function, use the `InitProcMenu` procedure to set the `mbResID` field of the current menu list to the resource ID of your custom `'MBDF'` resource.

```
PROCEDURE InitProcMenu (resID: Integer);
```

`resID`	The resource ID of your application's menu bar definition function in the upper 13 bits of this parameter; the variant in the lower 3 bits. You must use a resource ID greater than $100. For resources of type `'MBDF'`, Apple reserves resource IDs $000 through $100 for its own use.

DESCRIPTION

The `InitProcMenu` procedure creates the current menu list if it hasn't already been created by a previous call to `InitMenus`. The `InitProcMenu` procedure stores the resource ID that you specify in the `mbResID` field of the current menu list. The Menu Manager uses the menu bar definition function referred to in this field to draw the menu bar and to perform basic operations on menus.

SPECIAL CONSIDERATIONS

The resource ID of your application's menu bar definition function is maintained in the current menu list until your application next calls `InitMenus`; `InitMenus` initializes the `mbResID` field with the resource ID of the standard menu bar definition function. This can affect applications such as development environments that control other applications that may call `InitMenus`.

SEE ALSO

See the description of the `InitMenus` procedure on page 3-103; you should use `InitMenus` if your application uses the standard menu bar definition function.

Creating Menus

You can use the NewMenu or GetMenu function to create a pull-down menu, although you usually create all the menus in your menu bar at once by providing an 'MBAR' resource and using the GetNewMBar function. See "Getting and Setting the Menu Bar" on page 3-112 for information on creating a menu bar. You typically use the NewMenu or GetMenu function to create submenus or pop-up menus.

The NewMenu function creates a menu with the specified title, assigns it the specified menu ID, and creates a menu record for the menu. Use AppendMenu, InsertMenuItem, AppendResMenu, or InsertResMenu to add items to menus you create with NewMenu.

The GetMenu function creates a menu with the title, items, and characteristics defined in a specified 'MENU' resource.

Both NewMenu and GetMenu allocate space in your application's heap for the menu record and return a handle to the menu's newly created menu record.

To add menus created by NewMenu or GetMenu to the current menu list, use the InsertMenu procedure. To update the menu bar with any new menu titles, use DrawMenuBar.

NewMenu

You can use the NewMenu function to create an empty menu with a specified title and menu ID. In most cases you should store information about your menus (such as their titles, items, and characteristics) in resources; use the GetMenu or GetNewMBar function to create menus from resource definitions.

```
FUNCTION NewMenu (menuID: Integer; menuTitle: Str255): MenuHandle;
```

menuID	The menu ID of the menu. (Note that this is not the resource ID of a 'MENU' resource.) The menu ID is a number that identifies the menu. Use positive menu IDs for menus belonging to your application. Use negative menu IDs for desk accessories (except for submenus of a desk accessory). Submenus must have menu IDs from 1 through 255. For submenus of an application, use menu IDs from 1 through 235; for submenus of a desk accessory, use menu IDs from 236 through 255. Apple reserves the menu ID of 0.
menuTitle	The title of the new menu. Note that in most cases you should store the titles of menus in resources, so that your menu titles can be more easily localized.

DESCRIPTION

The `NewMenu` function creates a menu with the specified title, assigns it the specified menu ID, creates a menu record for the menu, and returns a handle to the menu record. It sets up the menu record to use the standard menu definition procedure (and it reads the standard menu definition procedure into memory if it isn't already there). The `NewMenu` function does not insert the newly created menu into the current menu list.

After creating a menu with `NewMenu`, use `AppendMenu`, `InsertMenuItem`, `AppendResMenu`, or `InsertResMenu` to add menu items to the menu. To add a menu created by `NewMenu` to the current menu list, use the `InsertMenu` procedure. To update the menu bar with any new menu titles, use the `DrawMenuBar` procedure.

SPECIAL CONSIDERATIONS

To release the memory associated with a menu that you created using `NewMenu`, first call `DeleteMenu` to remove the menu from the current menu list and to remove any entries for this menu in your application's menu color information table; then call `DisposeMenu` to dispose of the menu's menu record. After disposing of a menu, use `DrawMenuBar` to update the menu bar.

If the `NewMenu` function is unable to create the menu record, it returns `NIL` as its function result.

SEE ALSO

For information on how to add items to a menu, see the description of `AppendMenu` on page 3-124, `InsertMenuItem` on page 3-126, `AppendResMenu` on page 3-128, and `InsertResMenu` on page 3-129. For information on `InsertMenu`, see page 3-108. To dispose of a menu, see the description of `DeleteMenu` on page 3-109 and `DisposeMenu` on page 3-140.

GetMenu

Use the `GetMenu` function to create a menu with the title, items, and other characteristics defined in a `'MENU'` resource with the specified resource ID. You typically use this function only when you create submenus; you can create all your pull-down menus at once using the `GetNewMBar` function, and you can create pop-up menus using the standard pop-up control definition function.

```
FUNCTION GetMenu (resourceID: Integer): MenuHandle;
```

`resourceID`	The resource ID of the `'MENU'` resource that defines the characteristics of the menu. (You usually use the same number for a menu's resource ID as the number that you specify for the menu ID in the menu resource.)

DESCRIPTION

The GetMenu function creates a menu according to the specified menu resource, and it also creates a menu record for the menu. It reads the menu definition procedure (specified in the menu resource) into memory if it isn't already in memory, and it stores a handle to the menu definition procedure in the menu record. The GetMenu function does not insert the newly created menu into the current menu list.

After reading the 'MENU' resource, the GetMenu function searches for an 'mctb' resource with the same resource ID as the 'MENU' resource. If GetMenu finds this 'mctb' resource, it uses the information in the 'mctb' resource to add entries for this menu to the application's menu color information table. The GetMenu function uses SetMCEntries to add the entries defined by the 'mctb' resource to the application's menu color information table. If GetMenu doesn't find this 'mctb' resource, it uses the default colors specified in the menu bar entry of the application's menu color information, or, if the menu bar entry doesn't exist, it uses the standard colors for the menu.

The GetMenu function returns a handle to the menu record of the menu. You can use the returned menu handle to refer to this menu in most Menu Manager routines. If GetMenu is unable to read the menu or menu definition procedure from the resource file, GetMenu returns NIL.

After creating a menu with GetMenu, you can use AppendMenu, InsertMenuItem, AppendResMenu, or InsertResMenu to add more menu items to the menu if necessary.

To add a menu created by GetMenu to a menu list, use the InsertMenu procedure. To update the menu bar with any new menu titles, use the DrawMenuBar procedure.

Storing the definitions of your menus in resources (especially menu titles and menu items) makes your application easier to localize.

s **WARNING**
Menus in a resource must not be purgeable. s

SPECIAL CONSIDERATIONS

To release the memory associated with a menu that you read from a resource file using GetMenu, first call DeleteMenu to remove the menu from the menu list and to remove any menu title entry or menu item entries for this menu in the application's menu color information table, then call the Resource Manager procedure ReleaseResource to dispose of the menu's menu record. Use DrawMenuBar to update the menu bar.

s **WARNING**
Call GetMenu only once for a particular menu. If you need the handle of a menu currently in the menu list, use GetMenuHandle or the Resource Manager function GetResource. s

SEE ALSO

For a description of the 'MENU' resource, see “The Menu Resource” on page 3-151; for a sample 'MENU' resource in Rez format, see Listing 3-2 on page 3-48. For information on the 'mctb' resource, see “The Menu Color Information Table Resource” on page 3-155.

For details on how to add items to a menu, see the description of AppendMenu on page 3-124, InsertMenuItem on page 3-126, AppendResMenu on page 3-128, and InsertResMenu on page 3-129. To remove a menu, see the description of DeleteMenu on page 3-109. To update the menu bar, use the DrawMenuBar procedure, described on page 3-113.

Adding Menus to and Removing Menus From the Current Menu List

After creating a menu with NewMenu or GetMenu, use the InsertMenu procedure to insert the menu into the current menu list. Use the DeleteMenu procedure to delete a menu from the current menu list; use the ClearMenuBar procedure to remove all menus from the current menu list.

InsertMenu

Use the InsertMenu procedure to insert an existing menu into the current menu list.

```
PROCEDURE InsertMenu (theMenu: MenuHandle; beforeID: Integer);
```

theMenu	A handle to the menu record of the menu. The NewMenu and GetMenu functions return a handle to a menu record that you can use in this parameter.
beforeID	A number that indicates where in the current menu list the menu should be inserted. InsertMenu inserts the menu into the current menu list before the menu whose menu ID equals the number specified in the beforeID parameter. If the number in the beforeID parameter is 0 (or it isn’t the ID of any menu in the menu list), InsertMenu adds the new menu after all others (except before the Help, Keyboard, and Application menus). If the menu is already in the current menu list or the menu list is already full, InsertMenu does nothing. You can specify –1 for the beforeID parameter to insert a submenu into the current menu list. The submenus in the submenu portion of the menu list do not have to be currently associated with a hierarchical menu item; you can store submenus in the menu list and later specify that a menu item has a submenu if needed. However, note that the MenuKey function scans all menus in the menu list for keyboard equivalents, including submenus that are not associated with any menu item. You should not define keyboard equivalents for submenus that are in the current menu list but not associated with a menu item.

You can also specify –1 for the `beforeID` parameter to insert a pop-up menu into the current menu list. However, if you use the standard pop-up control definition function, the pop-up control automatically inserts the menu into the current menu list according to the needs of the pop-up control.

DESCRIPTION

The `InsertMenu` procedure inserts into the current menu list the menu identified by the specified handle to a menu record. To update the menu bar to reflect the new menu, use `DrawMenuBar`.

SEE ALSO

For details on how to update your application's menu bar, see the description of `DrawMenuBar` on page 3-113.

DeleteMenu

Use the `DeleteMenu` procedure to delete an existing menu from the current menu list.

```
PROCEDURE DeleteMenu (menuID: Integer);
```

`menuID`	The menu ID of the menu to delete from the current menu list. If the menu list does not contain a menu with the specified menu ID, `DeleteMenu` does nothing.

DESCRIPTION

The `DeleteMenu` procedure deletes the menu identified by the specified menu ID from the current menu list, and it removes all color entries for that menu from the application's menu color information table. `DeleteMenu` does not release the memory occupied by the menu's menu record. To release the memory occupied by the menu's associated data structures, use `DisposeMenu` if you created the menu using `NewMenu`; use the Resource Manager procedure `ReleaseResource` if you created the menu using `GetMenu` or you read the resource in using `GetNewMBar`.

The `DeleteMenu` procedure first checks the submenu portion of the current menu list for a menu ID with the specified ID. If it finds such a menu, it deletes that menu and returns. If `DeleteMenu` doesn't find the menu in the submenu portion, it checks the regular portion of the current menu list. This allows a desk accessory to delete a submenu without deleting an application's menu whose menu ID might conflict with the menu ID defined by a desk accessory.

After deleting a menu, use `DrawMenuBar` to update the menu bar to reflect the changes to the current menu list.

SEE ALSO

For details on how to dispose of a menu's associated data structures using `DisposeMenu`, see "Disposing of Menus" on page 3-140. For information on the `ReleaseResource` procedure, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

ClearMenuBar

Use the `ClearMenuBar` procedure to delete all menus from the current menu list.

```
PROCEDURE ClearMenuBar;
```

DESCRIPTION

The `ClearMenuBar` procedure deletes all menus from the current menu list and deletes all color entries from the application's menu color information table. `ClearMenuBar` does not release the memory occupied by any of the menus' menu records or the menu color information table. To release the memory occupied by the data structures associated with the menus, use `DisposeMenu` for each menu you created using `NewMenu`; use `ReleaseResource` for each menu you created using `GetMenu` or if you read the resource in using `GetNewMBar`.

After deleting all menus from the current menu list, use `DrawMenuBar` to update the appearance of the menu bar.

SEE ALSO

To update your application's menu bar, see the description of `DrawMenuBar` on page 3-113. For information on the `ReleaseResource` procedure, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Getting a Menu Bar Description From an 'MBAR' Resource

You usually create your application's menu bar by doing the following:

- defining the order and resource ID of your menus in an `'MBAR'` resource
- defining the menus in `'MENU'` resources
- reading in these descriptions using the `GetNewMBar` function
- setting the current menu list to the menu list returned by `GetNewMBar`
- updating the menu bar using `DrawMenuBar`

GetNewMBar

Use the `GetNewMBar` function to read in the definition of a menu bar from an `'MBAR'` resource.

```
FUNCTION GetNewMBar (menuBarID: Integer): Handle;
```

`menuBarID`	The resource ID of an `'MBAR'` resource that specifies the menus for a menu bar.

DESCRIPTION

The `GetNewMBar` function reads in the definition of a menu bar and its associated menus from an `'MBAR'` resource. The `'MBAR'` resource identifies the order of menus contained in its menu bar. For each menu, it also specifies the menu's resource ID. The `GetNewMBar` function reads in each menu from the `'MENU'` resource with the resource ID specified in the `'MBAR'` resource.

The `GetNewMBar` function creates a menu list for the menu bar defined by the `'MBAR'` resource and returns a handle to the menu list. (If the resource isn't already in memory, `GetNewMBar` reads it into memory.) If `GetNewMBar` can't read the resource, `GetNewMBar` returns `NIL`. `GetNewMBar` uses `GetMenu` to read in each individual menu.

After reading in menus from an `'MBAR'` resource, use `SetMenuBar` to make the menu list created by `GetNewMBar` the current menu list. Then use `DrawMenuBar` to update the menu bar.

To release the memory occupied by the data structures associated with the menus in a menu list, use `DisposeMenu` for each menu you created using `NewMenu`; use the Resource Manager procedure `ReleaseResource` for each menu you created using `GetMenu` or if you read the resource in using `GetNewMBar`. To release the memory occupied by a menu list, use the Memory Manager procedure `DisposeHandle`.

SPECIAL CONSIDERATIONS

The `GetNewMBar` function first saves the current menu list and then clears the current menu list and your application's menu color information table. It then creates a new menu list. Before returning a handle to the new menu list, the `GetNewMBar` function restores the current menu list to the previously saved menu list, but `GetNewMBar` does not restore the previous menu color information table. To save and then restore your application's current menu color information table, call the `GetMCInfo` function before `GetNewMBar` and call the `SetMCInfo` procedure afterward.

While you supply only the resource ID of an `'MBAR'` resource to the `GetNewMBar` function, your application often needs to use the menu IDs defined in each of your menus' `'MENU'` resources. Most Menu Manager routines require either a menu ID or a handle to a menu record to perform operations on a specific menu. For menus in the current menu list, you can use the `GetMenuHandle` function to get the handle to a menu record of a menu with a given menu ID.

SEE ALSO

For a description of the `'MENU'` resource, see "The Menu Resource" on page 3-151; for a sample `'MENU'` resource in Rez format, see Listing 3-2 on page 3-48. For a description of the `'MBAR'` resource, see "The Menu Bar Resource" on page 3-155; for a sample `'MBAR'` resource in Rez format, see Listing 3-4 on page 3-49. For information on the `'mctb'` resource, see "The Menu Color Information Table Resource" on page 3-155. For information about the Resource Manager, see *Inside Macintosh: More Macintosh Toolbox*.

Getting and Setting the Menu Bar

You can use the `GetMenuBar` function to get a handle to a copy of the current menu list. Use the `SetMenuBar` procedure to set the current menu bar to a menu list previously returned by `GetMenuBar` or `GetNewMBar`. You can get the height of the menu bar using the `GetMBarHeight` function.

GetMenuBar

Use the `GetMenuBar` function to get a handle to a copy of the current menu list.

```
FUNCTION GetMenuBar: Handle;
```

DESCRIPTION

The `GetMenuBar` function creates a copy of the current menu list and returns a handle to the copy. You can save the returned menu list and then add menus to or remove menus from the current menu list (using `InsertMenu`, `DeleteMenu`, or `ClearMenuBar`). You can later restore the saved menu list using `SetMenuBar`.

To release the memory occupied by a saved menu list, use the Memory Manager's `DisposeHandle` procedure.

s **WARNING**
`GetMenuBar` doesn't copy the menu records, just the menu list (which contains handles to the menu records). Do not dispose of any menus in a saved menu list if you wish to restore the menu list later. s

SetMenuBar

Use the `SetMenuBar` procedure to set the current menu list to a specified menu list.

```
PROCEDURE SetMenuBar (menuList: Handle);
```

`menuList`	A handle to a menu list that specifies the menus for a menu bar. You should specify a handle returned by `GetMenuBar` or `GetNewMBar`.

DESCRIPTION

The `SetMenuBar` procedure copies the given menu list to the current menu list. As with `GetMenuBar`, `SetMenuBar` doesn't copy the menu records, just the menu list (which contains handles to the menu records).

You can use `SetMenuBar` to restore a menu list that you previously saved using `GetMenuBar` or to set the current menu list to a menu list created by `GetNewMBar`.

The `SetMenuBar` procedure sets only the current menu list; to update the menu bar according to the new menu list, use the `DrawMenuBar` procedure.

GetMBarHeight

Use the `GetMBarHeight` function if you need to determine the current height of the menu bar. When the Roman script system is the current system script, the menu bar is 20 pixels high. If a non-Roman script is the current system script, the menu bar may be greater than 20 pixels high to accommodate the current system font.

```
FUNCTION GetMBarHeight: Integer;
```

DESCRIPTION

The `GetMBarHeight` function returns the current height, in pixels, of the menu bar.

Drawing the Menu Bar

Whenever your application adds menus to or removes menus from the current menu list, you should update the titles of the menus in the menu bar using the `DrawMenuBar` procedure. If you change the enabled state of a menu, you should call `DrawMenuBar` to update the menu title accordingly. Alternatively, you can use the `InvalMenuBar` procedure instead of `DrawMenuBar` to invalidate the menu bar; this causes the Event Manager to redraw the menu bar as part of its normal processing of update events.

DrawMenuBar

Use the `DrawMenuBar` procedure to draw the menu bar based on the current menu list.

```
PROCEDURE DrawMenuBar;
```

DESCRIPTION

The `DrawMenuBar` procedure draws (or redraws) the menu bar according to the current menu list. You must call `DrawMenuBar` to update the menu bar after adding menus to or deleting menus from the current menu list using `InsertMenu` or `DeleteMenu`, after setting the current menu list using `SetMenuBar`, after changing the enabled state of a menu, or after any other routine that changes the current menu list.

InvalMenuBar

Use the `InvalMenuBar` procedure to invalidate the menu bar.

```
PROCEDURE InvalMenuBar;
```

DESCRIPTION

The `InvalMenuBar` procedure marks the menu bar as changed and in need of updating. When the Event Manager scans update regions for regions that require updating, the Event Manager also checks to determine whether the menu bar requires updating (because of a call to `InvalMenuBar`). If the menu bar needs updating, the Event Manager calls the `DrawMenuBar` procedure to draw the menu bar.

You can use `InvalMenuBar` instead of `DrawMenuBar` to minimize blinking in the menu bar. For example, if you have several application-defined routines that can change the enabled state of a menu and each calls `DrawMenuBar`, you can replace the calls to `DrawMenuBar` with calls to `InvalMenuBar`. In this way the menu bar is redrawn only once instead of multiple times in quick succession. If you need to make immediate changes to the menu bar, use `DrawMenuBar`. If you want to redraw the menu bar at most once each time through your event loop, use `InvalMenuBar`. The `InvalMenuBar` procedure is available only in System 7.

Responding to the User's Choice of a Menu Command

When the user presses the mouse button while the cursor is in the menu bar, your application should call the `MenuSelect` function to allow the user to choose a command from the menu bar. If the user presses the mouse button while the cursor is over a pop-up menu that does not use the standard pop-up control definition function, your application should call the `PopUpMenuSelect` function to allow the user to make a choice from the pop-up menu.

You should also allow the user to choose a menu command by typing a keyboard equivalent. When the user presses a key on the keyboard, your application should determine if the Command key was pressed at the same time, and, if so, your application should call the `MenuKey` function to map this keyboard combination to any corresponding Command-key equivalent.

If the user chooses an item, both the `MenuSelect` and `MenuKey` functions highlight the title of the menu containing the chosen item and report the user's choice to your application. Your application should perform the corresponding command and, when finished, should unhighlight the menu title using the `HiliteMenu` procedure to indicate to the user that the command is completed.

If the user releases the mouse button while the cursor is over a disabled item or types the keyboard equivalent of a disabled item, `MenuSelect` and `MenuKey` do not report the menu ID or menu item of the item. To determine if the user chose a disabled item (for example, so that your application can provide assistance to the user or explain to the user why the command is disabled), you can use the `MenuChoice` function to return the menu ID and menu item of the disabled menu command.

Your application should adjust its menus before calling `MenuSelect` or `MenuKey`. For example, you should enable or disable menu items as appropriate and add any applicable checkmarks or dashes to items that show attributes.

MenuSelect

Use the `MenuSelect` function to allow the user to choose a menu item from the menus in your application's menu bar.

```
FUNCTION MenuSelect (startPt: Point): LongInt;
```

`startPt`	The point (in global coordinates) representing the location of the cursor at the time the mouse button was pressed.

DESCRIPTION

When the user presses the mouse button while the cursor is in the menu bar, your application receives a mouse-down event. To handle mouse-down events in the menu bar, pass the location of the cursor at the time of the mouse-down event as the `startPt` parameter to `MenuSelect`. The `MenuSelect` function displays and removes menus as the user moves the cursor over menu titles in the menu bar, and it handles all user interaction until the user releases the mouse button.

As the user drags the cursor through the menu bar, the `MenuSelect` function highlights the title of the menu the cursor is currently over and displays all items in that menu. If the user moves the cursor so that it is over a different menu, the `MenuSelect` function removes the previous menu and unhighlights its menu title.

The `MenuSelect` function highlights and unhighlights menu items as the user drags the cursor over the items in a menu. The `MenuSelect` function highlights a menu item if the item is enabled and the cursor is currently over it; it removes such highlighting when the user moves the cursor to another menu item. The `MenuSelect` function does not highlight disabled menu items.

If the user chooses an enabled menu item (including any item from a submenu), the `MenuSelect` function returns a value as its function result that indicates which menu and menu item the user chose. The high-order word of the function result contains the menu ID of the menu, and the low-order word contains the item number of the menu item chosen by the user. The `MenuSelect` function leaves the menu title highlighted; after performing the chosen task your application should unhighlight the menu title using the `HiliteMenu` procedure.

If the user chooses an item from a submenu, `MenuSelect` returns the menu ID of the submenu in the high-order word and the item chosen by the user in the low-order word of its function result. The `MenuSelect` function also highlights the title of the menu in the menu bar that the user originally displayed in order to begin traversing to the submenu. After performing the chosen task, your application should unhighlight the menu title.

If the user releases the mouse button while the cursor is over a disabled item, in the menu bar, or outside of any menu, the `MenuSelect` function returns 0 in the high-order word of its function result and the low-order word is undefined. If it is necessary for your application to find the item number of the disabled item, your application can call `MenuChoice` to return the menu ID and menu item.

If the user chooses an enabled item in a menu that a desk accessory has inserted into your application's menu list, `MenuSelect` uses the `SystemMenu` procedure to process this occurrence and returns 0 to your application in the high-order word.

SPECIAL CONSIDERATIONS

When the `MenuSelect` function pulls down a menu, it stores the bits behind the menu as a relocatable object in the application heap of your application.

ASSEMBLY-LANGUAGE INFORMATION

The `InitMenus` and `InitProcMenu` procedures initialize the `MenuHook` and `MBarHook` global variables to 0. If you choose, you can store the addresses of routines that `MenuSelect` calls in these global variables. The `MenuHook` global variable contains the address (if any) of a routine that `MenuSelect` calls repeatedly while the mouse button is down. `MenuSelect` does not pass any parameters to this routine.

The `MBarHook` global variable contains the address (if any) of a routine that `MenuSelect` calls after a menu title is highlighted and the menu rectangle is calculated but before the menu is drawn. The menu rectangle is the rectangle (in global coordinates) in which the menu will be drawn. `MenuSelect` passes a pointer to the menu rectangle on the stack. If you provide the address of a routine in the `MBarHook` global variable, it should normally return 0 in the D0 register, indicating that `MenuSelect` should continue; returning 1 causes `MenuSelect` to cancel its operation and return immediately to the application.

The `MenuSelect` function uses the global variable `MBarEnable` to determine if all menus in the current menu bar belong to a desk accessory or an application. If the `MBarEnable` global variable is nonzero, then all menus in the current menu bar belong to a desk accessory. If the `MBarEnable` global variable is 0, then all menus in the current menu bar belong to an application. If you're writing a desk accessory, you may need to set the `MBarEnable` global variable to a nonzero value; if you're writing an application, you should not change the value of the `MBarEnable` global variable.

The global variable `TheMenu` contains the ID of the currently highlighted menu in the menu bar. If the user chooses an item from a submenu, `TheMenu` contains the menu ID of the submenu, not the menu to which the submenu is attached.

SEE ALSO

For information on adjusting your application's menus before calling `MenuSelect`, see "Adjusting the Menus of an Application" beginning on page 3-73.

See the description of the `HiliteMenu` procedure on page 3-119 for details on how to unhighlight a menu. For information on how to determine if the user chose a disabled item, see the description of the `MenuChoice` function on page 3-118.

MenuKey

If the user presses another key while holding down the Command key, call the `MenuKey` function to determine if the keyboard combination maps to the keyboard equivalent of a menu item in a menu in the current menu list.

```
FUNCTION MenuKey (ch: Char): LongInt;
```

`ch`	The 1-byte character representing the key pressed by the user in combination with the Command key.

DESCRIPTION

The `MenuKey` function maps the given character to the menu and menu item with that keyboard equivalent. The `MenuKey` function returns as its function result a value that indicates the menu ID and menu item that has the keyboard equivalent corresponding to the given character.

The `MenuKey` function does not distinguish between uppercase and lowercase letters. It takes the 1-byte character passed to it and calls the `UpperText` procedure (which provides localizable uppercase conversion of the character). Thus, `MenuKey` translates any lowercase character to uppercase when comparing a keyboard event to keyboard equivalents. This allows a user to invoke a keyboard equivalent command, such as the Copy command, by pressing the Command key and “c” or “C”. For consistency between applications, you should define the keyboard equivalents of your commands so that they appear in uppercase in your menus.

If the given character maps to an enabled menu item in the current menu list, `MenuKey` highlights the menu title of the chosen menu, returns the menu ID in the high-order word of its function result, and returns the chosen menu item in the low-order word of its function result. After performing the chosen task, your application should unhighlight the menu title using the `HiliteMenu` procedure.

If the given character does not map to an enabled menu item in the current menu list, `MenuKey` returns 0 in its high-order word and the low-order word is undefined.

If the given character maps to a menu item in a menu that a desk accessory has inserted into your application’s menu list, `MenuSelect` uses the `SystemMenu` procedure to process this occurrence and returns 0 to your application in the high-order word.

You should not define menu items with identical keyboard equivalents. The `MenuKey` function scans the menus from right to left and the items from top to bottom. If you have defined more than one menu item with identical keyboard equivalents, `MenuKey` returns the first one it finds.

The MenuKey function first searches the regular portion of the current menu list for a menu item with a keyboard equivalent matching the given key. If it doesn't find one there, it searches the submenu portion of the current menu list. If the given key maps to a menu item in a submenu, MenuKey highlights the menu title in the menu bar that the user would normally pull down to begin traversing to the submenu. Your application should perform the desired command and then unhighlight the menu title.

You shouldn't assign a Command–Shift–number key sequence to a menu item as its keyboard equivalent; Command–Shift–number key sequences are reserved for use as 'FKEY' resources. Command–Shift–number key sequences are not returned to your application, but instead are processed by the Event Manager. The Event Manager invokes the 'FKEY' resource with a resource ID that corresponds to the number that activates it.

Apple reserves the Command-key codes $1B (Control-[ ) through $1F (Control-_) to indicate meanings other than keyboard equivalents. MenuKey ignores these character codes and returns a function result of 0 if you specify any of these values in the ch parameter. Your application should not use these character codes for its own use.

The global variable TheMenu contains the ID of the currently highlighted menu in the menu bar. If the user chooses an item from a submenu, TheMenu contains the menu ID of the submenu, not the menu to which the submenu is attached.

s **WARNING**
Do not define a "circular" hierarchical menu—that is, a hierarchical menu in which a submenu has a submenu whose submenu is a hierarchical menu higher in the chain. If MenuKey detects a circular hierarchical menu, it creates a system error with error number 86. s

SEE ALSO

To unhighlight a menu, use the HiliteMenu procedure, described on page 3-119. To provide support for keyboard equivalents other than Command-key equivalents, see the discussion of 'KCHR' resources in *Inside Macintosh: Text*.

MenuChoice

If your application needs to find the item number of a disabled menu item that the user attempted to choose, you can use the MenuChoice function to return the chosen menu item.

```
FUNCTION MenuChoice: LongInt;
```

DESCRIPTION

If the user chooses a disabled menu item, the MenuChoice function returns a value that indicates which menu and menu item the user chose. The high-order word of the function result contains the menu ID of the menu, and the low-order word contains the item number of the menu item chosen by the user.

The `MenuChoice` function returns 0 as the low-order word of its function result if the mouse button was released while the cursor was in the menu bar or outside the menu.

SPECIAL CONSIDERATIONS

The Menu Manager updates the global variable `MenuDisable` whenever a menu is displayed. As the user moves the cursor over each item, the Menu Manager calls the menu definition procedure of the menu to update the `MenuDisable` global variable to reflect the current menu ID and menu item. The standard menu definition procedure updates the global variable `MenuDisable` appropriately. If your application uses its own menu definition procedure, your menu definition procedure should support this feature; if you use a menu definition procedure that does not update the global variable `MenuDisable` appropriately, the result returned by `MenuChoice` is undefined.

HiliteMenu

You can use the `HiliteMenu` procedure to highlight or unhighlight menu titles. For example, after performing a menu command chosen by the user, use the `HiliteMenu` procedure to unhighlight the menu title.

```
PROCEDURE HiliteMenu (menuID: Integer);
```

`menuID`	The menu ID of the menu whose title should be highlighted. If the menu title of the specified menu is already highlighted, `HiliteMenu` does nothing. If the menu ID is 0 or the specified menu ID isn't in the current menu list, `HiliteMenu` unhighlights whichever menu title is currently highlighted (if any).

DESCRIPTION

The `MenuSelect` and `MenuKey` functions highlight the title of the menu containing the item chosen by the user. After performing the chosen task, your application should unhighlight the menu title by calling `HiliteMenu` and passing 0 in the `menuID` parameter.

The `HiliteMenu` procedure highlights a menu title by first saving the bits behind the title rectangle and then drawing the highlighted title. `HiliteMenu` unhighlights a menu title by restoring the bits behind the menu title.

The global variable `TheMenu` contains the ID of the currently highlighted menu in the menu bar. If the user chooses an item from a submenu, `TheMenu` contains the menu ID of the submenu, not the menu to which the submenu is attached.

SEE ALSO

To highlight the entire menu bar, use the `FlashMenuBar` procedure, described on page 3-141.

PopUpMenuSelect

To display a pop-up menu without using the standard pop-up control definition function, use the `PopUpMenuSelect` function to display the pop-up menu anywhere on the screen. If your application uses the standard pop-up control definition function, your application does not need to use `PopUpMenuSelect`.

```
FUNCTION PopUpMenuSelect (menu: MenuHandle;
                           Top: Integer; Left: Integer;
                           PopUpItem: Integer)
                           : LongInt;
```

`menu`	A handle to the menu record of the menu. The `NewMenu`, `GetMenu`, and `GetMenuHandle` functions return a handle to a specified menu's menu record.
`Top`	The top coordinate of the pop-up box when it is closed. This value should be in global coordinates.
`Left`	The left coordinate of the pop-up box when it is closed. This value should be in global coordinates.
`PopUpItem`	The item number of the current item minus 1. This value should correspond to the user's previous choice from this menu. If the user has not previously made a choice, this value should be set to the default value.

DESCRIPTION

The `PopUpMenuSelect` function uses the location specified by the `Top` and `Left` parameters to determine where to display the specified item of the pop-up menu. The `PopUpMenuSelect` function displays the pop-up menu so that the menu item specified in the `PopUpItem` parameter appears highlighted at the specified location. Figure 3-24 on page 3-34 shows the pop-up title and pop-up box of a pop-up menu.

The `PopUpMenuSelect` function highlights and unhighlights menu items and handles all user interaction until the user releases the mouse button. The `PopUpMenuSelect` function returns the menu ID of the chosen menu in the high-order word of its function result and the chosen menu item in the low-order word.

Your application is responsible for highlighting the pop-up title, setting the mark of the current menu item appropriately, and drawing the text and downward-pointing indicator in the pop-up box before calling `PopUpMenuSelect`. Your application should also make sure the pop-up menu is in the submenu portion of the current menu list before calling `PopUpMenuSelect`. (You can use the `InsertMenu` procedure and specify –1 in the `beforeID` parameter to insert the pop-up menu into the current menu list.)

After calling PopUpMenuSelect, your application can delete the pop-up menu from the current menu list or leave it in the current menu list.

Your application is also responsible for storing the current value of the menu item, drawing the text and downward-pointing indicator in the pop-up box, and unhighlighting the pop-up title after calling PopUpMenuSelect. If you use the standard pop-up control definition function, these actions are performed for you by the pop-up control and your application does not need to call PopUpMenuSelect.

When implementing pop-up menus, you should follow the guidelines for pop-up menus described in *Macintosh Human Interface Guidelines*. For example, you should define the pop-up box of your pop-up menu as a rectangle that is the same height as a menu item, with a one-pixel drop shadow, and should make the pop-up box wide enough to show the currently selected item and a downward-pointing indicator.

SystemMenu

The MenuSelect and MenuKey functions call the SystemMenu procedure when the user chooses an item in a menu that belongs to a desk accessory launched in your application's partition. Your application should not need to call the SystemMenu procedure.

```
PROCEDURE SystemMenu (menuResult: LongInt);
```

menuResult	The value that indicates the menu and menu item chosen by the user. The menu ID is in the high-order word, and the menu item is in the low-order word. The menu ID for a menu belonging to a desk accessory is a negative number.

DESCRIPTION

The SystemMenu procedure directs the desk accessory to perform the appropriate action for the given menu item by calling the desk accessory's control routine and passing the accMenu constant in the csCode parameter. The desk accessory should perform the desired action and return. See *Inside Macintosh: Devices* for more information on desk accessories.

ASSEMBLY-LANGUAGE INFORMATION

If you're writing a desk accessory, you may need to set the MBarEnable global variable to appropriate values. If the MBarEnable global variable is nonzero, then all menus in the current menu bar belong to a desk accessory. If the MBarEnable global variable is 0, then all menus in the current menu bar belong to an application. If you're writing an application, you should not change the value of the MBarEnable global variable.

SystemEdit

When the user chooses one of the standard editing commands in the Edit menu (Undo, Cut, Copy, Paste, and Clear), call the `SystemEdit` function to determine whether the active window belongs to a desk accessory that is launched in your application's partition. If so, the `SystemEdit` function directs the desk accessory to perform the editing command and returns `TRUE`. If the active window does not belong to a desk accessory launched in your application's partition, `SystemEdit` returns `FALSE` and your application should process the command.

```
FUNCTION SystemEdit (editCmd: Integer): Boolean;
```

`editCmd`	The item number of the standard editing command chosen by the user.

Getting a Handle to a Menu Record

Most Menu Manager routines that manage menus require that you specify a handle to the menu record of the menu on which you want to perform an operation. You can use the `HMGetHelpMenuHandle` function to get a handle to your application's Help menu. Use the `GetMenuHandle` function to get a handle to the menu record of any of your application's other pull-down menus or submenus in the current menu list. For pop-up menus that use the standard control definition function, you can access the control record to get the menu's handle.

GetMenuHandle

You can use the `GetMenuHandle` function to get a handle to the menu record of any of your application's menus other than its Help menu. (Use the `HMGetHelpMenuHandle` function to get a handle to the menu record of your application's Help menu.) The `GetMenuHandle` function is also available as the `GetMHandle` function.

```
FUNCTION GetMenuHandle (menuID: Integer): MenuHandle;
```

`menuID`	The menu ID of the menu. (Note that this is not the resource ID, although you often assign the menu ID so that it matches the resource ID.) You assign a menu ID in the `'MENU'` resource of a menu. If you do not define your menus in `'MENU'` resources, you can assign a menu ID using `NewMenu`.

DESCRIPTION

The `GetMenuHandle` function returns a handle to the menu record of the menu having the specified menu ID. If the menu is in the current menu list, `GetMenuHandle` returns a handle to the menu record of the menu as its function result. Otherwise, `GetMenuHandle` returns `NIL` as its function result.

SPECIAL CONSIDERATIONS

To get a handle to a menu record of a pop-up menu that you create using the pop-up control definition function, dereference the `cntrlData` field of the pop-up menu's control record instead of using `GetMenuHandle`.

HMGetHelpMenuHandle

Use the `HMGetHelpMenuHandle` function to get a handle to the menu record of your application's Help menu.

```
FUNCTION HMGetHelpMenuHandle (VAR mh: MenuHandle): OSErr;
```

`mh`	The `HMGetHelpMenuHandle` function returns a copy of a handle to your application's Help menu in this parameter.

DESCRIPTION

The `HMGetHelpMenuHandle` function returns in the `mh` parameter a copy of a handle to the menu record of your application's Help menu. With this handle, you can append items to your application's Help menu by using the `AppendMenu` procedure or other related Menu Manager routines. The Help Manager automatically adds the divider that separates your items from the rest of the Help menu items.

Be sure to define help balloons for your items in the Help menu by creating an `'hmnu'` resource and specifying the `kHMHelpMenuID` constant as its resource ID.

The Menu Manager functions `MenuSelect` and `MenuKey` return a result with the menu ID in the high-order word and the menu item in the low-order word. The `MenuSelect` function (and the `MenuKey` function, if the user chooses an item with a keyboard equivalent) returns the `kHMHelpMenuID` constant in the high-order word when the user chooses an appended item from the Help menu. The menu item number of the appended menu item is returned in the low-order word of the function result. Apple reserves the right to change the number of standard items in the Help menu. To determine the number of items in the Help menu, call the `CountMItems` function.

SPECIAL CONSIDERATIONS

Do not use the `GetMenuHandle` function to get a handle to the menu record of the Help menu. `GetMenuHandle` returns a handle to the menu record of the global Help menu, not the menu record of the Help menu that is specific to your application.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough room in heap zone
`resNotFound`	–192	Unable to read resource
`hmHelpManagerNotInited`	–855	Help menu not set up

SEE ALSO

For examples of how to add items to your application's Help menu and how to handle the user's choice of an item in the Help menu, see Listing 3-14 on page 3-68 and Listing 3-26 on page 3-81. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on creating help balloons for the menus of your application.

Adding and Deleting Menu Items

You can add the names of all resources of a specified type to a menu using the `InsertResMenu` or `AppendResMenu` procedure. You can add menu items that you define to a menu using the `AppendMenu` or `InsertMenuItem` procedure. You can also delete menu items using the `DeleteMenuItem` procedure. In most cases you should not insert or delete individual menu items from an already existing menu unless the user expects a menu (such as a list of currently open documents) to change.

If you add menu items using the `AppendMenu` or `InsertMenuItem` procedure, you should define in resources the text and other characteristics of the menu items that you add. This makes your application easier to localize for other regions.

AppendMenu

Use the `AppendMenu` procedure to append one or more items to a menu previously created using `NewMenu`, `GetMenu`, or `GetNewMBar`.

```
PROCEDURE AppendMenu (menu: MenuHandle; data: Str255);
```

`menu`	A handle to the menu record of the menu to which you wish to append the menu item or items.
`data`	A string that defines the characteristics of the new menu item or items. Note that in most cases you should store the text of a menu item in a resource, so that your menu items can be more easily localized. The `AppendMenu` procedure appends the menu items in the order in which they are listed in the `data` parameter.

DESCRIPTION

The `AppendMenu` procedure appends any defined menu items to the specified menu. The menu items are added to the end of the menu. You specify the text of any menu items and their characteristics in the `data` parameter. You can embed metacharacters in the string to define various characteristics of a menu item.

Here are the metacharacters that you can specify in the `data` parameter:

Metacharacter	Description
`;` or Return	Separates menu items.
`^`	When followed by an icon number, defines the icon for the item. If the keyboard equivalent field contains $1C, this number is interpreted as a script code.
`!`	When followed by a character, defines the mark for the item. If the keyboard equivalent field contains $1B, this value is interpreted as the menu ID of a submenu of this menu item.
`<`	When followed by one or more of the characters B, I, U, O, and S, defines the character style of the item to Bold, Italic, Underline, Outline, or Shadow, respectively.
`/`	When followed by a character, defines the keyboard equivalent for the item. When followed by $1B, specifies that this menu item has a submenu. To specify that the menu item has a script code, small icon, or reduced icon, use the `SetItemCmd` procedure to set the keyboard equivalent field to $1C, $1D, or $1E, respectively.
`(`	Defines the menu item as disabled.

You can specify any, all, or none of these metacharacters in the text string. The metacharacters that you specify aren't displayed in the menu item. (To use any of these metacharacters in the text of a menu item, first use `AppendMenu`, specifying at least one character as the item's text, and then use the `SetMenuItemText` procedure to set the item's text to the desired string.)

Note

If you add menu items using the `AppendMenu` procedure, you should define the text and any marks or keyboard equivalents in resources for easier localization. u

You can specify the first character that defines the text of a menu item as a hyphen to create a divider line. The string in the `data` parameter can be blank (containing one or more spaces), but it should not be an empty string.

If you do not define a specific characteristic of a menu item, the `AppendMenu` procedure assigns the default characteristic to the menu item. If you do not define any characteristic other than the text for a menu item, the `AppendMenu` procedure inserts the menu item so that it appears in the menu as an enabled item, without an icon or a mark, in the plain character style, and without a keyboard equivalent.

You can use `AppendMenu` to append items to a menu regardless of whether the menu is in the current menu list.

SEE ALSO

See "Adding Items to a Menu" on page 3-64 for examples of appending items to a menu.

InsertMenuItem

Use the `InsertMenuItem` procedure to insert one or more items to a menu previously created using `NewMenu`, `GetMenu`, or `GetNewMBar`.

The `InsertMenuItem` procedure is also available as the `InsMenuItem` procedure.

```
PROCEDURE InsertMenuItem (theMenu: MenuHandle; itemString: Str255;
                              afterItem: Integer);
```

`theMenu` A handle to the menu record of the menu to which you wish to add the menu item or items.

`itemString`
A string that defines the characteristics of the new menu items. Note that in most cases you should store the text of a menu item in a resource, so that your menu items can be more easily localized. You can specify the contents of the `itemString` parameter using metacharacters; the `InsertMenuItem` procedure accepts the same metacharacters as the `AppendMenu` procedure. However, if you specify multiple items, the `InsertMenuItem` procedure inserts the items in the reverse of their order in the `itemString` parameter.

`afterItem` The item number of the menu item after which the new menu items are to be added. Specify 0 in the `afterItem` parameter to insert the new items before the first menu item; specify the item number of a current menu item to insert the new menu items after it; specify a number greater than or equal to the last item in the menu to append the new items to the end of the menu.

DESCRIPTION

The `InsertMenuItem` procedure inserts any defined menu items to the specified menu. The menu items are inserted according to the location specified by the `afterItem` parameter. You specify the text of any menu items and their characteristics in the `itemString` parameter. You can embed metacharacters in the string you specify to define various characteristics of a menu item. The metacharacters aren't displayed in the menu.

Here are the metacharacters you can specify in the `itemString` parameter:

Metacharacter	Description
`;` or Return	Separates menu items.
^	When followed by an icon number, defines the icon for the item. If the keyboard equivalent field contains $1C, this number is interpreted as a script code.
!	When followed by a character, defines the mark for the item. If the keyboard equivalent field contains $1B, this value is interpreted as the menu ID of a submenu of this menu item.

Metacharacter	Description
`<`	When followed by one or more of the characters B, I, U, O, and S, defines the character style of the item to Bold, Italic, Underline, Outline, or Shadow, respectively.
`/`	When followed by a character, defines the keyboard equivalent for the item. When followed by $1B, specifies that this menu item has a submenu. To specify that the menu item has a script code, small icon, or reduced icon, use the `SetItemCmd` procedure to set the keyboard equivalent field to $1C, $1D, or $1E, respectively.
`(`	Defines the menu item as disabled.

You can specify any, all, or none of these metacharacters in the text string. The metacharacters that you specify aren't displayed in the menu item. To use any of these metacharacters in the text of a menu item, first use `InsertMenuItem`, specifying at least one character as the item's text, and then use the `SetMenuItemText` procedure to set the item's text to the desired string.

Note

If you add menu items using the `InsertMenuItem` procedure, you should define the text and any marks or keyboard equivalents in resources for easier localization. u

You can specify the first character that defines the text of a menu item as a hyphen to create a divider line. The string in the `itemString` parameter can be blank (containing one or more spaces), but it should not be an empty string.

If you do not define a specific characteristic of a menu item, the `InsertMenuItem` procedure assigns the default characteristic to the menu item. If you do not define any characteristic other than the text for a menu item, the `InsertMenuItem` procedure inserts the menu item so that it appears in the menu as an enabled item, without an icon or a mark, in the plain character style, and without a keyboard equivalent.

You can use `InsertMenuItem` to insert items into a menu regardless of whether the menu is in the current menu list.

SEE ALSO

See "Adding Items to a Menu" beginning on page 3-64 for examples.

DeleteMenuItem

Use the `DeleteMenuItem` procedure to delete an item from a menu. The `DeleteMenuItem` procedure is also available as the `DelMenuItem` procedure.

```
PROCEDURE DeleteMenuItem (theMenu: MenuHandle; item: Integer);
```

`theMenu`	A handle to the menu record of the menu from which you want to delete the menu item.
`item`	The item number of the menu item to delete. If you specify 0 or a number greater than the last item in the menu, `DeleteMenuItem` does not delete any item from the menu.

DESCRIPTION

The `DeleteMenuItem` procedure deletes a specified menu item from a menu. The `DeleteMenuItem` procedure also deletes the item's menu item entry from your application's menu color information table (if an entry exists). You should not delete items from an existing menu unless the user expects the menu (such as a menu that lists open documents) to change.

AppendResMenu

Use the `AppendResMenu` procedure to search all resource files open to your application for a given resource type and to append the names of any resources it finds to a specified menu. The specified menu must have been previously created using `NewMenu`, `GetMenu`, or `GetNewMBar`.

The `AppendResMenu` procedure is also available as the `AddResMenu` procedure.

```
PROCEDURE AppendResMenu (theMenu: MenuHandle; theType: ResType);
```

`theMenu`	A handle to the menu record of the menu to which to append the names of any resources of a given type that `AppendResMenu` finds.
`theType`	A four-character code that identifies the resource type for which to search.

DESCRIPTION

The `AppendResMenu` procedure searches all resource files open to your application for resources of the type defined by the parameter `theType`. It appends the names of any resources it finds of the given type to the end of the specified menu. `AppendResMenu` appends the names of found resources in alphabetical order; it does not alphabetize items already in the menu. The `AppendResMenu` procedure does not add resources with names that begin with a period (`.`) or a percent sign (%) to the menu.

The `AppendResMenu` procedure assigns default characteristics to each menu item. Each appended menu item appears in the menu as an enabled item, without an icon or a mark, in the plain character style, and without a keyboard equivalent. To get the name or to change other characteristics of an item appended by `AppendResMenu`, use the Menu Manager routines described in "Getting and Setting the Appearance of Menu Items" beginning on page 3-130.

If you specify that AppendResMenu add resources of type 'DRVR' to your Apple menu, AppendResMenu adds the name (and icon) of each item in the Apple Menu Items folder to the menu.

If you specify that AppendResMenu append resources of type 'FONT' or 'FOND', the Menu Manager performs special processing for any resources it finds that have font numbers greater than $4000. If the script system associated with the font name is installed in the system, AppendResMenu stores information in the itemDefinitions array (in the itemIcon and itemCmd fields for that item) in the menu's menu record. This allows the Menu Manager to display the font name in the correct script.

SPECIAL CONSIDERATIONS

The AppendResMenu procedure calls the Resource Manager procedure SetResLoad (specifying TRUE in the load parameter) before returning. The AppendResMenu procedure reads the resource data of the resources it finds into memory. If your application does not want the Resource Manager to read resource data into memory when your application calls other routines that read resources, you need to call SetResLoad and specify FALSE in the load parameter after AppendResMenu returns.

SEE ALSO

Listing 3-15 on page 3-69 shows a sample that adds items from the Apple Menu Items folder to the Apple menu, and Listing 3-16 on page 3-70 shows a sample that adds font names to a menu. See *Inside Macintosh: More Macintosh Toolbox* for information on the Resource Manager.

InsertResMenu

Use the InsertResMenu procedure to search all resource files open to your application for a given resource type and to insert the names of any resources it finds to a specified menu. The items are inserted after the specified menu item. The specified menu must have been previously created using NewMenu, GetMenu, or GetNewMBar.

```
PROCEDURE InsertResMenu (theMenu: MenuHandle; theType: ResType;
                         afterItem: Integer);
```

theMenu	A handle to the menu record of the menu to which to add the names of any resources of a given type that InsertResMenu finds.
theType	A four-character code that identifies the resource type for which to search.
afterItem	A number that indicates where in the menu to insert the names of any resources of the given type that InsertResMenu finds. Specify 0 in the afterItem parameter to insert the items before the first menu item; specify the item number of a menu item already in the menu to insert the items after the specified item number. If you specify a number greater than or equal to the last item in the menu, the items are inserted at the end of the menu.

DESCRIPTION

The InsertResMenu procedure searches all resource files open to your application for resources of the type defined by the parameter theType. It inserts the names of any resources it finds of the given type at the specified location in the specified menu. InsertResMenu adds the names of found resources in alphabetical order; it does not alphabetize items already in the menu.

The InsertResMenu procedure does not add resources with names that begin with a period (.) or a percent sign (%) to the menu.

The InsertResMenu procedure assigns default characteristics to each menu item. Each appended menu item appears in the menu as an enabled item, without an icon or a mark, in the plain character style, and without a keyboard equivalent. To get the name or to change other characteristics of an item appended by InsertResMenu, use the Menu Manager routines described in the next section, "Getting and Setting the Appearance of Menu Items."

If you specify that InsertResMenu add resources of type 'DRVR' to your Apple menu, InsertResMenu adds the name (and icon) of each item in the Apple Menu Items folder to the menu.

If you specify that InsertResMenu add resources of type 'FONT' or 'FOND', the Menu Manager performs special processing for any resources it finds that have font numbers greater than $4000. If the script associated with the font name is currently active, InsertResMenu stores information in the itemDefinitions array (in the itemIcon and itemCmd fields for that item) in the menu's menu record that allows the Menu Manager to display the font name in the correct script.

SPECIAL CONSIDERATIONS

The InsertResMenu procedure calls the Resource Manager procedure SetResLoad (specifying TRUE in the load parameter) before returning. The InsertResMenu procedure reads the resource data of the resources it finds into memory. If your application does not want the Resource Manager to read resource data into memory when your application calls other routines that read resources, you need to call SetResLoad and specify FALSE in the load parameter after InsertResMenu returns.

Getting and Setting the Appearance of Menu Items

You can get information about the characteristics of a menu item using Menu Manager routines. For example, you can get an item's text, style, mark, keyboard equivalent, script code, and associated icons. You can also determine if a menu item has a submenu associated with it and the menu ID of the submenu.

You can set the characteristics of a menu item, including associating a submenu with a menu item, using Menu Manager routines. Whenever possible, however, you should define your application's menu items in 'MENU' resources. This makes your application easier to localize for other regions.

You can also enable and disable menu items or entire menus using Menu Manager routines.

EnableItem

Use the `EnableItem` procedure to enable a menu item or a menu.

```
PROCEDURE EnableItem (theMenu: MenuHandle; item: Integer);
```

`theMenu`	A handle to the menu record of the menu containing the menu item to enable.
`item`	The item number of the menu item to enable, or 0 to enable the entire menu. You cannot individually enable a menu item with an item number greater than 31. If you specify 0 in the `item` parameter, the `EnableItem` procedure enables the menu title and all items in the menu that were not previously individually disabled.

DESCRIPTION

The `EnableItem` procedure enables a specified menu item so that it no longer appears dim and so that the user can choose the menu item.

Note that, if you enable a menu, the `EnableItem` procedure enables the menu title but only enables those menu items that are not currently disabled as a result of your application previously calling `DisableItem` and specifying each item's item number. For example, if all items in your application's Edit menu are enabled, you can disable the Cut and Copy commands individually using `DisableItem`. If you choose to disable the entire menu by passing 0 as the `item` parameter to `DisableItem`, the menu and all its items are disabled. If you then enable the entire menu by passing 0 as the `item` parameter to `EnableItem`, the menu and its items are enabled, except for the Cut and Copy commands, which remain disabled. In this case, to enable the Cut and Copy commands you must enable each one individually using `EnableItem`.

If your application enables a menu using `EnableItem`, it should call `DrawMenuBar` to update the menu bar's appearance.

SEE ALSO

See "Enabling and Disabling Menu Items" on page 3-58 for examples of enabling items in a menu.

DisableItem

Use the `DisableItem` procedure to disable a menu item or an entire menu.

```
PROCEDURE DisableItem (theMenu: MenuHandle; item: Integer);
```

`theMenu`	A handle to the menu record of the menu containing the menu item to disable.
`item`	The item number of the menu item to disable, or 0 to disable the entire menu. You cannot individually disable a menu item with an item number greater than 31. If you specify 0 in the `item` parameter, the `DisableItem` procedure disables the menu title and all items in the menu, including menu items with item numbers greater than 31.

DESCRIPTION

The `DisableItem` procedure disables a specified menu item so that it appears dim and cannot be chosen by the user.

If your application disables a menu using `DisableItem`, your application should call `DrawMenuBar` to update the menu bar's appearance.

SEE ALSO

See "Enabling and Disabling Menu Items" on page 3-58 for examples of disabling items in a menu.

GetMenuItemText

Use the `GetMenuItemText` procedure to get the text of a specific menu item. The `GetMenuItemText` procedure is also available as the `GetItem` procedure.

```
PROCEDURE GetMenuItemText (theMenu: MenuHandle; item: Integer;
                           VAR itemString: Str255);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose text you wish to get.
`item`	The item number of the menu item. The `GetMenuItemText` procedure returns the text of this item.
`itemString`	The `GetMenuItemText` procedure returns the text of the menu item in this parameter.

DESCRIPTION

The `GetMenuItemText` procedure returns the text of the specified menu item in the `itemString` parameter. Use other Menu Manager routines to get information about the other characteristics of a menu item.

SetMenuItemText

Use the `SetMenuItemText` procedure to set the text of a specific menu item to a given string. The `SetMenuItemText` procedure is also available as the `SetItem` procedure.

```
PROCEDURE SetMenuItemText (theMenu: MenuHandle; item: Integer;
                           itemString: Str255);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose text you wish you to set.
`item`	The item number of the menu item. The `SetMenuItemText` procedure sets the text of this item.
`itemString`	The `SetMenuItemText` procedure sets the text of the menu item according to the string specified in the `itemString` parameter. The `SetMenuItemText` procedure does not recognize metacharacters or set any other characteristics of the menu item. The `itemString` parameter can be blank, but it should not be an empty string.

DESCRIPTION

The `SetMenuItemText` procedure sets the text of the specified menu item to the text specified in the `itemString` parameter. The `SetMenuItemText` procedure does not recognize any metacharacters used by the `AppendMenu` and `InsertMenuItem` procedures. Use other Menu Manager routines to set other characteristics of a menu item.

If you set the text of a menu item using the `SetMenuItemText` procedure, you should store the text in a string resource so that your application can be more easily localized.

SEE ALSO

See Listing 3-9 on page 3-59 for an example of setting the text of a menu item.

GetItemStyle

Use the `GetItemStyle` procedure to get the style of the text in a specific menu item.

```
PROCEDURE GetItemStyle (theMenu: MenuHandle; item: Integer;
                        VAR chStyle: Style);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose style you wish to get.
`item`	The item number of the menu item. The `GetItemStyle` procedure returns the style of the text for this item.

`chStyle` The `GetItemStyle` procedure returns the style of the text for this item in the `chStyle` parameter. The `chStyle` parameter is a set defined by the `Style` data type.

```
TYPE
StyleItem = (bold, italic, underline, outline,
             shadow, condense, extend);
Style     = SET OF StyleItem;
```

DESCRIPTION

The `GetItemStyle` procedure returns the style of the text of the specified menu item in the `chStyle` parameter. The returned style can be one or more of the styles defined by the `Style` data type, or it is the empty set if the style of the text is Plain.

SetItemStyle

Use the `SetItemStyle` procedure to set the style of the text in a specific menu item.

```
PROCEDURE SetItemStyle (theMenu: MenuHandle; item: Integer;
                        chStyle: Style);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose style you wish to set.
`item`	The item number of the menu item. The `SetItemStyle` procedure sets the style of the text for this item.
`chStyle`	The `SetItemStyle` procedure sets the style of the text for this item according to the style described by the `chStyle` parameter. The `chStyle` parameter is a set defined by the `Style` data type.

```
TYPE
StyleItem = (bold, italic, underline, outline,
             shadow, condense, extend);
Style     = SET OF StyleItem;
```

You can set the style to one or more of the styles defined by the `Style` data type, or you can set it to Plain by specifying an empty set in the `chStyle` parameter.

DESCRIPTION

The `SetItemStyle` procedure sets the style of the text of the specified menu item to the style or styles defined by the `chStyle` parameter.

SEE ALSO

See Listing 3-10 on page 3-60 for examples of setting the style of a menu item.

GetItemMark

Use the `GetItemMark` procedure to get the mark of a specific menu item or the menu ID of the submenu associated with the menu item.

```
PROCEDURE GetItemMark (theMenu: MenuHandle; item: Integer;
                        VAR markChar: Char);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose mark or submenu you wish to get.
`item`	The item number of the menu item. The `GetItemMark` procedure returns the mark of this item or, if this item has a submenu associated with it, returns the menu ID of the submenu in the `markChar` parameter.
`markChar`	The `GetItemMark` procedure returns the mark or the submenu of this item in the `markChar` parameter. A menu item can have a mark or a submenu attached to it, but not both. If this menu item has a marking character, the `GetItemMark` procedure returns the mark. If this menu item has a submenu associated with it, the `GetItemMark` procedure returns the menu ID of the submenu. If the item doesn't have a mark or a submenu, `GetItemMark` returns 0 in this parameter.

DESCRIPTION

If the item has a mark or submenu, the `GetItemMark` procedure returns the mark or the menu ID of the submenu of the specified menu item in the `markChar` parameter (or 0 if the item doesn't have a mark or a submenu).

SetItemMark

Use the `SetItemMark` procedure to set the mark of a specific menu item or to change or set the submenu associated with a menu item.

```
PROCEDURE SetItemMark (theMenu: MenuHandle; item: Integer;
                        markChar: Char);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose mark or submenu you wish to set.
`item`	The item number of the menu item. The `SetItemMark` procedure sets the mark or the submenu of this item.
`markChar`	The `SetItemMark` procedure sets the mark or submenu of this item according to the information in the `markChar` parameter.

To set the mark of a menu item, specify the marking character in the `markChar` parameter. You can also use one of these constants to specify that the item has no mark, has a checkmark as the marking character, or has the diamond symbol as the marking character:

```
CONST
noMark       = 0;    {no marking character}
checkMark    = $12;  {checkmark}
diamondMark  = $13;  {diamond symbol}
```

To set the submenu associated with this menu item, specify the menu ID of the submenu in the `markChar` parameter.

DESCRIPTION

The `SetItemMark` procedure sets the mark or the submenu of the specified menu item.

SEE ALSO

See Listing 3-11 on page 3-61 for examples of setting the mark of a menu item.

CheckItem

Use the `CheckItem` procedure to set the mark of a specific menu item to a checkmark or to remove a mark from a menu item.

```
PROCEDURE CheckItem (theMenu: MenuHandle; item: Integer;
                     checked: Boolean);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose mark you wish to set to a checkmark or whose mark you wish to remove.
`item`	The item number of the menu item.
`checked`	The `CheckItem` procedure sets or removes the mark of the item according to the information in the `checked` parameter. To set the mark of a menu item to a checkmark, specify `TRUE` in the `checked` parameter. To remove a checkmark or any other mark from a menu item, specify `FALSE` in the `checked` parameter.

DESCRIPTION

The `CheckItem` procedure sets the mark of the specified menu item to a checkmark or removes any mark from the menu item.

SEE ALSO

See Listing 3-11 on page 3-61 for examples of setting the mark of a menu item.

GetItemIcon

Use the `GetItemIcon` procedure to get the icon or script code of a specific menu item. If the menu item's keyboard equivalent field contains $1C, the returned number represents the script code of the menu item. Otherwise, the returned number represents the item's icon number.

```
PROCEDURE GetItemIcon (theMenu: MenuHandle; item: Integer;
                        VAR iconIndex: Byte);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose icon or script code you wish to get.
`item`	The item number of the menu item. The `GetItemIcon` procedure returns the icon number or script code of this item.
`iconIndex`	For menu items that do not specify $1C in the keyboard equivalent field, the `GetItemIcon` procedure returns the icon number of the item's icon in this parameter. The icon number returned in this parameter is a value from 1 through 255 if the menu item has an icon associated with it and is 0 otherwise. You can add 256 to the icon number to generate the resource ID of the `'cicn'`, `'ICON'`, or `'SICN'` resource that describes the icon of the menu item. For example, if the `GetItemIcon` procedure returns 5 in this parameter, then the icon of the menu item is described by an icon resource with resource ID 261. For menu items that contain $1C in the keyboard equivalent field, the `GetItemIcon` procedure returns the script code of the menu item. The Menu Manager displays the menu item using this script code if the corresponding script system is installed.

DESCRIPTION

The `GetItemIcon` procedure returns the icon number or script code of the specified menu item in the `iconIndex` parameter (or 0 if the item doesn't have an icon or a script code).

SetItemIcon

Use the `SetItemIcon` procedure to set the icon number or script code of a specific menu item. Usually you display menu items in the current system script; however, if needed, you can use the `SetItemIcon` procedure to set the script code of a menu item. For an item's script code to be set, the keyboard equivalent field of the item must contain $1C. If the keyboard equivalent field contains any other value, the `SetItemIcon` procedure interprets the specified number as the item's icon number.

```
PROCEDURE SetItemIcon (theMenu: MenuHandle; item: Integer;
                        iconIndex: Byte);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose icon (or script code) you wish to set.
`item`	The item number of the menu item. The `SetItemIcon` procedure sets the icon (or script code) of this item.
`iconIndex`	If the menu item's keyboard equivalent field does not contain $1C, the `SetItemIcon` procedure sets the icon number of the item's icon to the number defined in this parameter. The icon number you specify should be a value from 1 through 255 (or from 1 through 254 if the item has a small or reduced icon) or 0 if the item does not have an icon.

The Menu Manager adds 256 to the icon number to generate the resource ID of the `'cicn'` or `'ICON'` resource that describes the icon of the menu item. For example, if you specify 5 as the value of the `iconIndex` parameter, when the Menu Manager needs to draw the item, it looks for an icon resource with resource ID 261.

If the menu item's keyboard equivalent field contains $1C, the `SetItemIcon` procedure sets the script code of the menu item to the number defined in the `iconIndex` parameter. The Menu Manager displays the menu item using the specified script code if the corresponding script system is installed.

You can specify 0 in the `iconIndex` parameter to indicate that the item uses the current system script and does not have an icon number.

DESCRIPTION

The `SetItemIcon` procedure sets the icon number or script code of the specified menu item to the value in the `iconIndex` parameter.

SEE ALSO

See "Changing the Icon or Script Code of Menu Items" beginning on page 3-62 for examples of setting the icon of a menu item.

GetItemCmd

Use the `GetItemCmd` procedure to get the value of the keyboard equivalent field of a menu item.

```
PROCEDURE GetItemCmd (theMenu: MenuHandle; item: Integer;
                      VAR cmdChar: Char);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose keyboard equivalent field you wish to get.
`item`	The item number of the menu item. The `GetItemCmd` procedure returns the keyboard equivalent field of this item.

`cmdChar`	The value of the item's keyboard equivalent field. The Menu Manager uses this value to map keyboard equivalents to menu commands or to indicate special characteristics of the menu item. If the `cmdChar` parameter contains $1B, the menu item has a submenu; a value of $1C indicates that the item has a script code; a value of $1D indicates that the Menu Manager reduces the item's `'ICON'` resource; and a value of $1E indicates that the item has an `'SICN'` resource.

DESCRIPTION

The `GetItemCmd` procedure returns the value in the keyboard equivalent field of the specified menu item in the `cmdChar` parameter (or 0 if the item doesn't have a keyboard equivalent, submenu, script code, reduced icon, or small icon).

SetItemCmd

Use the `SetItemCmd` procedure to set the value of the keyboard equivalent field of a menu item. You usually define the keyboard equivalents and other characteristics of your menu items in `'MENU'` resources rather than using the `SetItemCmd` procedure.

```
PROCEDURE SetItemCmd (theMenu: MenuHandle; item: Integer;
                      cmdChar: Char);
```

`theMenu`	A handle to the menu record of the menu containing the menu item whose keyboard equivalent field you wish to set.
`item`	The item number of the menu item. The `SetItemCmd` procedure sets the keyboard equivalent field of this item to the value specified in the `cmdChar` parameter.
`cmdChar`	The value of the item's keyboard equivalent field. The Menu Manager uses this value to map keyboard equivalents to menu commands or to define special characteristics of the menu item. To indicate that the menu item has a submenu, specify $1B in the `cmdChar` parameter; specify a value of $1C to indicate that the item has a script code; specify a value of $1D to indicate that the Menu Manager should reduce the item's `'ICON'` resource to the size of a small icon; and specify a value of $1E to indicate that the item has an `'SICN'` resource. The values $01 through $1A, as well as $1F and $20, are reserved for use by Apple. You should not use any of these reserved values in the `cmdChar` parameter.

DESCRIPTION

The `SetItemCmd` procedure sets the value in the keyboard equivalent field of the specified menu item in the `cmdChar` parameter (you can specify 0 if the item doesn't have a keyboard equivalent, submenu, script code, reduced icon, or small icon). If you specify that the item has a submenu, you should provide the menu ID of the submenu as the item's marking character. If you specify that the item has a script code, provide the script code in the icon field of the menu item. If you specify that the item has an `'SICN'` or a reduced `'ICON'` resource, provide the icon number in the icon field of the item.

Disposing of Menus

If you no longer need a menu in the menu list, you can delete the menu using `DeleteMenu`. You should then release the memory associated with that menu using the `DisposeMenu` procedure if you created the menu using `NewMenu`; otherwise, use the Resource Manager procedure `ReleaseResource`. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on the `ReleaseResource` routine.

DisposeMenu

To release the memory occupied by a menu's associated data structures, use either the `DisposeMenu` procedure or the Resource Manager procedure `ReleaseResource`. Use `DisposeMenu` if you created the menu using `NewMenu`; use `ReleaseResource` if you created the menu using `GetMenu` or read the resource in using `GetNewMBar`.

You should delete the menu from the current menu list using `DeleteMenu` or `ClearMenuBar` before calling the `DisposeMenu` procedure.

```
PROCEDURE DisposeMenu (theMenu: MenuHandle);
```

`theMenu`	A handle to the menu record of the menu you wish to dispose of.

DESCRIPTION

The `DisposeMenu` procedure releases the memory occupied by the specified menu's menu record. The handle that you pass in the parameter `theMenu` is not valid after `DisposeMenu` returns.

SEE ALSO

To delete a menu from the current menu list, see the description of the `DeleteMenu` procedure on page 3-109.

Counting the Items in a Menu

If your application needs to count the number of items in a menu—for example, in a menu that can contain a variable number of menu items such as the Font menu or Help menu—use the `CountMItems` function.

CountMItems

You can count the number of items in a menu using the `CountMItems` function.

```
FUNCTION CountMItems (theMenu: MenuHandle): Integer;
```

`theMenu`	A handle to the menu record of the menu whose items your application needs to count.

DESCRIPTION

The `CountMItems` function counts the number of items in the specified menu and returns as its function result the number of items in the menu.

Highlighting the Menu Bar

You can highlight (invert) a menu title or the entire menu bar using the `FlashMenuBar` procedure. (The `HiliteMenu` procedure highlights only menu titles.) In most cases your application should not highlight the menu bar; use `HiliteMenu` to highlight a menu title.

The user sets the number of times an enabled menu item flashes using the General Controls panel. The `SetMenuFlash` procedure can be used to control the number of times that menu items blink when the user chooses an enabled menu item; usually you should not change the setting chosen by the user.

FlashMenuBar

Use the `FlashMenuBar` procedure to highlight (invert) a menu title or the entire menu bar. You can call `FlashMenuBar` twice in a row to make the menu bar blink.

```
PROCEDURE FlashMenuBar (menuID: Integer);
```

`menuID`	The menu ID of the menu whose title you want to invert. Use 0 in this parameter to invert the entire menu bar. If the specified menu ID does not exist in the current menu list, the `FlashMenuBar` procedure inverts the entire menu bar.

DESCRIPTION

The `FlashMenuBar` procedure inverts the title of the specified menu or inverts the menu bar. To prevent unexpected colors from appearing in the menu bar, you should not call `FlashMenuBar` to invert a menu title while the entire menu bar is inverted.

Only one menu title can be inverted at a time. If no menus are currently highlighted, calling `FlashMenuBar` with a specific menu ID inverts the title of that menu. If you call `FlashMenuBar` again specifying another menu ID that is different from that of the previously inverted menu title, `FlashMenuBar` restores the previously highlighted menu to normal and then inverts the title of the specified menu.

SEE ALSO

You can also highlight a menu using the `HiliteMenu` procedure, described on page 3-119.

SetMenuFlash

Use the `SetMenuFlash` procedure to set the number of times a menu item blinks when the user chooses an enabled menu item. The user sets this value using the General Controls panel, and in most cases your application should not change the value set by the user.

```
PROCEDURE SetMenuFlash (count: Integer);
```

`count`	The number of times an enabled menu item should blink when the user chooses it. This value is initially set to 3 by the General Controls panel. A count of 0 disables the blinking. Values greater than 3 can be slow and distracting to the user.

DESCRIPTION

The `SetMenuFlash` procedure sets the number of times that the Menu Manager causes a menu item to blink when the user chooses an enabled menu item.

The appearance of blinking in a menu item is determined by the menu's menu definition procedure.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `MenuFlash` contains the current count (number of times) a menu item blinks when chosen by the user.

Recalculating Menu Dimensions

The Menu Manager uses the `CalcMenuSize` procedure to recalculate the dimensions of a menu whenever its contents have changed. In most cases your application does not need to use the `CalcMenuSize` procedure.

CalcMenuSize

The `CalcMenuSize` procedure recalculates the horizontal and vertical dimensions of a menu and stores the new values in the `menuWidth` and `menuHeight` fields of the menu record.

```
PROCEDURE CalcMenuSize (theMenu: MenuHandle);
```

`theMenu`	A handle to the menu record of the menu whose dimensions need recalculating.

DESCRIPTION

The `CalcMenuSize` procedure uses the menu definition procedure of the specified menu to calculate the dimensions of the menu.

Managing Entries in the Menu Color Information Table

The Menu Manager maintains color information about an application's menus in a menu color information table. The standard menu definition procedure defines the standard color for the menu bar, titles of menus, text and characteristics of a menu item, and background color of a displayed menu. You can change any of these colors by adding entries to your application's menu color information table. However, note that in most cases your application should use the default colors for its menus.

You can provide an `'mctb'` resource with resource ID 0 as one of your application's resources if you want to use colors other than the default colors for your application's menu bar and menus. (Or you can provide an `'mctb'` resource with the same resource ID as a `'MENU'` resource to define the color entries for a single menu.) You can also add entries to or delete entries from your application's menu color information table using the `SetMCEntries` and `DeleteMCEntries` procedures. You can get information about an entry using the `GetMCEntry` function. To get or set your application's menu color information table, use the `GetMCInfo` function or `SetMCInfo` procedure. To dispose of your application's menu color information table, use the `DisposeMCInfo` procedure.

Note that the menu color information table uses a format that is different from the standard color table format. "The Menu Color Information Table Record" beginning on page 3-98 describes the format of the menu color information table in detail.

GetMCInfo

Use the `GetMCInfo` function to get a handle to a copy of your application's menu color information table.

```
FUNCTION GetMCInfo: MCTableHandle;
```

DESCRIPTION

The `GetMCInfo` function creates a copy of your application's menu color information table and returns a handle to the copy. If the copy fails, `GetMCInfo` returns `NIL`.

SEE ALSO

See "The Menu Color Information Table Record" beginning on page 3-98 for a description of the format of the menu color information table.

SetMCInfo

Use the `SetMCInfo` procedure to set your application's menu color information table.

```
PROCEDURE SetMCInfo (menuCTbl: MCTableHandle);
```

`menuCTbl`	A handle to a menu color information table.

DESCRIPTION

The `SetMCInfo` procedure copies the table specified by the `menuCTbl` parameter to your application's menu color information table. If successful, the `SetMCInfo` procedure is responsible for disposing of your application's current menu color information table, so your application does not need to explicitly dispose of the current table.

Your application should call the Memory Manager function `MemError` to determine whether the `SetMCInfo` procedure successfully copied the table. If the `SetMCInfo` procedure cannot successfully copy the table, it does not dispose of the current menu color information table and the `MemError` function returns a nonzero result code. If the `SetMCInfo` procedure is able to successfully copy the table, it disposes of the current menu color information table and the `MemError` function returns the `noErr` result code.

If the menu color information table specifies a new menu bar color or new menu title colors, your application should call `DrawMenuBar` after calling `SetMCInfo`.

Note that `GetNewMBar` does not save your application's current menu color information table. If your application changes menu bars, you can save and restore your application's current menu color information table by calling `GetMCInfo` before `GetNewMBar` and calling `SetMCInfo` afterward.

SEE ALSO

See "The Menu Color Information Table Record" beginning on page 3-98 for a description of the format of the menu color information table. For an example of using the `GetMCInfo` and `SetMCInfo` routines to save and restore menu color information, see Listing 3-6 on page 3-52. See *Inside Macintosh: Memory* for information on the `MemError` function

DisposeMCInfo

Use the `DisposeMCInfo` procedure to dispose of a menu color information table. The `DisposeMCInfo` procedure is also available as the `DispMCInfo` procedure.

```
PROCEDURE DisposeMCInfo (menuCTbl: MCTableHandle);
```

`menuCTbl`	A handle to a menu color information table.

DESCRIPTION

The `DisposeMCInfo` procedure disposes of the menu color information table referred to by the `menuCTbl` parameter.

GetMCEntry

Use the `GetMCEntry` function to return information about an entry in your application's menu color information table. You can get information about the menu bar entry, a menu title entry, or a menu item entry.

```
FUNCTION GetMCEntry (menuID: Integer; menuItem: Integer)
                    : MCEntryPtr;
```

`menuID`	The menu ID that the `GetMCEntry` function should use to return information about the menu color information table. Specify 0 in the `menuID` parameter (and the `menuItem` parameter) to get the menu bar entry. Specify the menu ID of a menu in the current menu list in the `menuID` parameter and 0 in the `menuItem` parameter to get a specific menu title entry. Specify the menu ID of a menu in the current menu list in the `menuID` parameter and an item number in the `menuItem` parameter to get a specific menu item entry.
`menuItem`	The menu item that the `GetMCEntry` function should use to return information about the menu color information table. If you specify 0 in this parameter, `GetMCEntry` returns either the menu bar entry or the menu title entry, depending on the value of the `menuID` parameter. If you specify the item number of a menu item in this parameter and the menu ID of a menu in the current menu list in the `menuID` parameter, `GetMCEntry` returns a specific menu item entry.

DESCRIPTION

The `GetMCEntry` function returns a menu bar entry, a menu title entry, or a menu item entry according to the values specified in the `menuID` and `menuItem` parameters. If the `GetMCEntry` function finds the specified entry in your application's menu color information table, it returns a pointer to a record of data type `MCEntry`. If the specified entry is not found, `GetMCEntry` returns `NIL`.

s **WARNING**
The menu color information table is relocatable, so the pointer returned by the `GetMCEntry` function may not be valid across routines that may move or purge memory. Your application should make a copy of the menu color entry record if necessary. s

SEE ALSO

"The Menu Color Information Table Record" beginning on page 3-98 describes the entries in a menu color information table.

SetMCEntries

Use the `SetMCEntries` procedure to set entries in your application's menu color information table. You can set any or all of your application's menu item entries and menu title entries or the menu bar entry.

```
PROCEDURE SetMCEntries (numEntries: Integer;
                        menuCEntries: MCTablePtr);
```

`numEntries` The number of entries contained in the array of menu color entry records.

`menuCEntries`
A pointer to an array of menu color entry records. Specify the number of records in the array in the `numEntries` parameter.

DESCRIPTION

The `SetMCEntries` procedure sets any specified menu bar entry, menu title entry, or menu item entry according to the values specified in the menu color entry records. If an entry already exists for a specified menu color entry, the `SetMCEntries` procedure updates the entry in your application's menu color information table with the new values. If the entry doesn't exist, it is added to your application's menu color information table.

If any of the added entries specify a new menu bar color or new menu title colors, your application should call `DrawMenuBar` to update the menu bar with the new colors.

SPECIAL CONSIDERATIONS

The `SetMCEntries` procedure may move or purge memory. Your application should make sure that the array specified by the `menuCEntries` parameter is nonrelocatable before calling `SetMCEntries`.

SEE ALSO

"The Menu Color Information Table Record" beginning on page 3-98 describes the entries in a menu color information table.

DeleteMCEntries

Use the `DeleteMCEntries` procedure to delete one or all entries for a specific menu from your application's menu color information table. You can delete a menu item entry, a menu title entry, the menu bar entry, or all menu item entries of a specific menu. The `DeleteMCEntries` procedure is also available as the `DelMCEntries` procedure.

```
PROCEDURE DeleteMCEntries (menuID: Integer; menuItem: Integer);
```

`menuID`	The menu ID that the `DeleteMCEntries` procedure should use to determine which entry to delete from the menu color information table. Specify 0 in the `menuID` parameter (and the `menuItem` parameter) to delete the menu bar entry. Specify the menu ID of a menu in the current menu list in the `menuID` parameter and 0 in the `menuItem` parameter to delete a specific menu title entry. Specify the menu ID of a menu in the current menu list in the `menuID` parameter and an item number in the `menuItem` parameter to delete a specific menu item entry.
`menuItem`	The menu item that the `DeleteMCEntries` procedure should use to determine which entry to delete from the menu color information table. If you specify 0 in this parameter, `DeleteMCEntries` deletes either the menu bar entry or menu title entry, depending on the value of the `menuID` parameter. If you specify the item number of a menu item in this parameter and the menu ID of a menu in the current menu list in the `menuID` parameter, `DeleteMCEntries` deletes a specific menu item entry. You can also delete all menu item entries for a specific menu from your application's menu color information table using this constant:

```
CONST
mctAllItems     = -98;  {delete all menu item entries }
                        { for the specified menu}
```

DESCRIPTION

The `DeleteMCEntries` procedure deletes a menu bar entry, a menu title entry, a menu item entry, or all menu item entries of a given menu, according to the values specified in the `menuID` and `menuItem` parameters. If the `GetMCEntry` function does not find the specified entry in your application's menu color information table, it does not delete the entry. Your application should not delete the last entry in your application's menu color information table.

If any of the deleted entries changes the menu bar color or a menu title color, your application should call `DrawMenuBar` to update the menu bar.

Application-Defined Routine

Apple provides a standard menu definition procedure and standard menu bar definition function. The Menu Manager uses the menu definition procedure and menu bar definition function to display and perform basic operations on menus and the menu bar. Although the Menu Manager allows you to provide your own menu bar definition function, Apple recommends that you use the standard menu bar definition function. Similarly, in most cases the standard menu definition procedure should meet the needs of most applications. However, if your application has special needs, you can choose to provide your own menu definition procedure. If you do so, define your menu definition procedure so that it emulates the standard behavior of menus as much as possible. If you define your own menus, they should follow the guidelines described in this chapter and in *Macintosh Human Interface Guidelines*.

The Menu Definition Procedure

The Menu Manager uses the menu definition procedure of a menu to draw the menu items in the menu, to determine which item the user chose from the menu, and to calculate the menu's dimensions. If you provide your own menu definition procedure, it should also perform these tasks.

Apple provides a standard menu definition procedure, stored as a resource in the System file. The standard menu definition procedure is the `'MDEF'` resource with resource ID 0. When you define your menus, you specify the menu definition procedure the Menu Manager should use when managing them. You'll usually want to use the standard menu definition procedure for your application. However, if you need a feature not provided by the standard menu definition procedure (for example, if you want to include more graphics in your menus), you can choose to write your own menu definition procedure.

MyMenuDef

You can provide your own menu definition procedure if you need special features in a menu other than those provided by the standard menu definition procedure. This section describes how to define your own menu definition procedure, defines the parameters passed to your procedure by the Menu Manager, and describes the general actions your procedure should perform.

```
PROCEDURE MyMenuDef (message: Integer; theMenu: MenuHandle;
                     VAR menuRect: Rect; hitPt: Point;
                     VAR whichItem: Integer);
```

`message`	A number that identifies the operation that the menu definition procedure should perform. The `message` parameter can contain any one of these values:

```
CONST
      mDrawMsg        = 0; {draw the menu}
      mChooseMsg      = 1; {tell which item was chosen }
                           { and highlight it}
      mSizeMsg        = 2; {calculate menu dimensions}
      mPopUpMsg       = 3; {calculate rectangle of }
                           { the pop-up box}
```

Your menu definition procedure should not respond to any value other than the four constants listed above.

`theMenu`	A handle to the menu record of the menu that the operation should affect.
`menuRect`	The rectangle (in global coordinates) in which the menu is located; the Menu Manager provides this information to the menu definition procedure only when the value in the `message` parameter is the `mDrawMsg` or `mChooseMsg` constant. When the value in the `message` parameter is the `mPopUpMsg` constant, the menu definition procedure should calculate and then return the dimensions of the pop-up box in this parameter. When the value in the `message` parameter is the `mSizeMsg` constant, the menu definition procedure should calculate the horizontal and vertical dimensions of the menu rectangle and store these values in the `menuWidth` and `menuHeight` fields of the menu record.
`hitPt`	A mouse location (in global coordinates). The Menu Manager provides information in this parameter to the menu definition procedure when the value in the `message` parameter is the `mChooseMsg` or `mPopUpMsg` constant. When the menu definition procedure receives the `mChooseMsg` constant in the `message` parameter, it should determine whether the mouse location specified in the `hitPt` parameter is in an enabled menu item and highlight or unhighlight the item specified in the `whichItem` parameter appropriately. When the menu definition procedure receives the `mPopUpMsg` constant in the `message` parameter, the `hitPt` parameter contains the top-left coordinates of the closed pop-up box, which your procedure can use to calculate the rectangle of the open pop-up box.
`whichItem`	The item number of the last item chosen from this menu (or 0 if an item hasn't been chosen). The Menu Manager provides information in this parameter to the menu definition procedure when the value in the `message` parameter is the `mChooseMsg` constant. When the menu definition procedure receives the `mChooseMsg` constant in the `message` parameter, it should determine whether the mouse location specified in the `hitPt` parameter is in an enabled menu item. If so, the menu definition procedure should unhighlight the item specified by the `whichItem` parameter, highlight the new item, and return the new item number in `whichItem`. If the mouse location isn't in an enabled menu item, the menu definition procedure should unhighlight the item specified by the `whichItem` parameter and return 0 in the `whichItem` parameter.

DESCRIPTION

The Menu Manager calls your menu definition procedure whenever it needs your definition procedure to perform a certain action on a specific menu. The action your menu definition procedure should perform depends on the value of the `message` parameter.

If you provide your own menu definition procedure, store it in a resource of type `'MDEF'` and include its resource ID in the description of each menu that uses your own definition procedure. If you create a menu using `GetMenu` (or `GetNewMBar`), the Menu Manager reads the menu definition procedure into memory and stores a handle to it in the `menuProc` field of the menu's menu record.

If you create a menu using `NewMenu`, the Menu Manager stores a handle to the standard menu definition procedure in the `menuProc` field of the menu's menu record. In this case you must replace the value in the `menuProc` field with a handle to your own procedure and then call the `CalcMenuSize` procedure. If your menu definition procedure is in a resource file, you can get its handle by using the Resource Manager to read it from the resource file into memory. However, note that you should usually store your menus in resources (rather than using `NewMenu`) to make your application easier to localize. See the "Resource Manager" chapter in *Inside Macintosh: More Macintosh Toolbox* for information on the Resource Manager.

The menu definition procedure is responsible for drawing the contents of the menu and its menu items, determining whether the cursor is in a displayed menu, highlighting and unhighlighting menu items, and calculating a menu's dimensions.

When the Menu Manager requests your menu definition procedure to perform an action on a menu, it provides your procedure with a handle to its menu record. This allows your procedure to access the data in the menu record and to use any data in the variable data portion of the menu record to appropriately handle the menu items.

When the Menu Manager creates a menu as a result of an application calling `GetMenu` or `GetNewMBar`, it fills out the `menuID`, `menuProc`, `enableFlags`, `menuTitle`, and `itemDefinitions` fields of the menu record according to its resource definition. If the menu is managed by your menu definition procedure, the Menu Manager calls your procedure (specifying `mSizeMsg`) to calculate and fill in the `menuHeight` and `menuWidth` fields of the menu record. The menu items are described by a variable length field (`itemDefinitions`) in the menu record. Your menu definition procedure can define and use this variable-length data in any manner it chooses.

For pop-up menus that are not implemented as controls, the Menu Manager uses the menu definition procedure to support pop-up menus. If your menu definition procedure supports pop-up menus, it should respond appropriately to the `mPopUpMsg` constant.

The Menu Manager specifies the `mPopUpMsg` constant in the `message` parameter and calls your menu definition procedure whenever it needs to calculate the rectangle bounded by the pop-up box for a pop-up menu that is managed by your menu definition procedure. The parameter `theMenu` contains a handle to the menu record of the pop-up menu, the `hitPt` parameter contains the top-left coordinates of the pop-up box, and `whichItem` contains the previously chosen item. Your menu definition procedure should calculate the rectangle in which the pop-up menu is to appear and return this rectangle in the `menuRect` parameter. If the menu is so large that it scrolls, return the actual top of the menu in the `whichItem` parameter. For pop-up menus, your menu definition procedure also must place the pop-up menu's scrolling information in the global variables `TopMenuItem` and `AtMenuBottom`. Place in `TopMenuItem` the pixel value of the top of the scrollable menu, and place in `AtMenuBottom` the pixel value of the bottom of the scrollable menu.

Note

Your menu definition procedure should not assume that the A5 register is properly set up, so your procedure can't refer to any of the QuickDraw global variables. u

SEE ALSO

For additional information on how your menu definition procedure should respond when it receives the `mDrawMsg`, `mChooseMsg`, or `mSizeMsg` constant in the `message` parameter, see "Writing Your Own Menu Definition Procedure" beginning on page 3-87.

Resources

This section describes the menu (`'MENU'`) resource, menu bar (`'MBAR'`) resource, and menu color information table (`'mctb'`) resource. Usually you should define your menus using `'MENU'` resources, define the menus in your menu bar in an `'MBAR'` resource, and use the `GetNewMBar` function to read in the descriptions of your menus and menu bar.

If you want to use colors other than the default colors in a menu, you can provide an `'mctb'` resource with the same resource ID as its corresponding `'MENU'` resource, or you can provide an `'mctb'` resource with resource ID 0 to define colors for all your menus and your menu bar.

If you choose to provide your own menu definition procedure, you should store your routine in an `'MDEF'` resource.

To create a `'MENU'`, an `'MBAR'`, or an `'mctb'` resource, either you can specify the resource description in an input file and compile the resource using a resoure compiler, such as Rez, or you can directly create your resources in a resource file using a tool such as ResEdit. This section describes the structures of these resources after they are compiled by the Rez resource compiler. If you are interested in creating the Rez input files for these resources, see "Using the Menu Manager," beginning on page 3-41, for detailed information.

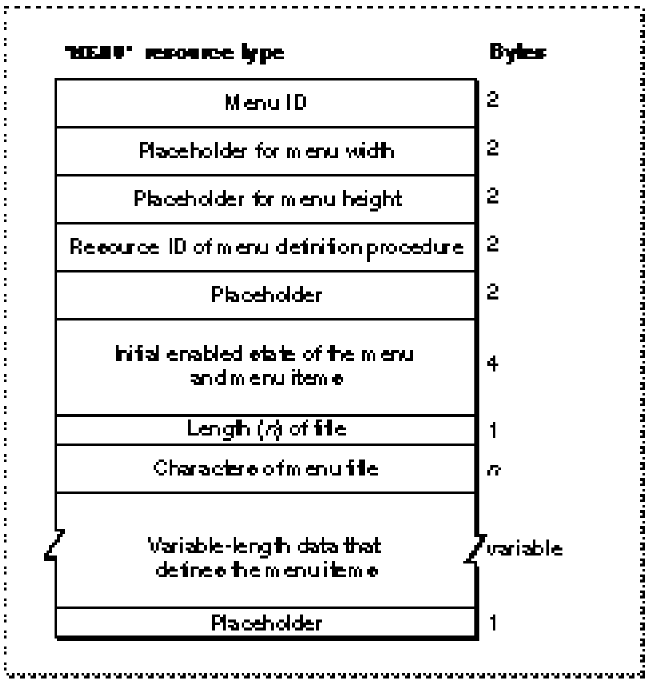
The Menu Resource

You can provide descriptions of your menus in `'MENU'` resources and use the `GetMenu` function or `GetNewMBar` function (if you also provide an `'MBAR'` resource) to read in the descriptions of your menus. After reading in the resource description, the Menu Manager stores the information about specific menus in menu records.

s **WARNING**
Menus in a resource must not be purgeable. s

Figure 3-37 shows the format of a compiled `'MENU'` resource. See Listing 3-1 on page 3-43 for a description of a `'MENU'` resource in Rez input format.

Figure 3-37 Structure of a compiled menu (`'MENU'`) resource


'MENU' resource type
Bytes
Menu ID
2
Placeholder for menu width
2
Placeholder for menu height
2
Resource ID of menu definition procedure
2
Placeholder
2
Initial enabled state of the menu and menu items
4
Length (n) of title
1
Characters of menu title
n
Variable-length data that defines the menu items
variable
Placeholder
1

A compiled version of a `'MENU'` resource contains the following elements:

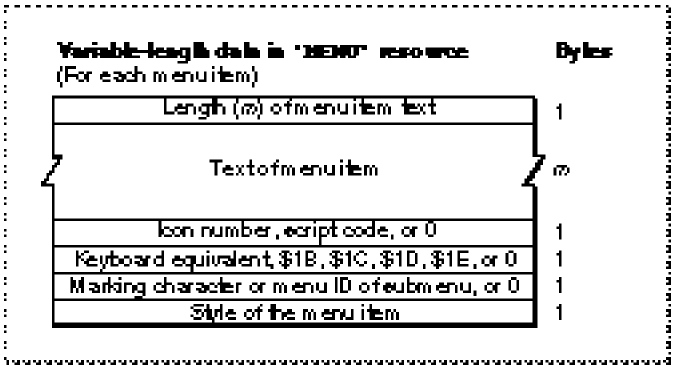
- Menu ID. Each menu in your application should have a unique menu ID. Note that the menu ID does not have to match the resource ID, although by convention most applications assign the same number for a menu's resource ID and menu ID. A negative menu ID indicates a menu belonging to a desk accessory (except for submenus of a desk accessory). A menu ID from 1 through 235 indicates a menu (or submenu) of an application; a menu ID from 236 through 255 indicates a submenu of a desk accessory. Apple reserves the menu ID of 0.
- Placeholder (two integers containing 0) for the menu's width and height. After reading in the resource data, the Menu Manager requests the menu's menu definition procedure to calculate the width and height of the menu and to store these values in the `menuWidth` and `menuHeight` fields of the menu record.
- Resource ID of the menu's menu definition procedure. If the integer 0 appears here (as specified by the `textMenuProc` constant in the Rez input file), the Menu Manager uses the standard menu definition procedure to manage the menu. If you provide your own menu definition procedure, its resource ID should appear in these bytes.

After reading in the menu's resource data, the Menu Manager reads in the menu definition procedure, if necessary. The Menu Manager stores a handle to the menu's menu definition procedure in the `menuProc` field of the menu record.

- Placeholder (an integer containing 0).
- The initial enabled state of the menu and first 31 menu items. This is a 32-bit value, where bits 1–31 indicate if the corresponding menu item is disabled or enabled, and bit 0 indicates whether the menu is enabled or disabled. The Menu Manager automatically enables menu items greater than 31 when a menu is created.
- The length (in bytes) of the menu title.
- The title of the menu.
- Variable-length data that describes the menu items. If you provide your own menu definition procedure, you can define and provide this variable-length data according to the needs of your procedure. The Menu Manager simply reads in the data for each menu item and stores it as variable data at the end of the menu record. The menu definition procedure is responsible for interpreting the contents of the data. For example, the standard menu definition procedure interprets this data according to the description given in the following paragraphs.
- Placeholder (a byte containing 0) to indicate the end of the menu item definitions.

If you use the standard menu definition procedure, your `'MENU'` resource should describe the menu items in this manner. For each menu item, you need to provide its text, the icon number, the keyboard equivalent or other value ($1B to indicate the menu item has a submenu, $1C to indicate a script code other than the system script for the item's text, $1D to indicate the item's icon should be reduced, or $1E to indicate that an `'SICN'` icon should be used), the marking character of the menu item or menu ID of the menu item's submenu, and the font style of the menu item's text. If an item doesn't have a particular characteristic, specify 0 for that characteristic. Figure 3-38 shows the variable-length data portion of a compiled `'MENU'` resource that uses the standard menu definition procedure.

Figure 3-38 The variable-length data that describes menu items as defined by the standard menu definition procedure


Variable-length data in 'MENU' resource
(For each menu item)
Bytes
Length (m) of menu item text
1
Text of menu item
m
Icon number, script code, or 0
1
Keyboard equivalent, $1B, $1C, $1D, $1E, or 0
1
Marking character or menu ID of submenu, or 0
1
Style of the menu item
1

The variable-length data portion of a compiled version of a `'MENU'` resource that uses the standard menu definition procedure contains the following elements:

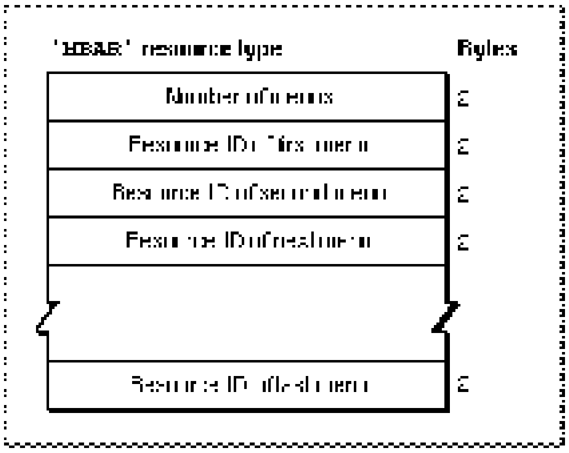
- Length (in bytes) of the menu item's text.
- Text of the menu item.
- Icon number, script code, or 0 (as specified by the `noicon` constant in a Rez input file) if the menu item doesn't contain an icon and uses the system script. The icon number is a number from 1 through 255 (or from 1 through 254 for small or reduced icons). The Menu Manager adds 256 to the icon number to generate the resource ID of the menu item's icon. If a menu item has an icon, you should also provide a `'cicn'` or an `'ICON'` resource with the resource ID equal to the icon number plus 256. If you want the Menu Manager to reduce an `'ICON'` resource to the size of a small icon, also provide the value $1D in the keyboard equivalent field. If you provide an `'SICN'` resource, provide $1E in the keyboard equivalent field. Otherwise, the Menu Manager looks first for a `'cicn'` resource with the calculated resource ID and uses that icon. If you want the Menu Manager to draw the item's text in a script other than the system script, specify the script code here and also provide $1C in the keyboard equivalent field. If the script system for the specified script is installed, the Menu Manager draws the item's text using that script. An item that is drawn in a script other than the system script cannot also have an icon.
- Keyboard equivalent (specified as a 1-byte character), the value $1B (as specified by the constant `hierarchicalMenu` in a Rez input file) if the item has a submenu, the value $1C if the item uses a script other than the system script, or 0 (as specified by the `nokey` constant in a Rez input file) if the item has neither a keyboard equivalent nor a submenu and uses the system script. A menu item can have a keyboard equivalent, a submenu, a small icon, a reduced icon, or a script code, but not more than one of these characteristics. For items containing icons, you can provide $1D in this field if you want the Menu Manager to reduce an `'ICON'` resource to the size of a small icon. Provide $1E if you want the Menu Manager to use an `'SICN'` resource for the item's icon. The values $01 through $1A as well as $1F and $20 are reserved for use by Apple; your application should not use any of these reserved values in this field.
- Marking character, the menu ID of the item's submenu, or 0 (as specified by the `nomark` constant in a Rez input file) if the item has neither a mark nor a submenu. A menu item can have a mark or a submenu, but not both. Submenus of an application should have menu IDs from 1 through 235; submenus of a desk accessory should have menu IDs from 236 through 255.
- Font style of the menu item. The constants `bold`, `italic`, `plain`, `outline`, and `shadow` can be used in a Rez input file to define their corresponding styles.

If you provide your own menu definition procedure, you should use the same format for your resource descriptions of menus as shown in Figure 3-37. You can use the same format or a format of your choosing to describe menu items. You can also use bits 1–31 of the `enableFlags` field of the menu record as you choose; however, bit 0 must still indicate whether the menu is enabled or disabled.

The Menu Bar Resource

You can describe the order and number of menus in your menu bar in an `'MBAR'` resource, and you can describe your menus in `'MENU'` resources. If you do so, you can use the `GetNewMBar` function to read in the descriptions of your menus and create a new menu list. The Menu Manager stores information about your application's menu bar in a menu list. Figure 3-39 shows the format of a compiled `'MBAR'` resource. (See Listing 3-4 on page 3-49 for a description of an `'MBAR'` resource in Rez input format.)

Figure 3-39 Structure of a compiled menu bar (`'MBAR'`) resource


'MBAR' resource type
Bytes
Number of menus
2
Resource ID of first menu
2
Resource ID of second menu
2
Resource ID of third menu
2
Resource ID of last menu
2

A compiled version of an `'MBAR'` resource contains the following elements:

n Number of menus described by this menu bar.

n A variable number (the amount should match the number declared in the first 2 bytes) of resource IDs; each resource ID should identify a `'MENU'` resource.

If you use the `GetNewMBar` function, the Menu Manager places the menus in the menu bar according to the order that they appear in the `'MBAR'` resource.

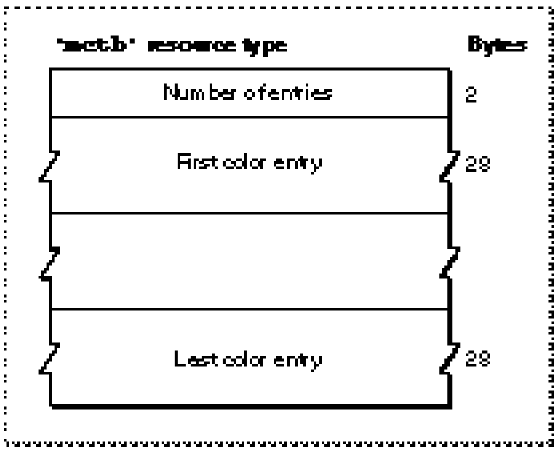
The Menu Color Information Table Resource

To use colors other than the default colors in a menu, provide a **menu color information table** (`'mctb'`) **resource** with the same resource ID as its corresponding `'MENU'` resource. You can also choose to provide an `'mctb'` resource with resource ID 0 to define colors for all your menus and your menu bar. Note that you should usually use the default colors provided by the Menu Manager.

The Menu Manager stores color information about your application's menus and menu bar in a menu color information table. If you provide an `'mctb'` resource with resource ID 0, the Menu Manager reads the resource in when your application calls `InitMenus` and stores the information in your application's menu color information table. If you provide an `'mctb'` resource with the same resource ID as a `'MENU'` resource, when you use `GetMenu` to read in the resource description of the menu (or `GetNewMBar` to read in all menus in the menu bar), the Menu Manager also reads in any associated `'mctb'` resource (if it exists). "The Menu Color Information Table Record" beginning on page 3-98 describes the format of the menu color information table.

Figure 3-40 shows the format of a compiled `'mctb'` resource.

Figure 3-40 Structure of a compiled menu color information table (`'mctb'`) resource



A compiled version of an `'mctb'` resource contains the following elements:

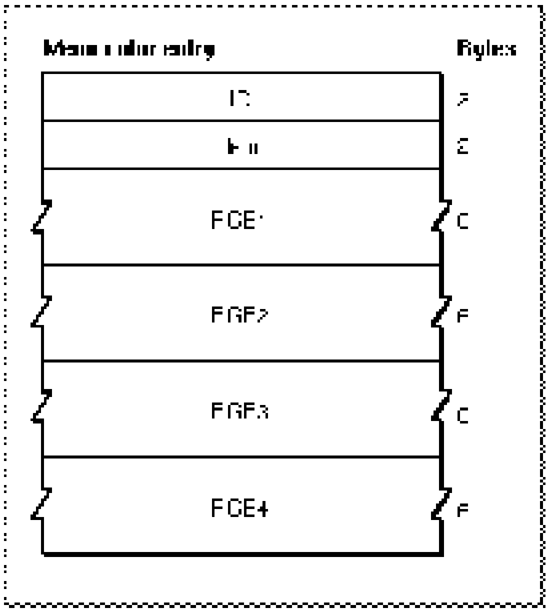
- a count of the number of menu color entry descriptions
- a variable number of menu color entries

A color entry defines colors for various parts of the menu and menu bar. Figure 3-41 on the next page shows the format of a compiled menu color entry in an `'mctb'` resource.

Each menu color entry in an `'mctb'` resource contains the following:

- A menu ID to indicate that this entry is either a menu item entry or menu title entry, 0 to indicate that this entry is a menu bar entry, or –99 to indicate that this is the last entry in this resource.
- An item number to indicate that this entry is a menu item entry, or 0 to indicate that this is either a menu title or menu bar entry. Together, the menu ID and menu item determine how the type of menu color entry is described. See Table 3-7 on page 3-100 for a complete description of how the menu ID and menu item specifications define the type of menu color entry.
- RGB1: for a menu bar entry, the default color for menu titles; for a menu title entry, the title color of a specific menu; for a menu item entry, the mark color for a specific item.
- RGB2: for a menu bar entry, the default background color of a displayed menu; for a menu title entry, the default color for the menu bar; for a menu item entry, the color for the text of a specific item.

Figure 3-41 Structure of a menu color entry in an `'mctb'` resource



- RGB3: for a menu bar entry, the default color of items in a displayed menu; for a menu title entry, the default color for items in a specific menu; for a menu item entry, the color for the keyboard equivalent of a specific item.
- RGB4: for a menu bar entry, the default color of the menu bar; for a menu title entry, the background color of a specific menu; for a menu item entry, the background color of a specific menu.

The Menu Definition Procedure Resource

If you provide your own menu definition procedure, you should store it in a resource of type `'MDEF'`. Provide as the resource data the compiled or assembled code of your menu definition procedure. The entry point of your procedure must be at the beginning of the resource data.

If you define your menus in `'MENU'` resources (and use the `GetMenu` or `GetNewMBar` function), you specify the menu definition procedure that the Menu Manager should use to manage the menu in the `'MENU'` resource. If you use the `NewMenu` function (instead of `'MENU'` resources), your application must explicitly replace the handle to the standard menu definition procedure in the `menuProc` field of the menu record with a handle to the desired menu definition procedure.

Summary of the Menu Manager

Pascal Summary

Constants

```
CONST
      noMark         = 0;  {menu item doesn't have a marking character}

      {values for the message parameter to the menu definition procedure}
      mDrawMsg       = 0;  {draw the menu items of a menu}
      mChooseMsg     = 1;  {highlight or unhighlight a menu item as }
                           { appropriate if the cursor is in a menu item}
      mSizeMsg       = 2;  {calculate the dimensions of a menu}
      mPopUpMsg      = 3;  {calculate the open pop-up box rectangle}

      textMenuProc   = 0;  {resource ID of standard menu definition }
                           { procedure}
      hMenuCmd       = 27; {constant ($1B) specified as keyboard equivalent }
                           { to indicate a menu item has a submenu}
      hierMenu       = -1; {constant used with InsertMenu routine to insert }
                           { a submenu or pop-up menu into the submenu }
                           { portion of the current menu list}
      mctAllItems    = -98;{search for all items with the given ID}
      mctLastIDIndic = -99;{last menu color table entry has this value }
                           { in the ID field of the entry}
```

Data Types

```
TYPE
      MenuInfo =                 {menu record}
      RECORD
         menuID:     Integer;    {number that identifies the menu}
         menuWidth:  Integer;    {width (in pixels) of the menu}
         menuHeight: Integer;    {height (in pixels) of the menu}
         menuProc:   Handle;     {menu definition procedure}
         enableFlags:LongInt;    {indicates whether menu and }
                                 { menu items are enabled}
         menuData:   Str255;      {title of menu}
         {itemDefinitions}        {variable-length data that }
                                 { defines the menu items}
      END;

      MenuPtr     = ^MenuInfo;    {pointer to a menu record}
      MenuHandle  = ^MenuPtr;     {handle to a menu record}

      MCEntry =                   {menu color entry record}
      RECORD
         mctID:      Integer;     {menu ID or 0 for menu bar}
         mctItem:    Integer;     {menu item number or 0 for }
                                 { menu title}
         mctRGB1:    RGBColor;    {usage depends on mctID and }
                                 { mctItem}
         mctRGB2:    RGBColor;    {usage depends on mctID and }
                                 { mctItem}
         mctRGB3:    RGBColor;    {usage depends on mctID and }
                                 { mctItem}
         mctRGB4:    RGBColor;    {usage depends on mctID and }
                                 { mctItem}
         mctReserved:Integer;     {reserved}
      END;

      MCEntryPtr  = ^MCEntry;     {pointer to a menu color entry record}

      MCTable     = ARRAY[0..0] OF MCEntry;  {menu color table}
      MCTablePtr  = ^MCTable;        {pointer to a menu color table}
      MCTableHandle = ^MCTablePtr;  {handle to a menu color table}
```

Menu Manager Routines

Initializing the Menu Manager

```
PROCEDURE InitMenus;
PROCEDURE InitProcMenu        (resID: Integer);
```

Creating Menus

```
FUNCTION NewMenu              (menuID: Integer; menuTitle: Str255)
                               : MenuHandle;
FUNCTION GetMenu              (resourceID: Integer): MenuHandle;
```

Adding Menus to and Removing Menus From the Current Menu List

```
PROCEDURE InsertMenu        (theMenu: MenuHandle; beforeID: Integer);
PROCEDURE DeleteMenu        (menuID: Integer);
PROCEDURE ClearMenuBar;
```

Getting a Menu Bar Description From an 'MBAR' Resource

```
FUNCTION GetNewMBar         (menuBarID: Integer): Handle;
```

Getting and Setting the Menu Bar

```
FUNCTION GetMenuBar: Handle;
PROCEDURE SetMenuBar        (menuList: Handle);
FUNCTION GetMBarHeight: Integer;
```

Drawing the Menu Bar

```
PROCEDURE DrawMenuBar;
PROCEDURE InvalMenuBar;
```

Responding to the User's Choice of a Menu Command

```
FUNCTION MenuSelect         (startPt: Point): LongInt;
FUNCTION MenuKey            (ch: Char): LongInt;
FUNCTION MenuChoice: LongInt;
PROCEDURE HiliteMenu        (menuID: Integer);
FUNCTION PopUpMenuSelect    (menu: MenuHandle;
                             Top: Integer; Left: Integer;
                             PopUpItem: Integer): LongInt;
PROCEDURE SystemMenu        (menuResult: LongInt);
FUNCTION SystemEdit         (editCmd: Integer): Boolean;
```

Getting a Handle to a Menu Record

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
FUNCTION GetMenuHandle      (menuID: Integer): MenuHandle;
FUNCTION HMGetHelpMenuHandle
                            (VAR mh: MenuHandle): OSErr;
```

Adding and Deleting Menu Items

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
PROCEDURE AppendMenu        (menu: MenuHandle; data: Str255);
PROCEDURE InsertMenuItem    (theMenu: MenuHandle; itemString: Str255;
                             afterItem: Integer);
```

```
PROCEDURE DeleteMenuItem    (theMenu: MenuHandle; item: Integer);
PROCEDURE AppendResMenu     (theMenu: MenuHandle; theType: ResType);
PROCEDURE InsertResMenu     (theMenu: MenuHandle; theType: ResType;
                             afterItem: Integer);
```

Getting and Setting the Appearance of Menu Items

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
PROCEDURE EnableItem        (theMenu: MenuHandle; item: Integer);
PROCEDURE DisableItem       (theMenu: MenuHandle; item: Integer);
PROCEDURE GetMenuItemText   (theMenu: MenuHandle; item: Integer;
                             VAR itemString: Str255);
PROCEDURE SetMenuItemText   (theMenu: MenuHandle; item: Integer;
                             itemString: Str255);
PROCEDURE GetItemStyle      (theMenu: MenuHandle; item: Integer;
                             VAR chStyle: Style);
PROCEDURE SetItemStyle      (theMenu: MenuHandle; item: Integer;
                             chStyle: Style);
PROCEDURE GetItemMark       (theMenu: MenuHandle; item: Integer;
                             VAR markChar: Char);
PROCEDURE SetItemMark       (theMenu: MenuHandle; item: Integer;
                             markChar: Char);
PROCEDURE CheckItem         (theMenu: MenuHandle; item: Integer;
                             checked: Boolean);
PROCEDURE GetItemIcon       (theMenu: MenuHandle; item: Integer;
                             VAR iconIndex: Byte);
PROCEDURE SetItemIcon       (theMenu: MenuHandle; item: Integer;
                             iconIndex: Byte);
PROCEDURE GetItemCmd        (theMenu: MenuHandle; item: Integer;
                             VAR cmdChar: CHAR);
PROCEDURE SetItemCmd        (theMenu: MenuHandle; item: Integer;
                             cmdChar: CHAR);
```

Disposing of Menus

```
PROCEDURE DisposeMenu       (theMenu: MenuHandle);
```

Counting the Items in a Menu

```
FUNCTION CountMItems        (theMenu: MenuHandle): Integer;
```

Highlighting the Menu Bar

```
PROCEDURE FlashMenuBar      (menuID: Integer);
PROCEDURE SetMenuFlash      (count: Integer);
```

Recalculating Menu Dimensions

```
PROCEDURE CalcMenuSize      (theMenu: MenuHandle);
```

Managing Entries in the Menu Color Information Table

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
FUNCTION GetMCInfo: MCTableHandle;
PROCEDURE SetMCInfo         (menuCTbl: MCTableHandle);
PROCEDURE DisposeMCInfo     (menuCTbl: MCTableHandle);
FUNCTION GetMCEntry         (menuID: Integer; menuItem: Integer)
                             : MCEntryPtr;
PROCEDURE SetMCEntries      (numEntries: Integer;
                             menuCEntries: MCTablePtr);
PROCEDURE DeleteMCEntries   (menuID: Integer; menuItem: Integer);
```

Application-Defined Routine

```
PROCEDURE MyMenuDef         (message: Integer; theMenu: MenuHandle;
                             VAR menuRect: Rect; hitPt: Point;
                             VAR whichItem: Integer);
```

C Summary

Constants

```
enum {
      #define noMark '\0'  /*menu item doesn't have a marking character*/

      /*values for the message parameter to the menu definition procedure*/
      mDrawMsg       = 0,  /*draw the menu items of a menu*/
      mChooseMsg     = 1,  /*highlight or unhighlight a menu item as */
                           /* appropriate if the cursor is in a menu item*/
      mSizeMsg       = 2,  /*calculate the dimensions of a menu*/
      mPopUpMsg      = 3,  /*calculate the open pop-up box rectangle*/

      textMenuProc   = 0,  /*resource ID of standard menu definition */
                           /* procedure*/
      hMenuCmd       = 27, /*constant ($1B) specified as keyboard */
                           /* equivalent to indicate an item has a submenu*/
      hierMenu       = -1, /*constant used with InsertMenu to insert */
                           /* a submenu or pop-up menu into the submenu */
                           /* portion of the current menu list*/
      mctAllItems    = -98,/*search for all items with the given ID*/
      mctLastIDIndic = -99 /*last menu color table entry has this value */
                           /* in the ID field of the entry*/
   };
```

Data Types

```
struct MenuInfo {          /*menu record*/
         short       menuID;         /*number that identifies the menu*/
         short       menuWidth;      /*width (in pixels) of the menu*/
         short       menuHeight;     /*height (in pixels) of the menu*/
         Handle      menuProc;       /*menu definition procedure*/
         long        enableFlags;    /*indicates whether menu and */
                                     /* menu items are enabled*/
         Str255      menuData;       /*title of menu*/
         /*itemDefinitions*/         /*variable-length data that */
                                     /* defines the menu items*/
      };

typedef struct MenuInfo MenuInfo;         /*pointer to a menu record*/
typedef MenuInfo *MenuPtr, **MenuHandle;  /*handle to a menu record*/

struct MCEntry {           /*menu color entry record*/
         short       mctID;          /*menu ID or 0 for menu bar*/
         short       mctItem;        /*menu item number or 0 for */
                                     /* menu title*/
         RGBColor    mctRGB1;        /*usage depends on mctID and */
                                     /* mctItem*/
         RGBColor    mctRGB2;        /*usage depends on mctID and */
                                     /* mctItem*/
         RGBColor    mctRGB3;        /*usage depends on mctID and */
                                     /* mctItem*/
         RGBColor    mctRGB4;        /*usage depends on mctID and */
                                     /* mctItem*/
         short       mctReserved;    /*reserved*/
      };

typedef struct MCEntry MCEntry;
typedef MCEntry *MCEntryPtr;        /*pointer to a menu color entry record*/
                                     /*menu color table*/
typedef MCEntry MCTable[1], *MCTablePtr, **MCTableHandle;
```

Menu Manager Routines

Initializing the Menu Manager

```
pascal void InitMenus       (void);
pascal void InitProcMenu    (short resID);
```

Creating Menus

```
pascal MenuHandle NewMenu   (short menuID, const Str255 menuTitle);
pascal MenuHandle GetMenu   (short resourceID);
```

Adding Menus to and Removing Menus From the Current Menu List

```
pascal void InsertMenu      (MenuHandle theMenu, short beforeID);
pascal void DeleteMenu      (short menuID);
pascal void ClearMenuBar    (void);
```

Getting a Menu Bar Description From an 'MBAR' Resource

```
pascal Handle GetNewMBar    (short menuBarID);
```

Getting and Setting the Menu Bar

```
pascal Handle GetMenuBar    (void);
pascal void SetMenuBar      (Handle menuList);
#define GetMBarHeight()     (* (short*) 0x0BAA)
```

Drawing the Menu Bar

```
pascal void DrawMenuBar     (void);
pascal void InvalMenuBar    (void);
```

Responding to the User’s Choice of a Menu Command

```
pascal long MenuSelect      (Point startPt);
pascal long MenuKey         (short ch);
pascal long MenuChoice      (void);
pascal void HiliteMenu      (short menuID);
pascal long PopUpMenuSelect (MenuHandle menu, short top, short left,
                             short popUpItem);
pascal void SystemMenu      (long menuResult);
pascal Boolean SystemEdit   (short editCmd);
```

Getting a Handle to a Menu Record

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
pascal MenuHandle GetMenuHandle
                             (short menuID);
pascal OSErr HMGetHelpMenuHandle
                             (MenuHandle *mh);
```

Adding and Deleting Menu Items

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
pascal void AppendMenu      (MenuHandle menu, ConstStr255Param data);
pascal void InsertMenuItem  (MenuHandle theMenu,
                             ConstStr255Param itemString,
                             short afterItem);
pascal void DeleteMenuItem  (MenuHandle theMenu, short item);
pascal void AppendResMenu   (MenuHandle theMenu, ResType theType);
pascal void InsertResMenu   (MenuHandle theMenu, ResType theType,
                             short afterItem);
```

Getting and Setting the Appearance of Menu Items

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
pascal void EnableItem      (MenuHandle theMenu, short item);
pascal void DisableItem     (MenuHandle theMenu, short item);
pascal void GetMenuItemText (MenuHandle theMenu, short item,
                             Str255 itemString);
pascal void SetMenuItemText (MenuHandle theMenu, short item,
                             ConstStr255Param itemString);
pascal void GetItemStyle    (MenuHandle theMenu, short item,
                             Style *chStyle);
pascal void SetItemStyle    (MenuHandle theMenu, short item, short chStyle);
pascal void GetItemMark     (MenuHandle theMenu, short item,
                             short *markChar);
pascal void SetItemMark     (MenuHandle theMenu, short item,
                             short markChar);
pascal void CheckItem       (MenuHandle theMenu, short item,
                             Boolean checked);
pascal void GetItemIcon     (MenuHandle theMenu, short item,
                             short *iconIndex);
pascal void SetItemIcon     (MenuHandle theMenu, short item,
                             short iconIndex);
```

```
pascal void GetItemCmd      (MenuHandle theMenu, short item, short
                             *cmdChar);
pascal void SetItemCmd      (MenuHandle theMenu, short item, short cmdChar);
```

Disposing of Menus

```
pascal void DisposeMenu     (MenuHandle theMenu);
```

Counting the Items in a Menu

```
pascal short CountMItems    (MenuHandle theMenu);
```

Highlighting the Menu Bar

```
pascal void FlashMenuBar    (short menuID);
pascal void SetMenuFlash    (short count);
```

Recalculating Menu Dimensions

```
pascal void CalcMenuSize    (MenuHandle theMenu);
```

Managing Entries in the Menu Color Information Table

```
{some routines have two spellings, see Table 3-8 for the alternate spelling}
pascal MCTableHandle GetMCInfo(void);
pascal void SetMCInfo       (MCTableHandle menuCTbl);
pascal void DisposeMCInfo   (MCTableHandle menuCTbl);
pascal MCEntryPtr GetMCEntry(short menuID, short menuItem);
pascal void SetMCEntries    (short numEntries, MCTablePtr menuCEntries);
pascal void DeleteMCEntries (short menuID, short menuItem);
```

Application-Defined Routine

```
pascal void MyMenuDef       (short message, MenuHandle theMenu,
                             Rect *menuRect, Point hitPt,
                             short *whichItem);
```

Assembly-Language Summary

Data Structures

The Menu Information Data Structure

0	`menuID`	word	number that identifies the menu
2	`menuWidth`	word	width (in pixels) of the menu
4	`menuHeight`	word	height (in pixels) of the menu
6	`menuDefHandle`	long	menu definition procedure
10	`menuEnable`	long	enable flags
14	`menuData`	256 bytes	menu title followed by menu item information

Global Variables

`AtMenuBottom`	The pixel value at the bottom of the scrollable menu.
`MBarEnable`	Contains 0 if all menus in the current menu bar belong to an application; contains a nonzero value if all menus belong to a desk accessory.
`MBarHeight`	Contains current height of the menu bar, in pixels.
`MBarHook`	Address of routine that `MenuSelect` calls repeatedly while the mouse button is down.
`MenuCInfo`	Contains a handle to application's menu color information table.
`MenuDisable`	Contains the menu ID and item number of the last item chosen, regardless of whether the item was disabled or enabled.
`MenuFlash`	Contains the current count (number of times) a menu item blinks when chosen by the user.
`MenuHook`	Address of routine that `MenuSelect` calls after a menu title is highlighted and the menu rectangle is calculated but before the menu is drawn.
`TheMenu`	Contains the menu ID of the highlighted menu in the menu bar.
`TopMenuItem`	The pixel value at the top of the scrollable menu.

Result Codes

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough room in heap zone
`resNotFound`	–192	Unable to read resource
`hmHelpManagerNotInited`	–855	Help menu not set up

CHAPTER 4

Window Manager

Contents

This chapter describes how your application can use the Window Manager to create and manage windows.

A Macintosh application uses windows for most communication with the user, from discrete interactions like presenting and acknowledging alert boxes to open-ended interactions like creating and editing documents. Users generally type words and formulas, draw pictures, or otherwise enter data in a window on the screen. Your application typically lets the user save this data in a file, open saved files, and view the saved data in a window. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for more information about handling files.

A window can be any size or shape, and the user can display any number of windows, within the limits of available memory, on the screen at once.

The Window Manager defines a set of standard windows and provides a set of routines for managing them. The Window Manager helps your application display windows that are consistent with the Macintosh user interface. See *Macintosh Human Interface Guidelines* for a detailed description of windows and their behavior.

You typically store information about your windows in resources. This chapter describes the standard window resources. For general information on resources, see the chapter "Introduction to the Macintosh Toolbox" in this book. For information on Resource Manager routines, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox.*

The Window Manager itself depends on QuickDraw, the part of the Macintosh system software that handles quick manipulation of graphics. QuickDraw supports drawing into graphics ports, which are individual and complete drawing environments with independent coordinate systems. Each window represents a graphics port, which is described in *Inside Macintosh: Imaging.*

To maintain its windows, your application needs to know what actions the user is taking on the desktop. It receives this information through events**,** which are messages that describe user actions and report on the processing status of your application. This chapter describes the events that affect window display and considers mouse-down and keyboard events as they relate to windows. For a complete description of events and how your application handles them, see the chapter "Event Manager" in this book.

Most document windows contain controls**,** which are screen images the user manipulates to control the display or the behavior of the application. This chapter illustrates the controls most commonly used in windows. For more information on creating and responding to controls, see the chapter "Control Manager" in this book.

You use the Window Manager to create and display a new window when the user creates a new document or opens an existing document. When the user clicks or holds down the mouse button while the cursor is in a window created by your application, you use the Window Manager to determine the location of the mouse action and to alter the window display as appropriate. When the user closes a window, you use the Window Manager to remove the window from the screen.

This chapter describes how the Window Manager supports windows and then explains how you can use the Window Manager to

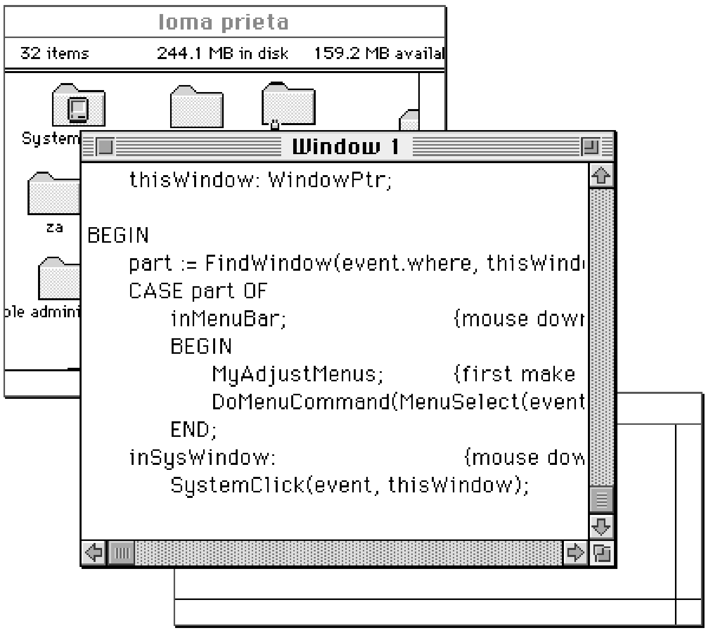
- create and display windows
- handle events in windows
- change the display when the user moves or resizes windows
- remove windows

Introduction to Windows

A **window** is a user interface element, an area on the screen in which the user can enter or view information.

The user can have multiple windows on the desktop at once, from a number of different applications. The user can change the size and location of most windows and can place windows entirely or partially in front of other windows. Figure 4-1 shows a few windows on the desktop.

Figure 4-1 Multiple windows



```
    thisWindow: WindowPtr;

BEGIN
    part := FindWindow(event.where, thisWind
    CASE part OF
        inMenuBar;                  {mouse dowr
        BEGIN
            MyAdjustMenus;          {first make
            DoMenuCommand(MenuSelect(event
        END;
    inSysWindow:                    {mouse dow
        SystemClick(event, thisWindow);
```

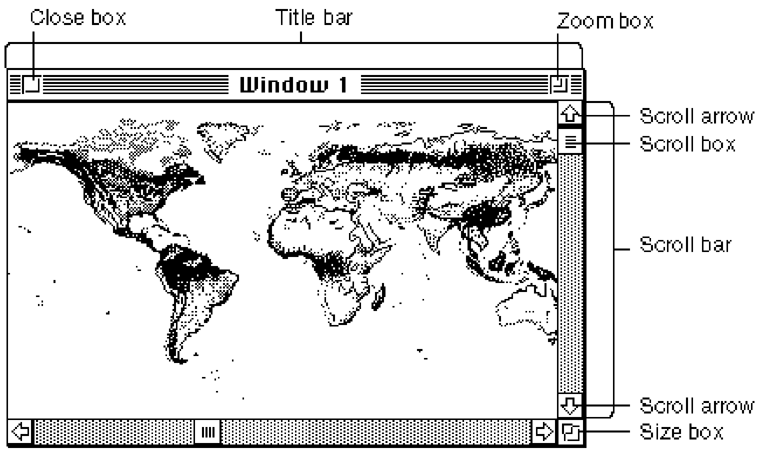
Your application typically creates **document windows** that allow the user to enter and display text, graphics, or other information. For an illustration of a document window in full color, see Plate 1 at the beginning of this book.

A document window is a view into the document—if the document is larger than the window, the window is a view of a portion of the document. Your application can put one or more windows on the screen, each window showing a view of a document or of auxiliary information used to process the document.

The Window Manager defines and supports a set of standard window elements through which the user can manipulate windows. It's important that your application follow the standard conventions for drawing, moving, resizing, and closing windows. By presenting the standard interface, you make experienced users instantly familiar with many aspects of your application, allowing them to focus on learning its unique features.

Figure 4-2 illustrates a standard document window and its elements.

Figure 4-2 A document window



The **title bar** displays the name of the window and indicates whether it's active or not. The Window Manager displays the title of the window in the center of the title bar, in the system font and system font size. If the system font is in the Roman script system, the title bar is 20 pixels high.

When the user creates a new document, you ordinarily display a new document window with the title "untitled", spelled in lowercase letters. If the user creates a second new document window without saving the first, you title the second window "untitled 2", with a space between the word and the number. Continue to add 1 to the number in the title as long as the user continues to create new windows without saving previously numbered, untitled windows.

When the user opens a saved document, you assign the document's filename to the window in which it is displayed.

The user expects to move a window by dragging it by its title bar. You can support moving the window by calling the Window Manager's `DragWindow` procedure, as described in "Moving a Window" on page 4-53.

The **close box** offers the user a quick way to close a window. You can use the `TrackGoAway` function to track mouse activity in the close box and the `CloseWindow` and `DisposeWindow` procedures to close windows. Closing windows is described in "Closing a Window" beginning on page 4-60.

The **zoom box** offers the user a quick way to switch between two different window sizes. You use the `TrackBox` function to track mouse activity in the zoom box and the `ZoomWindow` procedure to zoom windows. Zooming windows is described in "Zooming a Window" beginning on page 4-53.

The **size box** lets the user change the size and dimensions of the window. You use the `GrowWindow` function to track mouse activity in the size box and the `SizeWindow` procedure to resize windows. Sizing windows is described in "Resizing a Window" beginning on page 4-57.

The **scroll bars** let the user see different parts of a document that contains more information than can be displayed at once in the window. Although the Macintosh user interface guidelines specify that you place scroll bars on the right and lower edges of a window that needs them, scroll bars are not part of the window structure. You create and control the scroll bars through the Control Manager, described in the chapter "Control Manager" in this book.

The **content region** is the part of the window in which your application displays the contents of a document, the size box, and the window controls.

The window **frame** is the part of the window drawn automatically by the Window Manager—the title bar, including the close box and zoom box, and the window's outline.

The **structure region** is the entire screen area occupied by a window, including the frame and content region. (See Figure 4-10 on page 4-12.)

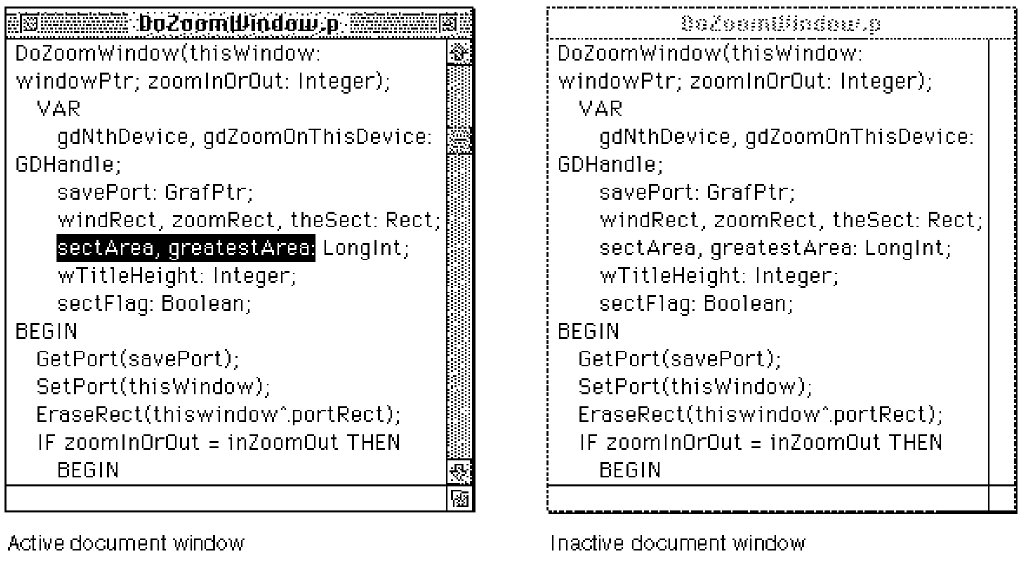
Active and Inactive Windows

The window in which the user is currently working is the **active window.** The active window is the frontmost window on the desktop. It is identified visually by the "racing stripes" in its title bar.

The active window is the target of keyboard activity. It often contains a blinking insertion point (also called the caret) marking the place where new text or graphics will appear. When the user selects text in an active window, your application should highlight the text with inverse video; if the window becomes inactive, you remove the highlighting. You can use a secondary selection technique, such as an outline, to mark a selection in an inactive window. You display scroll bars only in the active window. Figure 4-3 illustrates a sample document window in active and inactive states.

Except for the active window, all document windows on the desktop, whether they belong to your application or another, are inactive. Your application can process documents in inactive windows, but only the active window interacts with the user. For example, if the user chooses Save from the File menu, your application saves only the document in the active window.

Figure 4-3 Active and inactive document windows



```
DoZoomWindow(thisWindow:
windowPtr; zoomInOrOut: Integer);
  VAR
    gdNthDevice, gdZoomOnThisDevice:
GDHandle;
    savePort: GrafPtr;
    windRect, zoomRect, theSect: Rect;
    sectArea, greatestArea: LongInt;
    wTitleHeight: Integer;
    sectFlag: Boolean;
BEGIN
  GetPort(savePort);
  SetPort(thisWindow);
  EraseRect(thiswindow^.portRect);
  IF zoomInOrOut = inZoomOut THEN
    BEGIN
```

```
DoZoomWindow(thisWindow:
windowPtr; zoomInOrOut: Integer);
  VAR
    gdNthDevice, gdZoomOnThisDevice:
GDHandle;
    savePort: GrafPtr;
    windRect, zoomRect, theSect: Rect;
    sectArea, greatestArea: LongInt;
    wTitleHeight: Integer;
    sectFlag: Boolean;
BEGIN
  GetPort(savePort);
  SetPort(thisWindow);
  EraseRect(thiswindow^.portRect);
  IF zoomInOrOut = inZoomOut THEN
    BEGIN
```

To make a window active, the user clicks anywhere in its contents or frame. When the user activates one of your windows, you call the Window Manager to highlight the window frame and title bar; you activate the controls and window contents. As a window becomes active, it appears to the user to move forward, in front of all other windows.

When the user clicks in an inactive document window, you should make the window active but not make any selections in the window in response to the click. To make a selection in the window, the user must click again. This behavior protects the user from losing an existing selection unintentionally when activating a window.

Note

The Finder makes selections in response to the first click in an inactive window, because this action is more natural for the way Finder windows are used. You might find that users expect the first click to cause a selection in some other special-purpose windows created by your application. This behavior is seldom appropriate in document windows. u

When a window that belongs to your application becomes inactive, the Window Manager redraws the frame, removing the highlighting from the title bar and hiding the close and zoom boxes. Your application hides the controls and the size box and removes highlighting from application-controlled elements.

When the user reactivates a window, reinstate the window as it was before it was deactivated. Draw the scroll box in the same position and restore the insertion point or highlight the previous selection.

Types of Windows

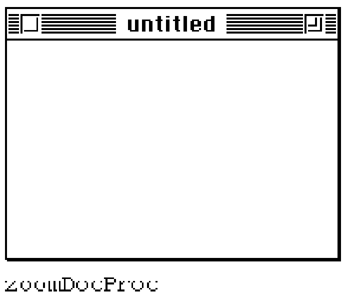
Because windows have so many uses, their appearances vary. The Window Manager defines a number of **window types** that meet the basic needs of most applications. A window type is the general description of how a window looks and behaves. Some windows have title bars and others don't, for example, and windows can have almost any combination of the window-manipulation elements: close box, zoom box, and size box.

This section describes the nine basic window types supported by the Window Manager and their uses. You can create windows of these types by specifying one of the window type constants: `zoomDocProc`, `dBoxProc`, `altDBoxProc`, `plainDBoxProc`, `movableDBoxProc`, `noGrowDocProc`, `documentProc`, `zoomNoGrow`, and `rDocProc`. For instructions for creating windows, see "Creating a Window" beginning on page 4-25.

To give the user maximum flexibility and control, you can use the `zoomDocProc` window type for your document windows. A `zoomDocProc` window supports all of the window-manipulation elements shown in Figure 4-2 on page 4-5: title bar, close box, zoom box, and size box. The Window Manager does not necessarily draw the close box and size box, however. You must call the Window Manager's `DrawGrowIcon` procedure to draw the size box, and you can optionally suppress the close box when you create the window. For more information on defining a window's characteristics, see "Creating a Window" beginning on page 4-25.

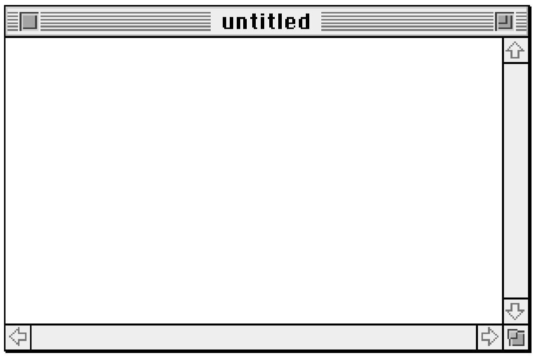
Figure 4-4 illustrates a window of type `zoomDocProc` with a close box, as drawn by the Window Manager before you add the size box and scroll bars.

Figure 4-4 A window of type `zoomDocProc`



In most cases, a window of type `zoomDocProc` should contain both a close box and a size box. When the related document contains more data than fits in the window, you activate the scroll bars and adjust them to show where in the document the user is working. Figure 4-5 illustrates a window of type `zoomDocProc` with a size box and scroll bars.

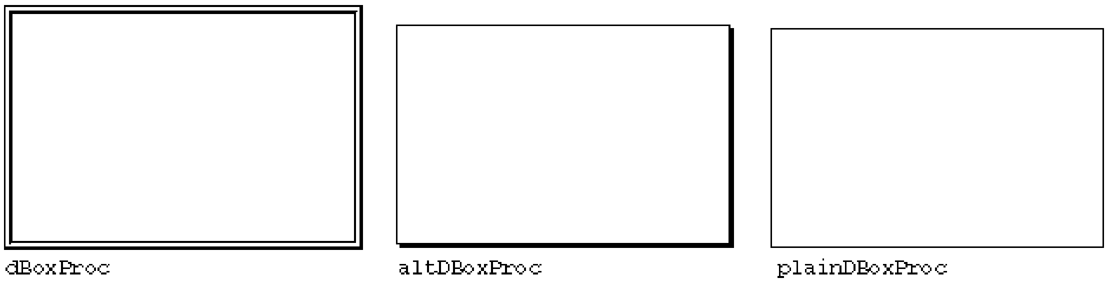
Figure 4-5 A window of type `zoomDocProc`, with size box and inactive scroll bars



You also use windows to display alert boxes and dialog boxes. This section describes the window types used for alert boxes and dialog boxes. For more thorough descriptions of the different kinds of alert boxes and dialog boxes, see the chapter "Dialog Manager" in this book.

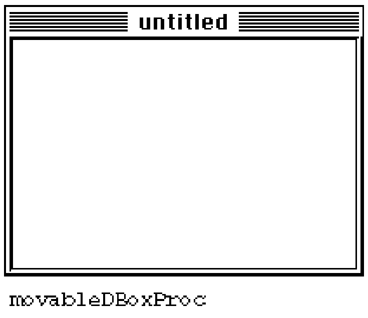
Alert boxes and fixed-position modal dialog boxes contain no window-manipulation elements. The user cannot move, resize, zoom, or close them manually. An alert box or a modal dialog box remains on the screen as the active window until the Dialog Manager or your application removes it—usually when the user completes the interaction by clicking one of the buttons. Figure 4-6 illustrates the three window types available for alert boxes and fixed-position modal dialog boxes.

Figure 4-6 Window types for alert boxes and fixed-position modal dialog boxes



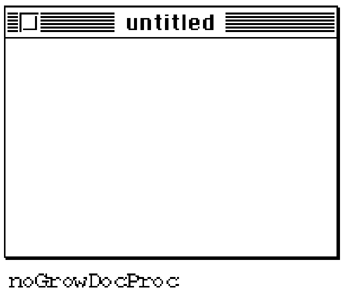
When you want to let the user move a modal dialog box window—in order, for example, to see text that might be obscured by the window—you can implement a movable modal dialog box. A movable modal dialog box cannot be resized, closed, or zoomed, but it can be moved. Figure 4-7 on the next page illustrates the `movableDBoxProc` window type. Like a fixed-position modal dialog box, the movable modal dialog box remains active until the user completes the dialog.

Figure 4-7 A window of type `movableDBoxProc`



Whenever possible, avoid modal dialog boxes and instead use modeless dialog boxes, which allow the user to perform other tasks without dismissing the dialog box. Windows of type `noGrowDocProc`, used for displaying modeless dialog boxes, can be moved or closed but not resized or zoomed. You can implement modeless dialog boxes with other window types if necessary, but it's easier to conform to the user interface guidelines if you keep your dialog box windows as simple as possible. Figure 4-8 illustrates the modeless dialog box window.

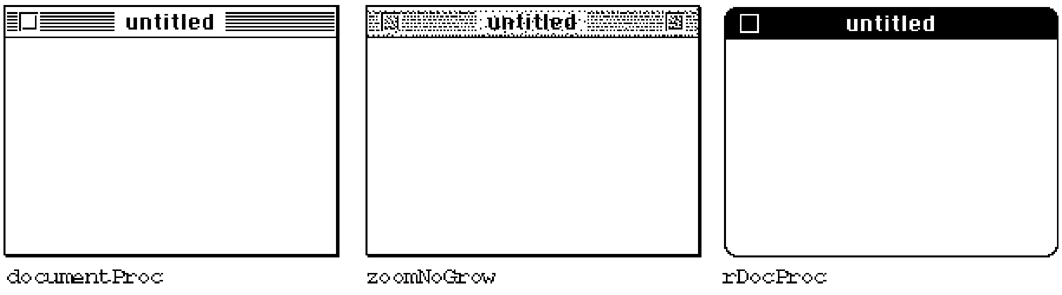
Figure 4-8 A window of type `noGrowDocProc`



The Window Manager also supports a few window types that are seldom used. The `documentProc` window type, for example, has a title bar and supports a close box and size box but no zoom box. The `zoomNoGrow` window type is virtually never appropriate: `zoomNoGrow` supports a close box and a zoom box, but not a size box. The `rDocProc` window type is a rounded-corner window with a title bar and a close box; it is used by desk accessories. Figure 4-9 illustrates these three seldom-used window types.

The **window definition function** defines the general appearance and behavior of a window. The system software and various Window Manager routines call a window's window definition function when they need to perform certain window-dependent actions, such as drawing or resizing a window's frame.

Figure 4-9 Seldom-used window types



The Window Manager supplies two standard window definition functions that handle the nine standard window types. A window definition function draws the window's frame, draws the close box and window title (if any), determines which region the cursor is in within the window, calculates the window's structure and content regions, draws the window's zoom box (if any), draws the window's size box (if any), and performs any special initialization or disposal tasks.

A single window definition function can support up to 16 different window types. The window definition function defines a **variation code,** an integer from 0 through 15, for each window type it supports.

A **window definition ID** is a single value incorporating both the window's definition function and its variation code. (The resource ID of the window definition function is stored in the upper 12 bits of the integer, and the variation code is stored in the lower 4 bits.) The window-type constants described in this section are in fact window definition IDs.

Constant	Window definition ID	Description
`documentProc`	0	movable, sizable window, no zoom box
`dBoxProc`	1	alert box or modal dialog box
`plainDBox`	2	plain box
`altDBoxProc`	3	plain box with shadow
`noGrowDocProc`	4	movable window, no size box or zoom box
`movableDBoxProc`	5	movable modal dialog box
`zoomDocProc`	8	standard document window
`zoomNoGrow`	12	zoomable, nonresizable window
`rDocProc`	16	rounded-corner window

You can provide your own window definition function if you need a window with unusual characteristics, as described in "The Window Definition Function" beginning on page 4-120. Always be careful to conform window behavior to the guidelines in *Macintosh Human Interface Guidelines.*

Window Regions

The Window Manager recognizes a number of different special-purpose **window regions,** which are defined by either the Window Manager or the window definition functions.

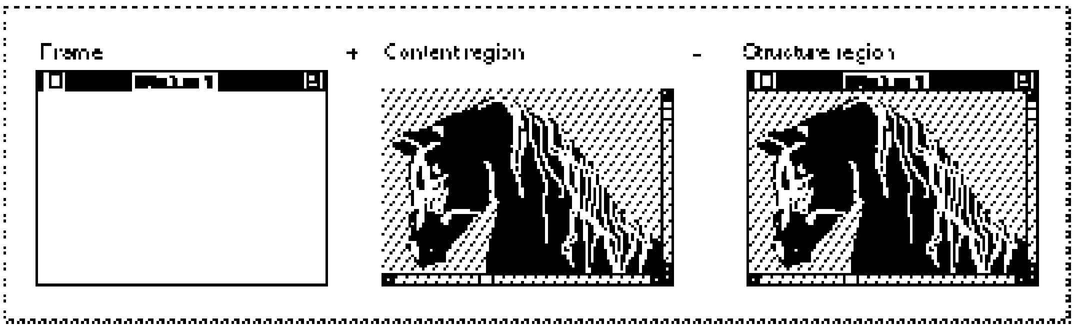
The most obvious window regions are the parts of the visible window that the user manipulates to control the display. These window regions correspond to the standard window parts. The **drag region** is the area occupied by the title bar, except for the close box and zoom box. (The user moves the window by dragging it by its title bar.) The **size region, close region,** and **zoom region** are the areas occupied by the size box, close box, and zoom box, respectively.

When the user presses the mouse button while the cursor is in one of your windows, you use the Window Manager function `FindWindow` to determine the region in which the mouse-down event occurred. (The `FindWindow` function calls the window's window definition function, which defines and interprets the window-manipulation regions.) Depending on the result, you then call the appropriate Window Manager routine or your own routine for handling the event. For more information about determining where the cursor is when the user presses the mouse button, see "Handling Mouse Events in Windows" on page 4-42. For discussions of how to use the Window Manager routines for moving, sizing, closing, and zooming windows, see "Moving a Window" beginning on page 4-53 and the sections that follow it.

The Window Manager also makes a broad distinction between the parts of the window it draws automatically and the parts drawn by your application. The Window Manager draws the window frame—the title bar, including the close box and zoom box, and the window's outline. (The Window Manager also draws the size box, but only when your application calls the `DrawGrowIcon` procedure.) Your application is responsible for drawing the content region—that is, the part of the window in which the contents of a document, the size box, and the window controls (including the scroll bars) are displayed.

The entire screen area occupied by a window, including the window outline, title bar, and content region, is the structure region. Figure 4-10 illustrates the frame, content region, and structure region of a window.

Figure 4-10 Window frame, content region, and structure region



The drawing region of a graphics port associated with a window encompasses only the window's content region.

As the user creates, moves, resizes, and closes windows on the desktop, portions of windows may be obscured and uncovered. The Window Manager keeps track of these changes, accumulating a dynamic region known as the **update region** for each window. The update region contains all areas of a window's content region that need updating. The Event Manager periodically scans the update regions of all windows on the desktop, generating update events for windows whose update regions are not empty. When your application receives an update event, it redraws the update region. Both your application and the Window Manager can manipulate a window's update region. The sections "Updating the Content Region" on page 4-40 and "Maintaining the Update Region" on page 4-41 describe how the Window Manager and your application track and use the update region.

Dialog Boxes and Alert Boxes

Macintosh applications use alert boxes and dialog boxes to give the user messages and to solicit information. A text-processing application, for example, might display an alert box telling the user that a newly inserted graphic does not fit within the page boundaries. It might display a dialog box in which the user can specify margins, tabs, and other formatting information. (The chapter "Dialog Manager" in this book explains how to use the various kinds of alert boxes and dialog boxes.)

Alert boxes and dialog boxes are merely special-purpose windows. You can handle all alert boxes and most modal dialog boxes through the Dialog Manager, which itself calls the Window Manager. You supply the Dialog Manager with lists of the items in your alert boxes and dialog boxes, and the Dialog Manager displays the windows, tells you which items the user is manipulating, and disposes of the windows when the user is done. Your application provides the code that responds to the user's selections in the alert and dialog boxes.

Although you can specify any window type for your alert boxes and modal dialog boxes, the Dialog Manager functions that handle alert boxes and modal dialog boxes do not support window manipulation. You should therefore use one of the window types without a title bar or size box, most typically the `dBoxProc` window type, for alert boxes and modal dialog boxes. (When the user is responding to a modal dialog box, mouse-down events outside the menu bar or the content region of the dialog box result only in the sounding of the system alert. Note that the Process Manager does not perform major switching while the `ModalDialog` procedure is handling events.)

You use the `movableDBox` window type for movable modal dialog boxes. As described in the chapter "Dialog Manager" in this book, your application can use the Dialog Manager to help handle events in a movable modal dialog box. Your application, however, must handle window-manipulation events—ordinarily only the moving of the movable modal dialog box window.

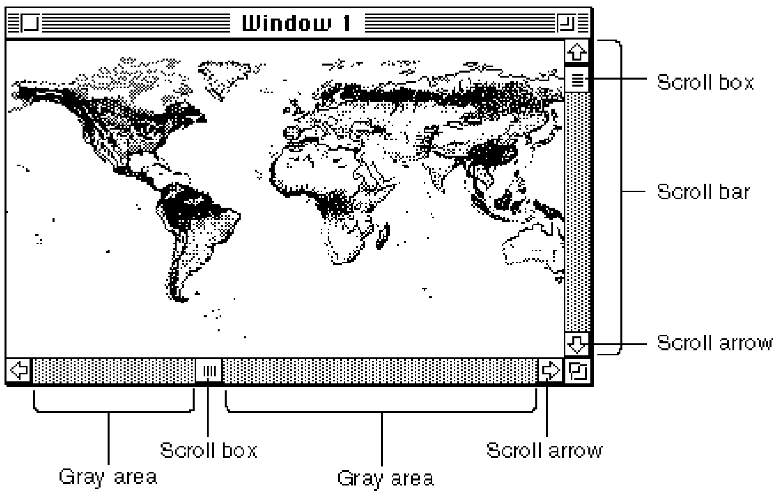
Use the `noGrowDocProc` window type for modeless dialog boxes. You typically use the Dialog Manager to handle events in a modeless dialog box, much like events in a movable modal dialog box. Your application handles window-manipulation events in modeless dialog boxes just as it handles them in document windows.

If you use complex dialog boxes, you might find it's more efficient to use the Window Manager and other parts of the Toolbox, instead of the Dialog Manager, to create and manage your own dialog box windows. Again, see the chapter "Dialog Manager" in this book for a list of characteristics to consider when evaluating the complexity of a dialog box and for examples of customized dialog boxes.

Controls

Most windows contain **controls,** which are screen images that the user manipulates to control the display or the behavior of the application. The most common control in a document window is the scroll bar, illustrated in Figure 4-11.

Figure 4-11 Scroll bars

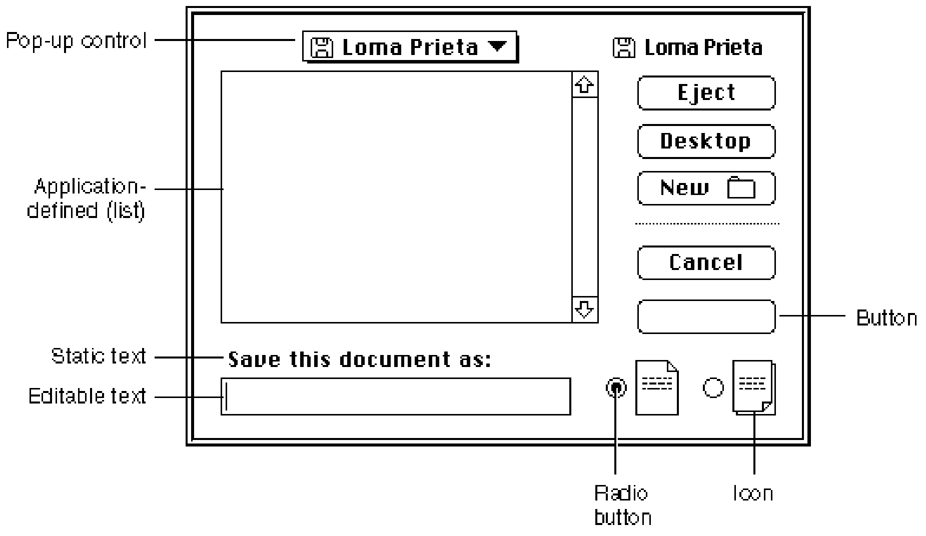


You use scroll bars to show the relative position, within the entire document, of the portion of the document displayed in the window. You should allow the user to drag the scroll box or click in the gray areas or the scroll arrows to move parts of the document into and out of the window. You activate scroll bars in a window any time there is more data than can be shown at one time in the space available.

You use the Control Manager to create, display, and manipulate the scroll bars and any other controls in your windows. Each control "belongs" to a window and is displayed within the graphics port that represents that window. For each window your application creates, the Window Manager maintains a **control list,** a series of entries pointing to the descriptions of the controls associated with the window.

Most alert boxes and dialog boxes contain **buttons,** rounded rectangles that cause an immediate or continuous action when clicked, and most dialog boxes contain additional screen images, like **radio buttons,** that display and retain settings. Figure 4-12 illustrates a dialog box with buttons, radio buttons, and a number of other controls and dialog items.

Figure 4-12 Controls in a dialog box



Buttons ordinarily appear only in alert boxes and dialog boxes. Most of the other elements illustrated in Figure 4-12 appear only in dialog boxes. If you use the Dialog Manager to create your alert boxes and dialog boxes, it draws your controls for you and lets you know when the user has clicked one of them. You can, however, call the Control Manager yourself to display and track buttons and other controls in any windows your application creates. You can also write your own control definition functions to create and control other kinds of controls. For a complete description of how to create and support controls, see the chapter “Control Manager” in this book.

Windows on the Desktop

Multiple windows, from different applications, can appear simultaneously on the desktop. The Window Manager tracks all windows, using its own private data structure called the **window list.** Entries appear in the window list in their order on the desktop, beginning with the frontmost, active window. When the user changes the ordering of windows on the desktop, the Window Manager generates events telling your application to activate, deactivate, and redraw windows as necessary. The Window Manager prevents you from drawing accidentally in the windows of other applications.

The user can interact with only one application at a time. The application with which the user is interacting (that is, the application that owns the window in which the user is working) is the active application, or **foreground process,** and the others are inactive applications, or **background processes.** One way the user can switch applications is by clicking in a window that belongs to a background process. The Process Manager then generates events telling the previously active application that it's about to be suspended and telling the newly active application that it can resume processing. (For more information about the workings of foreground and background processes and about the events that support simultaneous running of multiple applications, see the chapter "Event Manager" in this book.)

Your application is likely to have multiple windows on the desktop at once: one or more document windows, possibly one or more dialog box windows, and possibly some other special-purpose windows. The section "Managing Multiple Windows" beginning on page 4-23 suggests a technique for keeping track of multiple windows.

On the original Macintosh computer, the desktop area was limited to a single screen of known dimensions. Contemporary systems, however, can support multiple monitors of various sizes and capabilities. To place its windows in the appropriate place on the desktop, your application must pay attention to what screen space is available and where the user is working. For the rules governing window placement, see *Macintosh Human Interface Guidelines.* For techniques for managing windows on multiple screens, see "Positioning a Document Window on the Desktop" beginning on page 4-30.

The entire area of the desktop—that is, the screen area that is not occupied by the menu bar—is known as the **gray region.** The Window Manager maintains a pointer to the gray region in a global variable named `GrayRgn`; you can retrieve a pointer to the gray region with the Window Manager function `GetGrayRgn`.

About the Window Manager

The Window Manager provides a complete set of routines for creating, moving, resizing, and otherwise manipulating windows. It also provides lower-level support by managing the layering of windows on the desktop and by alerting your application to desktop changes that affect its windows. Your application and the Window Manager work together to provide the user with a consistent window interface.

When, for example, the user presses the mouse button while the cursor is in the drag region of a window's title bar, you can call the `DragWindow` procedure, which moves a dotted outline of the window around the screen in response to mouse movements. When the user releases the mouse button, `DragWindow` calls the `MoveWindow` procedure, which redraws the window in its new location. If part or all of an inactive window belonging to your application is exposed by the move, the Window Manager triggers an update event that tells your application to redraw the exposed region.

Similarly, if the user clicks in an inactive window, you can call the `SelectWindow` procedure. `SelectWindow` adjusts the window highlighting and layering and also generates activate events that tell your application which windows to activate and deactivate.

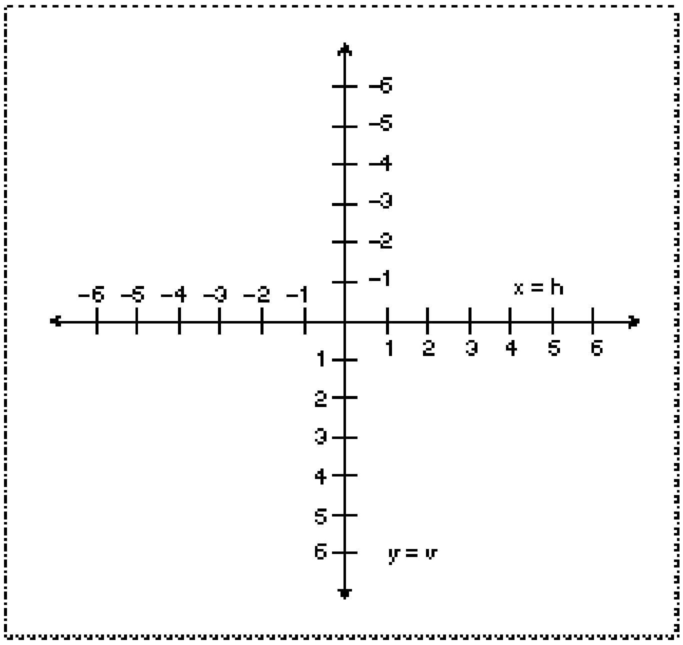
The Window Manager has built-in support for the nine basic window types described in “Types of Windows” beginning on page 4-8. When you are using one of these window types, the Window Manager draws the window’s frame, determines what region of the window the cursor is in, calculates the window’s structure and content regions, draws the window’s size box, draws the window’s close box and zoom box, and performs any special initialization or disposal tasks. If necessary, you can write your own window definition function to handle other types of windows.

Graphics Ports

Each window represents a QuickDraw **graphics port,** which is a drawing environment with its own coordinate system. (See *Inside Macintosh: Imaging* for a complete description of graphics ports and coordinate systems.) When you create a window, the Window Manager creates a graphics port in which the window’s contents are displayed.

The location of a window on the screen is defined in **global coordinates**—that is, coordinates that reflect the entire potential drawing space. QuickDraw and Color QuickDraw recognize a coordinate plane whose origin is the upper-left corner of the main screen, whose positive x-axis extends rightward, and whose positive y-axis extends downward. In QuickDraw, the horizontal offset is ordinarily labeled `h`, and the vertical offset `v`. Figure 4-13 illustrates the QuickDraw global coordinate system.

Figure 4-13 The QuickDraw global coordinate plane



Note
The orientation of the vertical axis, while convenient for computer graphics, differs from mathematical convention. Also, the coordinate plane is bounded by the limits of QuickDraw coordinates, which range from –32,768 to 32,767.

QuickDraw stores points and rectangles in its own data structures of type `Point` and `Rect`. In these structures, the vertical coordinate (`v`) appears first, followed by the horizontal coordinate (`h`). Most, but not all, QuickDraw routines that handle points require you to specify the coordinates in this order. u

When QuickDraw creates a new graphics port (usually because you've created a new window through the Window Manager), it defines a bounding rectangle for the port, in global coordinates. Ordinarily, the bounding rectangle represents the entire area of the screen on which the window appears. The bounding rectangle is stored in the graphics port data structure, in the `bounds` field of a structure called a pixel map in Color QuickDraw and a bitmap in QuickDraw.

The graphics port data structure also includes a field called `portRect`, which defines a rectangle to be used for drawing. In a graphics port that represents a window, the `portRect` rectangle represents the window's content region.

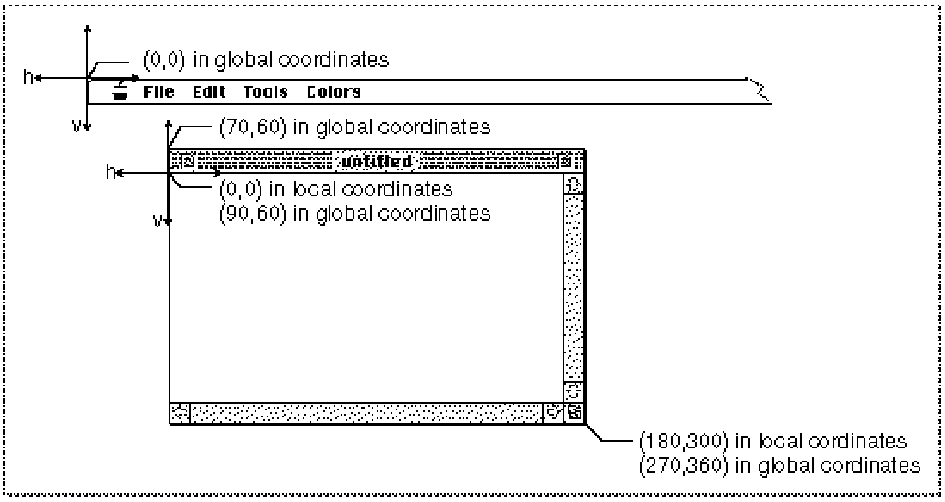
Note
When you place a window on the screen, you specify the location of its content region, in global coordinates. Remember to allow space for the window's title bar. On the main screen, remember to leave space for the menu bar. In the Roman script system, both the standard document title bar and the menu bar are 20 pixels high. You can determine the height of the menu bar with the Menu Manager `GetMBarHeight` function. You can calculate the height of the title bar by comparing the top of the window's structure region with the top of the window's content region. See Listing 4-12 on page 4-55 for a sample procedure that considers the menu bar and title bar when placing a window on the screen. u

Within the port rectangle, the drawing area is described in **local coordinates**—that is, in the coordinate system defined by the port rectangle. You draw into a window in local coordinates, without regard to the window's location on the screen (which is described in global coordinates). Figure 4-14 illustrates the local and global coordinate systems for a sample window 180 pixels high by 300 pixels wide, placed with its content region 70 pixels down and 60 pixels to the right of the upper-left corner of the screen.

When the Window Manager creates a window, it places the origin of the local coordinate system at the upper-left corner of the window's port rectangle. You can redefine the coordinates of the port rectangle's upper-left corner with the QuickDraw procedure `SetOrigin`.

The Event Manager describes mouse events in global coordinates, and you do most of your window manipulation in global coordinates. You generally display user data and manipulate your controls in local coordinates. When you need to convert between the two, you can call the QuickDraw functions `GlobalToLocal` and `LocalToGlobal`, described in *Inside Macintosh: Imaging.*

Figure 4-14 A window's local and global coordinate systems


(0,0) in global coordinates
h
v
File Edit Tools Colors
(70,60) in global coordinates
untitled
h
v
(0,0) in local coordinates
(90,60) in global coordinates
(180,300) in local coordinates
(270,360) in global cordinates

Window Records

Each window has a number of descriptive characteristics such as a title, control list, and visibility status. The Window Manager stores this information in a **window record,** which is a data structure of type `WindowRecord`.

The window record includes

- the window's graphics port data structure
- the window's class, which specifies whether it was created directly through the Window Manager or indirectly through the Dialog Manager
- the window title
- a series of flags that specify whether the window is visible, whether it's highlighted, whether it has a zoom box, and whether it has a close box
- pointers to the structure, content, and update regions
- a handle to the window's definition function
- a handle to the window's control list
- an optional handle to a picture of the window's contents
- a reference constant field that your application can use as needed

The window record is described in detail in "The Color Window Record" beginning on page 4-65.

The first field in the window record is the window's graphics port. The `WindowPtr` data type is therefore defined as a pointer to a graphics port.

```
TYPE  WindowPtr   =  GrafPtr;
```

You draw into a window by drawing into its graphics port, passing a window pointer to the QuickDraw drawing routines. You also pass window pointers to most Window Manager routines.

You don't usually need to access or directly modify fields in a window record. When you do, however, you can refer to them through the `WindowPeek` data type, which is a pointer to a window record.

```
TYPE  WindowPeek  =  ^WindowRecord;
```

The close box, drag region, zoom box, and size box are not included in the window record because they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are.

Your application seldom accesses a window record directly; the Window Manager automatically updates the window record when you make any changes to the window, such as changing its title. The Window Manager also supplies routines for changing and reading some parts of the window record.

Color Windows

Since the introduction of Color QuickDraw, the Window Manager has supported color windows. Color windows are displayed in color graphics ports, as described in *Inside Macintosh: Imaging.* The color window record is exactly like the window record described in "Window Records" on page 4-19, except that it contains a color graphics port instead of a monochrome graphics port.

Whether or not your application uses color explicitly, and whether or not a color monitor is currently installed, your application should work with color windows whenever Color QuickDraw is available. Once you have created a window, you can use the window record and window pointer for a color window interchangeably with the window record and window pointer for a monochrome window.

On a monitor that is set to display 4-bit color or greater, the Window Manager automatically displays the window title and parts of the frame and controls in color (or gray scale, depending on the capabilities of the monitor). The user can adjust these colors through the Color control panel. Except in unusual circumstances, your application should not try to change the colors of the window frame. On a monitor that's set to display 1-bit color, the Window Manager draws the window title, frame, and controls in black and white.

Various elements of a window's colors are controlled by the **window color table,** which contains a series of part codes for different window elements and the RGB values associated with each part.

Because the user can select window display colors for the entire desktop, and because the Window Manager performs some complex display calculations automatically if you don't override it, your application typically uses the default system window color table.

If your application explicitly controls the colors used in a window, however, you can define your own window color tables.

You define a window color table for a window by providing a window color table resource (that is, a resource of type `'wctb'`) with the same resource ID as the window's `'WIND'` resource. The Window Manager creates a window color table when it creates the window record. The Window Manager maintains its own linked list, using **auxiliary window records,** which associates your application's windows with their corresponding window color tables. The window color table and the auxiliary window record are described in "The Window Color Table Record" beginning on page 4-71 and "The Auxiliary Window Record" beginning on page 4-73.

Except in unusual circumstances, your application doesn't need to manipulate window color tables or the auxiliary window record.

For compatibility with other applications in the shared environment, your application should not manipulate system color tables directly but should use the Palette Manager, as described in *Inside Macintosh: Imaging.* If your application provides its own window and control definition functions, they should apply the user's desktop color choices just as the default definition functions do.

Events in Windows

Events are messages that describe user actions and report on the processing status of your application. The Window Manager generates two kinds of events: activate events and update events. Activate events tell your application that a specified window is becoming active or inactive. Update events tell your application that it must redraw part or all of a window's content region. The section "Handling Events in Windows" beginning on page 4-41 describes when these events occur and how your application responds.

One of the basic functions of the Window Manager is to report where the cursor is when your application receives a mouse-down event. The Window Manager function `FindWindow` tells your application whether the cursor is in a window and, if it's in a window, which window it's in and where in that window (that is, the title bar, the drag region, and so on). You can use the `FindWindow` function as a first filter for mouse-down events, separating events that merely affect the window display from events that manipulate user data.

The Window Manager also provides a set of routines that help you implement the standard window-manipulation conventions:

User action	Application response
Dragging the title bar	Moves the window
Dragging the size box	Resizes the window
Clicking the zoom box	Toggles the window between two sizes and locations, known as the *user state* and the *standard state*
Clicking the close box	Closes the window

The next section, “Using the Window Manager,” describes how you can use the Window Manager to move, resize, zoom, and close windows.

You can call the Control Manager to handle events in window controls, as described in the chapter “Control Manager” in this book. If you use the Dialog Manager for your alert boxes and modal dialog boxes, the Dialog Manager handles keyboard activity and mouse events in these windows. You can also use the Dialog Manager to handle keyboard activity and mouse events in the content region of movable modal dialog boxes and modeless dialog boxes. Your application, however, must handle mouse events in the title bar and close box of a movable modal or modeless dialog box.

When your application is active, a mouse-down event in a window belonging to any other application, including the Finder, switches your application to the background (unless there’s an alert box or a modal dialog box pending, in which case the Dialog Manager merely sounds the system alert).

Using the Window Manager

Virtually every Macintosh application uses the Window Manager, both to simplify the display and management of windows and to retrieve basic information about user activities.

Your application works in conjunction with the Window Manager to present the standard user interface for windows. When the user clicks in an inactive window belonging to your application, for example, you can call the procedure `SelectWindow`, which highlights the newly active window, removes the highlighting from the previously active window, and generates the activate events that trigger the activation and deactivation of the two affected windows.

Your application can also use Window Manager routines to handle direct window manipulation. For example, if the user presses the mouse button when the cursor is in the title bar of a window, you can call the `DragWindow` procedure to track the mouse and drag an outline of the window on the screen until the user releases the mouse button.

You typically create windows from window resources, which are resources of type `'WIND'`. The Window Manager supports the nine types of windows described in “Types of Windows” beginning on page 4-8. (You can also write your own window definition functions to support your own window types. Window definition functions are stored as resources of type `'WDEF'`.) Alert box windows and dialog box windows use alert (`'ALRT'`), dialog (`'DLOG'`), and item list (`'DITL'`) resources; the chapter “Dialog Manager” describes how to create these resources. Most windows contain controls, which are defined through control (`'CNTL'`) resources; the chapter “Control Manager” describes how to create control resources.

Your application typically uses the Window Manager in conjunction with both the Control Manager and the Dialog Manager. You use the Control Manager to define, draw, and manipulate controls in your windows. If your window includes scroll bars, for example, you can use the `TrackControl` function to track the mouse while the user drags the scroll box. You can use the Dialog Manager to create, display, and track events in alert boxes and dialog boxes.

System 7 provides help balloons for the window frame—that is, the title bar, zoom box, and close box—of a window created with one of the standard window definition functions. You should provide help balloons for your window content region—that is, the size box, controls, and data area—and for the window frames of any window types you define. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for a description of how to use help balloons.

Before using the Window Manager, you must call the procedure `InitGraf` to initialize QuickDraw, the procedure `InitFonts` to initialize the Font Manager, and finally the procedure `InitWindows` to initialize the Window Manager.

Managing Multiple Windows

Your application is likely to have multiple windows on the desktop at once: one or more document windows, possibly one or more dialog boxes, and possibly some special-purpose windows of your own. Only one window is active at a time, however.

When your application receives an event, it responds according to what kind of window is currently active and where the event occurred. When it receives a mouse-down event in the content region of an active document window, your application follows its own conventions: inserting text, making a selection, or adding graphics, for example. When it receives a mouse-down event in the menu bar, your application enables and disables menu items as appropriate—which again depends on what kind of window is active and what is selected in that window. If the user has the insertion point in an editable text field in a modal dialog box, for example, the only menu item available might be Paste in the Edit menu—and then only if there is something in the scrap to be pasted.

You can use various strategies for keeping track of different kinds of windows. The `refCon` field in the window record is set aside specifically for use by applications. You can use the `refCon` field to store different kinds of data, such as a number that represents a window type or a handle to a record that describes the window.

The sample code in this chapter—excerpts from the SurfWriter application used throughout this book—uses a hybrid strategy:

- For document windows, the `refCon` field holds a handle to a document record.
- For modeless or movable modal dialog boxes, the `refCon` field holds a number that represents a type of dialog box.

You may well find other approaches more practical.

The SurfWriter application stores document information about the user's data, the window display, and the file, if any, associated with the data in a document record. The document record takes this form:

```
TYPE MyDocRec =
   RECORD
      editRec:       TEHandle;      {handle to text being edited}
      vScrollBar:    ControlHandle; {control handle to the }
                                    { vertical scroll bars}
      hScrollBar:    ControlHandle; {control handle to the }
                                    { horizontal scroll bars}
      fileRefNum:    Integer;       {reference number for file}
      fileFSSpec:    FSSpec;        {FSSpec record for file}
      windowDirty:   Boolean;       {whether data has changed }
                                    { since last save}
   END;
   MyDocRecPtr       = ^MyDocRec;
   MyDocRecHnd       = ^MyDocRecPtr;
```

The SurfWriter application creates a document record every time it creates a document window, and it stores a handle to the document record in the `refCon` field of the window record. (See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for a more complete illustration of how to use document records.)

When SurfWriter creates a modeless dialog box or a movable modal dialog box, it stores a constant that represents that dialog box (that is, it specifies the constant in the dialog resource, and the Window Manager sets the `refCon` field to that value when it creates the window record). For example, a `refCon` value of 20 might specify a modeless dialog box that accepts input for the Find command, and a value of 21 might specify a modeless dialog box that accepts input for the spelling checker.

When SurfWriter receives notification of an event in one of its windows, it first determines the function of the window and then dispatches the event as appropriate. Listing 4-1 illustrates an application-defined routine `MyGetWindowType` that determines the window's type.

Note

The `MyGetWindowType` function determines the type of a window from among a set of application-defined window types, which reflect the different kinds of windows the application creates. These window types are different from the standard window types defined by the definition functions, which determine how windows look and behave. To find out which one of the standard window types a window is, call the Window Manager function `GetWVariant`. u

The sample code later in this chapter calls the `MyGetWindowType` function as part of its event-handling procedure, described in the section "Handling Events in Windows" beginning on page 4-41.

Listing 4-1 Determining the window type

```
FUNCTION MyGetWindowType (thisWindow: WindowPtr): Integer;
VAR
   myWindowType:  Integer;
BEGIN
  IF thisWindow <> NIL THEN
  BEGIN
    myWindowType := WindowPeek(thisWindow)^.windowKind;
    IF myWindowType < 0 THEN                {window belongs to }
      MyGetWindowType := kDAWindow          { a desk accessory}
    ELSE
      IF myWindowType = userKind THEN       {document window}
        MyGetWindowType := kMyDocWindow
      ELSE                                  {dialog window}
         MyGetWindowType := GetWRefCon(window);    {get dialog ID}
  END
  ELSE
   MyGetWindowType := kNil;
END;
```

Notice that `MyGetWindowType` checks whether the window belongs to a desk accessory. This step ensures compatibility with older versions of system software. When your application is running in System 7, it should receive events only for its own windows and for windows belonging to desk accessories that were launched in its partition. See *Inside Macintosh: Memory* for information about partitions and *Inside Macintosh: Processes* for information about launching applications and desk accessories.

Creating a Window

You typically specify the characteristics of your windows—such as their initial size, location, title, and type—in window (`'WIND'`) resources. Once you have defined your window resources, you can call the function `GetNewCWindow` (or `GetNewWindow`) to create windows.

Defining a Window Resource

You typically define a window resource for each type of window that your application creates. If, for example, your application creates both document windows and special-purpose windows, you would probably define two window resources. Defining your windows in window resources lets you localize your window titles for different languages by changing only the window resources. (You specify the characteristics of alert boxes and dialog boxes with the alert and dialog resources, described in the chapter “Dialog Manager” in this book.)

Listing 4-2 shows a window resource, in Rez input format, that an application might use to create a document window. The resource specifies the attributes for windows created from the resource of type 'WIND' with resource ID 128. The system software loads the resource into memory immediately after opening the resource file, and the Memory Manager can purge the memory occupied by the resource.

Listing 4-2 Rez input for a window ('WIND') resource for a document window

```
#define rDocWindow      128

resource 'WIND' (rDocWindow, preload, purgeable) {
      {64, 60, 314, 460},  /*initial window size and location*/
      zoomDocProc,         /*window definition ID: */
                           /* incorporates definition function */
                           /* and variation code*/
      invisible,           /*window is initially invisible*/
      goAway,              /*window has close box*/
      0x0,                 /*reference constant*/
      "untitled",          /*window title*/
      staggerParentWindowScreen
                           /*optional positioning specification*/
};
```

The four numbers in the first element of this resource specify the upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional positioning code described later in this section. When specifying a window's position on the desktop, remember to leave room for the window's frame and, on the main screen, for the menu bar.

The second element contains the window's definition ID, which specifies both the window definition function that will handle the window and an optional variation code that defines a window type. If you are using one of the standard window types (described in "Types of Windows" beginning on page 4-8), you need to specify only one of the window-type constants listed in "The Window Resource" beginning on page 4-124.

The third element in the window resource specifies whether the window is initially visible or invisible. This element determines only whether the Window Manager displays the window when it first creates it, not whether the window can be seen on the screen. (A window entirely covered by other windows, for example, might be "visible," even though the user cannot see it.) You typically create new windows in an invisible state, build the content area of the window, and then display the completed window by calling ShowWindow to make it visible.

The fourth element in the window resource specifies whether the window has a close box. Only some of the standard window types (`zoomDocProc`, `noGrowDocProc`, `documentProc`, `zoomNoGrow`, and `rDocProc`) support close boxes. The close-box element has no effect if the second field of the resource specifies a window type that does not support a close box. The Window Manager draws the close box when it draws the window frame.

The fifth element in the window resource is a reference constant, in which your application can store whatever data it needs. When it builds a new window record, the Window Manager stores in the `refCon` field whatever value you specify here. You can also put a placeholder here (such as `0x0`, in this example) and then set the `refCon` field yourself by calling the `SetWRefCon` procedure.

The sixth element in the window resource is a string that specifies the window title.

The optional seventh element in the window resource specifies a positioning rule that overrides the window position specified by the rectangle in the first element. In the window resource for a document window, you typically specify the positioning constant `staggerParentWindowScreen`. For a complete list of the positioning constants and their effects, see "The Window Resource" beginning on page 4-124.

The positioning constants are convenient when the user is creating a new document or when you're handling your own dialog boxes and alert boxes. When you're creating a new window to display a previously saved document, however, the new window should appear, if possible, in the same rectangle as the previous window (that is, the window used during the last save). For the rules of window placement, see "Positioning a Document Window on the Desktop" beginning on page 4-30.

Use the function `GetNewCWindow` or `GetNewWindow` to create a window from a `'WIND'` resource.

Creating a Window From a Resource

You typically create a new window every time the user creates a new document, opens a previously saved document, or issues a command that triggers a dialog box.

You create document windows from a window resource using the function `GetNewCWindow` or `GetNewWindow`. (Whenever Color QuickDraw is available, use `GetNewCWindow` to create color windows, whether or not a color monitor is currently installed. A color window record is the same size as a window record, and `GetNewCWindow` returns a pointer of type `WindowPtr`, so most code can handle color windows and monochrome windows identically.)

You can allow `GetNewCWindow` to allocate the memory for your window record. You can maintain more control over memory use, however, by allocating the memory yourself from a block allocated for such purposes during your own initialization routine, and then passing the pointer to `GetNewCWindow`.

You typically create the scroll bars from control (`'CNTL'`) resources at the time that you create a document window and then display them when you make the window visible.

Listing 4-3 illustrates an application-defined procedure, DoNewCmd, which SurfWriter calls when the user chooses New from the File menu. Windows are typically invisible when created and displayed only after all elements are in place.

Listing 4-3 Creating a new window

```
PROCEDURE DoNewCmd (newDocument: Boolean; VAR window: WindowPtr);
VAR
   myData:        MyDocRecHnd;   {the document's data record}
   windStorage:   Ptr;           {memory for window record}
   destRect,                     {rectangles for creating }
   viewRect:      Rect;          { TextEdit edit record}
   good:          Boolean;       {success flag}
BEGIN
   window := NIL;                   {no window created yet}
   good := FALSE;                   {no success yet}
   {allocate memory for window record from previously allocated block}
   windStorage := MyPtrAllocationProc;
   IF windStorage <> NIL THEN       {memory allocation succeeded}
   BEGIN                            {create window}
      IF gColorQDAvailable THEN
         window := GetNewCWindow(rDocWindow, windStorage, WindowPtr(-1))
      ELSE
         window := GetNewWindow(rDocWindow, windStorage, WindowPtr(-1));
   END;
                                    {create document record}
   myData := MyDocRecHnd(NewHandle(SIZEOF(MyDocRec)));
   IF (window <> NIL) AND (myData <> NIL) THEN  {window record and document }
   BEGIN                                        { record both allocated}
      SetPort(window);              {set current port}
      HLock(Handle(myData));        {lock handle to doc record}
      SetWRefCon(window, LongInt(myData));   {link document record to window}
      WITH window^, myData^^ DO     {fill in document record}
      BEGIN
         MyGetTERect(window, viewRect); {set up a viewRect for TextEdit}
         destRect := viewRect;
         destRect.right := destRect.left + kMaxDocWidth;
         editRec := TENew(destRect, viewRect);
         IF editRec <> NIL THEN           {it's a good edit record}
         BEGIN
            good := TRUE;                    {set success flag}
            MyAdjustViewRect(editRec);       {set up edit record}
            TEAutoView(TRUE, editRec);
         END
         ELSE
            good := FALSE;             {clear success flag}
         IF good THEN
         BEGIN                         {create scroll bars}
            vScrollBar := GetNewControl(rVScroll, window);
            hScrollBar := GetNewControl(rHScroll, window);
            good := (vScrollBar <> NIL) AND (hScrollBar <> NIL);
         END;
         IF good THEN                  {it's a good document}
         BEGIN
            MyAdjustScrollBars(window, FALSE);  {adjust scroll bars}
            fileRefNum := 0;           {no file yet}
            windowDirty := FALSE;      {no changes yet}
            IF newDocument THEN        {if it's a new (empty) document, }
               ShowWindow(window);     { make it visible}
         END;
      END;     {end of WITH statement}
      HUnlock(Handle(myData));         {unlock document record}
   END;     {end of IF (window <> NIL) AND (myData <> NIL)}
   IF NOT good THEN
   BEGIN
      IF windStorage <> NIL THEN  {memory for window record was allocated}
         DisposePtr(windStorage); {dispose of it}
      IF myData <> NIL THEN       {memory for document record was allocated}
      BEGIN
         IF myData^^.editRec <> NIL THEN  {edit record was allocated}
            TEDispose(myData^^.editRec);  {dispose of it}
         DisposeHandle(Handle(myData));   {dispose of document record}
      END;
      IF window <> NIL THEN      {window pointer exists, but it's invalid}
         CloseWindow(window);    {clean up window pointer}
      window := NIL;             {set window to NIL to indicate failure}
   END;
END; {DoNewCmd}
```

The `DoNewCmd` procedure first sets the window pointer and success flags to show that a valid window doesn't yet exist. Then it calls the application-defined function `MyPtrAllocationProc`, which allocates memory for a window record from a block set aside during program initialization for that purpose. If `MyPtrAllocationProc` successfully allocates memory and returns a valid pointer, `DoNewCmd` creates a window, specifying the `'WIND'` resource with resource ID 128, as specified by the constant `rDocWindow`. Using this window resource (defined in Listing 4-2 on page 4-26), the Window Manager creates an invisible window of type `zoomDocProc`. Because the `behind` parameter to `GetNewCWindow` or `GetNewWindow` has the value `WindowPtr(-1)`, the Window Manager places the new window in front of all others on the desktop.

The `DoNewCmd` procedure then creates a document record. It locks the document record in memory while manipulating it, sets the `refCon` field in the window record so that it points to the document record, and fills in the document record. While filling in the document record, `DoNewCmd` sets up a TextEdit record to hold the user's data. If that succeeds, `DoNewCmd` sets up horizontal and vertical scroll bars. If that succeeds, `DoNewCmd` adjusts the scroll bars (see the chapter "Control Manager" in this book for the application-defined procedure `MyAdjustScrollbars`) and fills in the remaining parts of the document record. If the window is being created to display a new document, that is, if no user data needs to be read from a disk, `DoNewCmd` calls the `ShowWindow` procedure to make the window visible immediately.

If your window resource specifies that a new window is visible, `GetNewCWindow` displays the window immediately. If you're creating a document window, however, you're more likely to create the window in an invisible state and then make it visible when you're ready to display it.

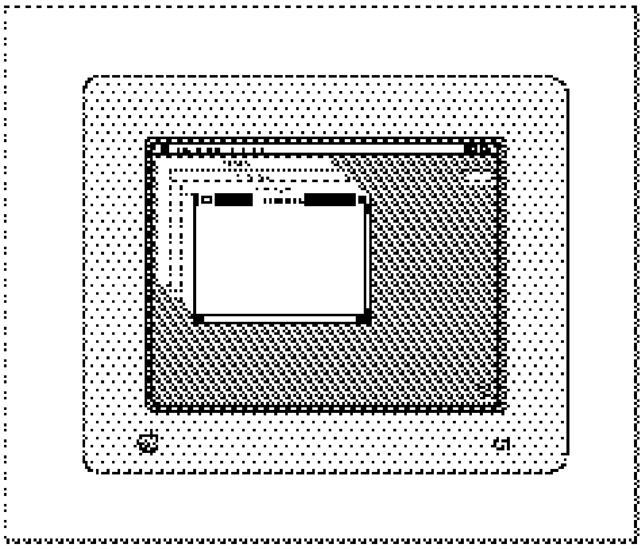
- If you're creating a window because the user is creating a new document, you can display the window immediately by calling the procedure `ShowWindow` to make the window frame visible. This change in visibility adds to the update region and triggers an update event. Your application then invokes its own procedure for drawing the content region in response to the update event.
- If you're creating a new window to display a saved document, you must retrieve the user's data before displaying it. (See *Inside Macintosh: Files* for information about reading saved files.) If possible, the size and location of the window that displays the document should be the same as when the document was last saved. (See the next section, "Positioning a Document Window on the Desktop," for a discussion of window placement.) Once you have positioned the window and set up its content region, you can make the window visible by calling `ShowWindow`, which triggers an update event. Your application then invokes its own procedure for drawing the content region.

Positioning a Document Window on the Desktop

Your goal in positioning a window on the desktop is to place it where the user expects it. For a new document, this usually means just below and to the right of the last document window in which the user was working. For a saved document, it usually means the location of the document window when the document was last saved (if it was saved on a computer with the same screen configuration). This section describes the placement of document windows. The chapter "Dialog Manager" in this book describes the placement of alert boxes and dialog boxes. See *Macintosh Human Interface Guidelines* for a complete description of window placement.

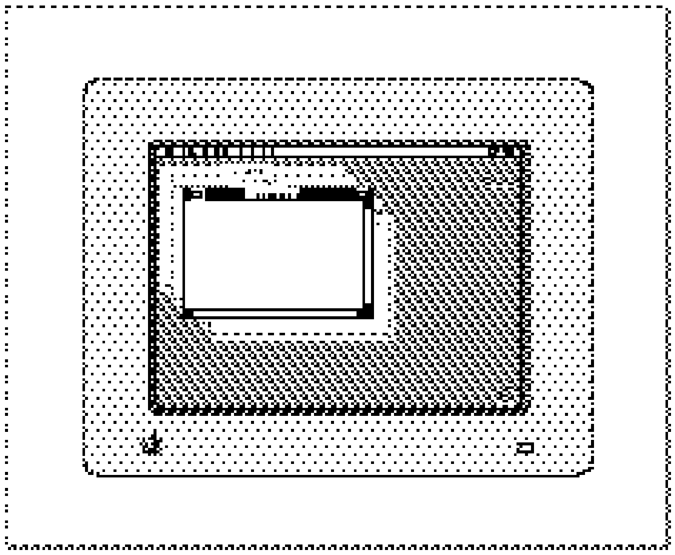
On Macintosh computers with a single screen of known size, positioning windows is fairly straightforward. You position the first new document window on the upper-left corner of the desktop. Open each additional new document window with its upper-left corner slightly below and to the right of the upper-left corner of its predecessor. Figure 4-15 illustrates how to position multiple documents on a single screen.

Figure 4-15 Document window positions on a single screen



If the user closes one or more document windows, display subsequent windows in the "empty" positions before adding more positions below and to the right. Figure 4-16 illustrates how you fill in an empty position when the user opens a new document after closing one created earlier.

Figure 4-16 "Filling in" an empty document window position



On computers with multiple monitors, window placement depends on a number of factors:

- the number of screens available and their dimensions
- the location of the main screen—that is, the screen that contains the menu bar
- the location of the screen on which the user was most recently working

In general, you place the first new document window on the main screen, and you place subsequent document windows on the screen that contains the largest portion of the most recently active document window. That is, if you display a blank document window when the user starts up your application, you place the window on the main screen. If the user moves the window to another screen and then creates another new document, you place the new document window on the other screen. Although the user is free to place windows so that they cross screen boundaries, you should never display a new window that spans multiple screens.

When the user opens a saved document, you replicate the size and location of the window in which the document was last saved, if possible.

The Window Manager recognizes a set of positioning constants in the window resource that let you position new windows automatically. You typically use the constant `staggerParentWindowScreen` for positioning document windows. The `staggerParentWindowScreen` constant specifies the basic guidelines for document window placement: When creating windows from a template that includes `staggerParentWindowScreen`, the Window Manager places the first window in the upper-left corner of the main screen. It places subsequent windows with their upper-left corners 20 pixels to the right and 20 pixels below the upper-left corner of the last window in which the user was working. Figure 4-17 illustrates how the Window Manager positions a new document window when the `staggerParentWindowScreen` specification is in effect and the user has been working in a window off the main screen.

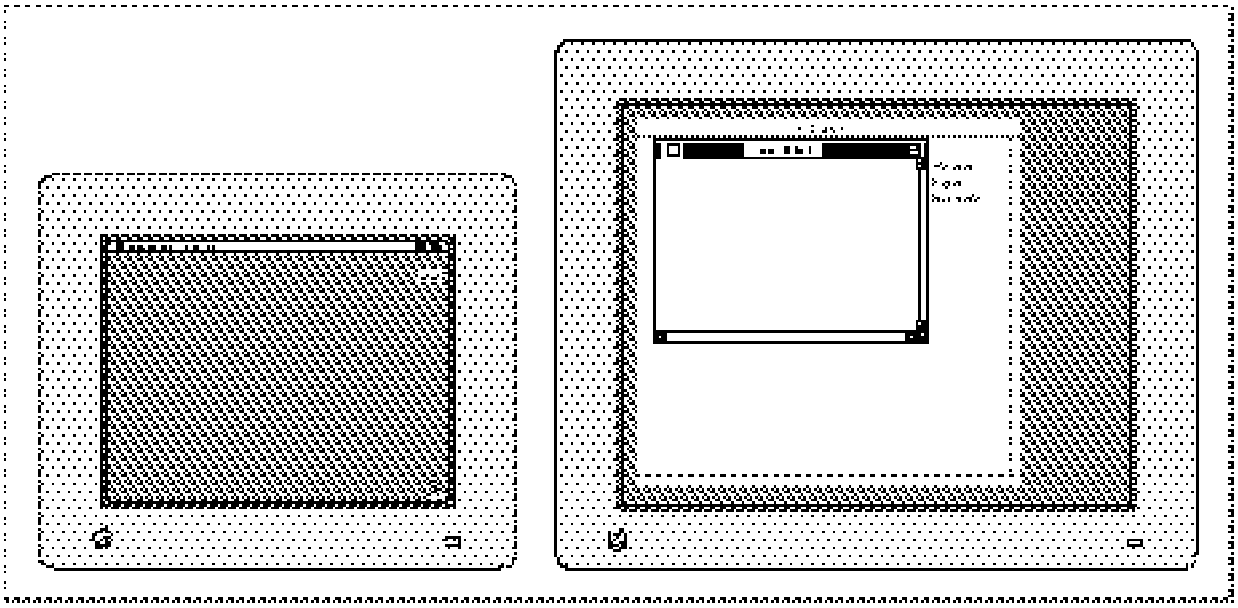
If the user moves or closes a window that occupies one of the interim positions, and the window template specifies `staggerParentWindowScreen`, the Window Manager uses the “empty” slot for the next new window created before moving further down and to the right.

For a complete list of the positioning constants and their effects, see “The Window Resource” beginning on page 4-124.

You can usually use the `staggerParentWindowScreen` positioning constant when creating a window that is to display a new document. You must perform your own window-placement calculations, however, when opening saved documents and when zooming windows.

When the user saves a document, the document window can be in one of two states: the user state or the standard state.

Figure 4-17 Document window positions on multiple screens



The **user state** is the last size and location the user established for the window.

The **standard state** is what your application determines is the most convenient size for the window, considering the function of the document and the screen space available. For a more complete description of the standard state, see "Zooming a Window" beginning on page 4-53. Your application typically calculates the standard state each time the user zooms to that state.

The user and standard states are stored in the state data record, whose handle appears in the `dataHandle` field of the window record.

```
TYPE WStateData =
   RECORD
      userState:  Rect;    {size and location established by user}
      stdState:   Rect;    {size and location established by }
                           { application}
   END;
```

When the user saves a document, you must save the user state rectangle and the state of the window (that is, whether the window is in the user state or the standard state). Then, when the user opens the document again later, you can replicate the window's status. You typically store the state data as a resource in the resource fork of the document file.

Listing 4-4 illustrates an application-defined data structure for storing the window's user rectangle and state.

Listing 4-4 Application-defined data structure for storing a window's state data

```
TYPE MyWindowState =
   RECORD
      userStateRect: Rect;    {user state rectangle}
      zoomState:     Boolean; {window state: TRUE = standard; }
                              {              FALSE = user}
   END;

MyWindowStatePtr = ^MyWindowState;
MyWindowStateHnd = ^MyWindowStatePtr;
```

This structure translates into an application-defined resource that is stored in the resource fork of the document when the user saves the document.

Listing 4-5 shows an application-defined routine for saving a document's state data. The SurfWriter application calls the procedure `MySaveWindowPosition` when the user saves a document.

Listing 4-5 Saving a document window's position

```
PROCEDURE MySaveWindowPosition (myWindow: WindowPtr;
                                 myResFileRefNum: Integer);
VAR
   lastWindowState:  MyWindowState;
   myStateHandle:    MyWindowStateHnd;
   curResRefNum:     Integer;
BEGIN
   {Set user state provisionally and determine whether window is zoomed.}
   lastWindowState.userStateRect := WindowPeek(myWindow)^.contRgn^^.rgnBBox;
   lastWindowState.zoomState := EqualRect(lastWindowState.userStateRect,
                                          MyGetWindowStdState(myWindow));
   {if window is in standard state, then set the window's user state from }
   { the userState field in the state data record}
   IF lastWindowState.zoomState THEN      {window was in standard state}
      lastWindowState.userStateRect := MyGetWindowUserState(myWindow);
   curResRefNum := CurResFile;   {save the refNum of current resource file}
   UseResFile(myResFileRefNum);  {set the current resource file}
   myStateHandle := MyWindowStateHnd(Get1Resource(rWinState,
                                                   kLastWinStateID));
   IF myStateHandle <> NIL THEN        {a state data resource already exists}
   BEGIN                               {update it}
      myStateHandle^^ := lastWindowState;
      ChangedResource(Handle(myStateHandle));
   END
   ELSE                                {no state data has yet been saved}
   BEGIN                               {add state data resource}
      myStateHandle := MyWindowStateHnd(NewHandle(SizeOf(MyWindowState)));
      IF myStateHandle <> NIL THEN
      BEGIN
         myStateHandle^^ := lastWindowState;
         AddResource(Handle(myStateHandle), rWinState, kLastWinStateID,
                     'last window state');
      END;
   END;
   IF myStateHandle <> NIL THEN
   BEGIN
      UpdateResFile(myResFileRefNum);
      ReleaseResource(Handle(myStateHandle));
   END;
   UseResFile(curResRefNum);
END;
```

The `MySaveWindowPosition` procedure first determines whether the window is in the user state or the standard state by setting its own user state field from the bounding rectangle of the window's content region and comparing that rectangle with the user state stored in the state data record. (If the two match, the window is in the user state; if not, the standard state.) If the window is in the standard state, the procedure replaces its own user state data with the rectangle stored in the `userState` field of the state data record. The rest of the procedure saves the application-defined state data record in the resource fork of the document.

When creating a new window to display a saved document, SurfWriter restores the saved user state data and recalculates the standard state. Before using the saved rectangle, however, SurfWriter verifies that the location is reachable on the desktop. (If the user saves a document on a computer equipped with multiple monitors and then opens it later on a system with only one monitor, for example, the saved window location could be entirely or partially off the screen.)

Listing 4-6 on the next page shows `MySetWindowPosition`, the application-defined routine that SurfWriter calls when the user opens a saved document. The `MySetWindowPosition` procedure retrieves the document's saved state data and then calls another application- defined routine, `MyVerifyPosition`, to verify that the saved location is practical.

Listing 4-6 Positioning the window when the user opens a saved document

```
PROCEDURE MySetWindowPosition (myWindow: WindowPtr);
VAR
   myData:              MyDocRecHnd;
   lastUserStateRect:   Rect;
   stdStateRect:        Rect;
   curStateRect:        Rect;
   myRefNum:            Integer;
   myStateHandle:       MyWindowStateHnd;
   resourceGood:        Boolean;
   savePort:            GrafPtr;
   myErr:               OSErr;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(myWindow));    {get document record}
   HLock(Handle(myData));        {lock the record while manipulating it}
   {open the resource fork and get its file reference number}
   myRefNum := FSpOpenResFile(myData^^.fileFSSpec, fsRdWrPerm);
   myErr := ResError;
   IF myErr <> noErr THEN
      Exit(MySetWindowPosition);
   {get handle to rectangle that describes document's last window position}
   myStateHandle := MyWindowStateHnd(Get1Resource(rWinState,
                                                kLastWinStateID));
   IF myStateHandle <> NIL THEN                 {handle to data succeeded}
   BEGIN    {retrieve the saved user state}
      lastUserStateRect := myStateHandle^^.userStateRect;
      resourceGood := TRUE;
   END
   ELSE
   BEGIN
      lastUserStateRect.top := 0;   {force MyVerifyPosition to calculate }
      resourceGood := FALSE;        { the default position}
   END;
   {verify that user state is practical and calculate new standard state}
   MyVerifyPosition(myWindow, lastUserStateRect, stdStateRect);
   IF resourceGood THEN                  {document had state resource}
      IF myStateHandle^^.zoomState THEN  {if window was in standard state }
         curStateRect := stdStateRect    { when saved, display it in }
                                         { newly calculated standard state}
      ELSE                {otherwise, current state is the user state}
         curStateRect := lastUserStateRect
   ELSE                                  {document had no state resource}
      curStateRect := lastUserStateRect; {use default user state}
   {move window}
   MoveWindow(myWindow, curStateRect.left, curStateRect.top, FALSE);
   {Convert to local coordinates and resize window.}
   GetPort(savePort);
   SetPort(myWindow);
   GlobalToLocal(curStateRect.topLeft);
   GlobalToLocal(curStateRect.botRight);
   SizeWindow(myWindow, curStateRect.right, curStateRect.bottom, TRUE);
   IF resourceGood THEN       {reset user state and standard }
   BEGIN                     { state--SizeWindow may have changed them}
      MySetWindowUserState(myWindow, lastUserStateRect);
      MySetWindowStdState(myWindow, stdStateRect);
   END;
   ReleaseResource(Handle(myStateHandle));         {clean up}
   CloseResFile(myRefNum);
   HUnLock(Handle(myData));
END;
```

The `MyVerifyPosition` routine, not shown here, compares the saved location against available screen space. (See Listing 4-12 on page 4-55 for a strategy for comparing the saved rectangle with the available screen space.) `MyVerifyPosition` alters the user state rectangle, if necessary (using the same size, if possible, but placing it on available screen space) and calculates a new standard state for displaying the window on the screen containing the user state.

After determining valid user and standard state rectangles, the procedure `MySetWindowPosition` sets a temporary positioning rectangle to the appropriate size and location, based on the state of the document's window when the document was saved. The `MySetWindowPosition` procedure then calls the Window Manager procedures `MoveWindow` and `SizeWindow` to establish the window's location and size before cleaning up.

The SurfWriter application calls `MySetWindowPosition` from its routine for opening saved documents, after reading the document's data from its data fork. Listing 4-7 shows the application-defined `DoOpenFile` function that SurfWriter calls when the user opens a saved document.

Listing 4-7 Opening a saved document

```
FUNCTION DoOpenFile (mySpec: FSSpec): OSErr;
VAR
   myWindow:      WindowPtr;
   myData:        MyDocRecHnd;
   myFileRefNum:  Integer;
   myErr:         OSErr;
```

```
BEGIN
   DoNewCmd(FALSE, myWindow);    {FALSE tells DoNewCmd not to }
                                 { show the window}
   IF myWindow = NIL THEN
   BEGIN
      DoOpenFile := kOpenFileError;
      Exit(DoOpenFile);
   END;
   SetWTitle(myWindow, mySpec.name);
   {open the file's data fork, passing the file spec-- }
   { FSpOpenDF returns a file reference number}
   myErr := FSpOpenDF(mySpec, fsRdWrPerm, myFileRefNum);
   IF (myErr <> noErr) AND (myErr <> opWrErr) THEN {open failed}
   BEGIN                                           {clean up}
      DisposeWindow(myWindow);
      DoOpenFile := myErr;
      Exit(DoOpenFile);
   END;
   {get a handle to the window's document record}
   myData := MyDocRecHnd(GetWRefCon(myWindow));
   myData^^.fileRefNum := myFileRefNum;   {save file ref num}
   myData^^.fileFSSpec := mySpec;         {save fsspec}
   myErr := DoReadFile(myWindow);         {read file's data}
   {retrieve saved state data and establish valid position}
   MySetWindowPosition(myWindow);
   {MyResizeWindow invalidates the whole portRgn, guaranteeing }
   { an update event--the window's contents are redrawn then}
   MyResizeWindow(myWindow);
   ShowWindow(myWindow);                  {show window}
   DoOpenFile := myErr;
END;
```

`DoOpenFile` first calls the application-defined procedure `DoNewCmd` to create a new window, suppressing the immediate display of the window. (Listing 4-3 on page page 4-28 illustrates the procedure `DoNewCmd`.) Then `DoOpenFile` sets the window title to the name of the document file and reads in the data. Then it calls `MySetWindowPosition` to determine where to place the new window. After establishing a valid position, `DoOpenFile` calls the application-defined routine `MyResizeWindow` (shown in Listing 4-14 on page 4-59) to set up the content region in the new dimensions, and then it finally makes the window visible.

Drawing the Window Contents

Your application and the Window Manager work together to display windows on the screen. Once you have created a window and made it visible, the Window Manager automatically draws the window frame in the appropriate location. As the user makes changes to the desktop, moving and resizing different windows, the Window Manager alters the window frames as necessary. The window frame includes the window outline, the title bar, and the close and zoom boxes.

Your application is responsible for drawing the window's content region. It typically uses the Control Manager to draw the window controls, uses the Window Manager to draw the size box, and draws the user data itself. The sample code in this chapter uses the simple model of a content region that contains only controls, the size box, and a TextEdit record. (See *Inside Macintosh: Text* for a description of TextEdit.)

Listing 4-8 illustrates an application-defined procedure that draws the content region of a window.

Listing 4-8 Drawing a window

```
PROCEDURE MyDrawWindow (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
   BEGIN
      EraseRect(portRect);    {erase content area}
      UpdateControls(window, visRgn);  {draw window controls}
      DrawGrowIcon(window);   {draw size box}
      {update window contents as appropriate to your }
      { application (in this case use TextEdit)}
      TEUpdate(portRect, myData^^.editRec);
   END;
   HUnLock(Handle(myData));
END;
```

The `MyDrawWindow` procedure first sets the current port to the window's port and gets a handle to the window's document record. Using the data in the document record, the procedure first erases the content region, draws the controls, and draws the size box. Finally, it draws the user's data, in this case the contents of a TextEdit edit record.

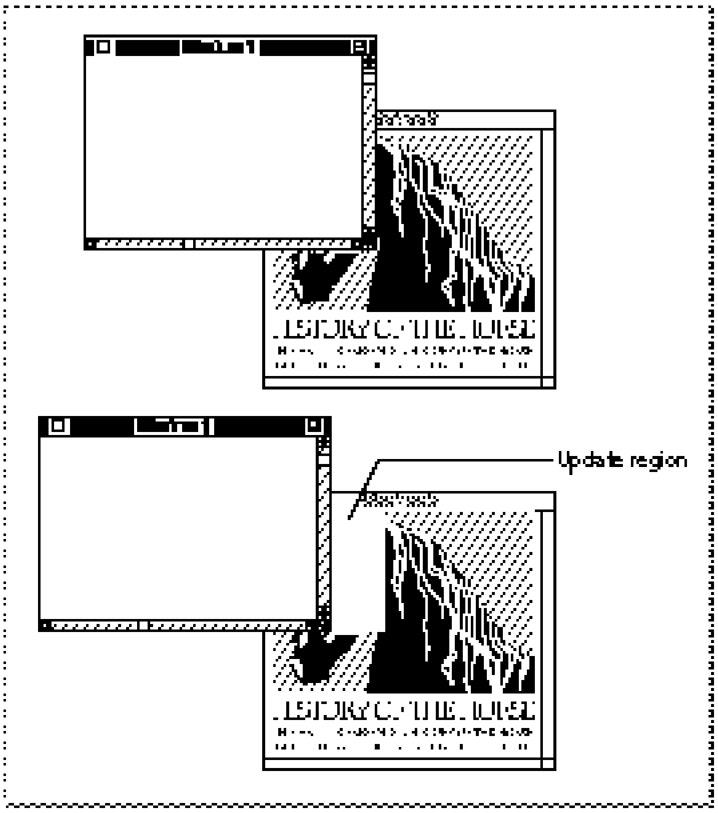
If your application creates a window that contains a static display, you can let the Window Manager take care of drawing and updating the content region by placing a handle to a picture in the `windowPic` field of the window record. See the description of the `SetWindowPic` procedure on page 4-110.

Updating the Content Region

The Window Manager helps your application keep the window display current by maintaining an update region, which represents the parts of your content region that have been affected by changes to the desktop. If a user exposes part of an inactive window by dragging an active window to a new location, for example, the Window Manager adds the newly exposed area of the inactive window to that window's update region.

Figure 4-18 illustrates how the Window Manager adds part of a window's content region to its update region when the user exposes additional content area.

Figure 4-18 Moving one window and adding to another window's update region



The Event Manager periodically scans the update regions of all windows on the desktop. If it finds one whose update region is not empty, it generates an update event for that window. When your application receives an update event, it redraws as much of the content area as necessary, as described in the section "Handling Update Events" beginning on page 4-48.

As the user makes changes to a document, your application must update both the document data and the document display in the content area of its window. You can use one of two strategies for updating the display:

- If your application doesn't require continuous scrolling or rapid response, you can add changed areas of the content region to the window's update region. The Event Manager then sends your application an update event, and your application invokes its standard update procedure.
- For continuous scrolling and a faster response time, you can draw directly into the content area of the window.

In either case, your application ultimately draws in the graphics port that represents the window. You draw controls through the Control Manager, and you draw text and graphics with the routines described in *Inside Macintosh: Text* and *Inside Macintosh: Imaging.*

Maintaining the Update Region

Your application can force and suppress update events by manipulating the update region, using Window Manager routines provided for this purpose.

Your application usually manipulates the update region, for example, when the user resizes a window that contains a size box and scroll bars. If the user enlarges the window, the Window Manager adds the newly exposed area to the window's update region but does not add the area formerly occupied by the scroll bars. Before calling the `SizeWindow` procedure to resize the window, your application can call the `InvalRect` procedure twice to add the scroll bar and size box areas to the update region. The next time it receives an update event, your application erases the scroll bars and draws whatever parts of the document content might be visible at that location.

Similarly, you can remove an area from the update region when you know that it is in fact valid. Limiting the size of the update region decreases time spent redrawing. Listing 4-13 on page 4-58, for example, uses the `ValidRect` procedure to remove the unaffected text area from the update region of a window that is being resized.

Handling Events in Windows

Your application must be prepared to handle two kinds of window-related events:

- mouse and keyboard events in your application's windows, which are reported by the Event Manager in direct response to user actions
- activate and update events, which are generated by the Window Manager and the Event Manager as an indirect result of user actions

In System 7 your application receives mouse-down events if it is the foreground process and the user clicks in the menu bar, a window belonging to your application, or a window belonging to a desk accessory that was launched in your application's partition. (If the user clicks in a window belonging to another application, the Event Manager sends your application a suspend event and performs a major switch to the other application—unless the frontmost window is an alert box or a modal dialog box, in which case the Dialog Manager merely sounds the system alert, and the Process Manager retains your application as the foreground process.) When it receives a mouse-down event, your application first calls the `FindWindow` function to map the cursor location to a window region, and then it branches to one of its own routines, as described in the next section, "Handling Mouse Events in Windows."

The Event Manager sends your application an update event when changes on the desktop or in a window require that part or all of a window's content region be updated. The Window Manager and your application can both trigger update events by adding regions that need updating to the update region, as described in the section "Handling Update Events" beginning on page 4-48.

Your application receives activate events when an inactive window becomes active or an active window becomes inactive. Activate events are an example of the close cooperation between your application and the Window Manager. When you receive a mouse-down event in one of your application's inactive windows, you can call the `SelectWindow` procedure, which removes the highlighting from the previously active window and adds highlighting to the newly active window. It also generates two activate events: one telling your application to deactivate the previously active window and one to activate the newly active window. Your application then activates and deactivates the content regions, as described in the section "Handling Activate Events" beginning on page 4-50.

When the user first clicks in an inactive window, most applications do not make a selection or otherwise change the window or document, beyond making the window active. When your application receives a resume event because the user clicked in one of its windows, you might not even want to receive the mouse-down event that caused your application to become the foreground process. You control whether or not you receive this event through the `'SIZE'` resource, described in the chapter "Event Manager" earlier in this book.

Handling Mouse Events in Windows

When your application is active, it receives notice of all keyboard activity and mouse-down events in the menu bar, in one of its windows, or in any windows belonging to desk accessories that were launched in its partition.

When it receives a mouse-down event, your application calls the `FindWindow` function to map the cursor location to a window region.

The function specifies the region by returning one of these constants:

```
CONST inDesk      = 0;  {none of the following}
      inMenuBar   = 1;  {in menu bar}
      inSysWindow = 2;  {in desk accessory window}
      inContent   = 3;  {anywhere in content region except size }
                        { box if window is active, }
                        { anywhere including size box if window }
                        { is inactive}
      inDrag      = 4;  {in drag (title bar) region}
      inGrow      = 5;  {in size box (active window only)}
      inGoAway    = 6;  {in close box}
      inZoomIn    = 7;  {in zoom box (window in standard state)}
      inZoomOut   = 8;  {in zoom box (window in user state)}
```

When the user presses the mouse button while the cursor is in a window, `FindWindow` not only returns a constant that identifies the window region but also sets a variable parameter that points to the window.

In System 7, if `FindWindow` returns `inDesk`, the cursor is somewhere other than in the menu bar, one of your windows, or a window created by a desk accessory launched in your application's partition. The function may return `inDesk` if, for example, the cursor is in the window frame but not in the drag region, close box, or zoom box. `FindWindow` seldom returns the value `inDesk`, and you can generally ignore the rare instances of this function result.

If the user presses the mouse button with the cursor in the menu bar (`inMenuBar`), you call your own routines for displaying menus and allowing the user to choose menu items.

The `FindWindow` function returns the value `inSysWindow` only when the user presses the mouse button with the cursor in a window that belongs to a desk accessory launched in your application's partition. You can then call the `SystemClick` procedure, passing it the event record and window pointer. The `SystemClick` procedure, documented in the chapter "Event Manager" in this book, makes sure that the event is handled by the appropriate desk accessory.

The `FindWindow` function returns one of the other values when the user presses the mouse button while the cursor is in one of your application's windows. Your response depends on whether the cursor is in the active window and, if not, what kind of window is active.

When you receive a mouse-down event in the active window, you route the event to the appropriate routine for changing the window display or the document contents. When the user presses the mouse button while the cursor is in the zoom box, for example, you call the Window Manager function `TrackBox` to highlight the zoom box and track the mouse until the button is released.

When you receive a mouse-down event in an inactive window, your response depends on what kind of window is active:

- If the active window is a movable modal dialog box, you should sound the system alert and take no other action. (If the active window is a modal dialog box handled by the `ModalDialog` procedure, the Dialog Manager doesn't pass the event to your application but sounds the system alert itself.)

- If the active window is a document window or a modeless dialog box, you can call `SelectWindow`, passing it the window pointer. The `SelectWindow` procedure removes highlighting from the previously active window, brings the newly activated window to the front, highlights it, and generates the activate and update events necessary to tell all affected applications which windows must be redrawn.

Listing 4-9 illustrates an application-defined procedure that handles mouse-down events.

Listing 4-9 Handling mouse-down events

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:       Integer;
   thisWindow: WindowPtr;
BEGIN
   part := FindWindow(event.where, thisWindow); {find out where cursor is}
   CASE part OF
   inMenuBar:           {cursor is in menu bar}
      BEGIN
         {make sure menu items are properly enabled/disabled}
         MyAdjustMenus;
         {let user choose a menu command}
         DoMenuCommand(MenuSelect(event.where));
      END;
   inSysWindow:            {cursor is in a desk accessory window}
      SystemClick(event, thisWindow);
   inContent:              {cursor is in the content region of one }
                           { of your application's windows}
      IF thisWindow <> FrontWindow THEN   {cursor is not in front window}
      BEGIN
         IF MyIsMovableModal(FrontWindow) THEN     {front window is }
            SysBeep(30)                            { movable modal}
         ELSE                         {front window is not movable modal}
            SelectWindow(thisWindow);  {make thisWindow active}
      END
      ELSE                 {cursor is in content region of active window}
         DoContentClick(thisWindow, event);  {handle event in content region}
   inDrag:                 {cursor is in drag area}
      {if a movable modal is active, ignore click in an inactive title bar}
      IF (thisWindow <> FrontWindow) AND MyIsMovableModal(FrontWindow) THEN
         SysBeep(30)
      ELSE
         {let Window Manager drag window}
         DragWindow(thisWindow, event.where, GetGrayRgn^^.rgnBBox);
   inGrow:                 {cursor is in size box}
      DoGrowWindow(thisWindow, event);    {change window size}
   inGoAway:               {cursor is in close box}
      {call TrackGoAway to handle mouse until button is released}
      IF TrackGoAway(thisWindow, event.where) THEN
         DoCloseCmd;                     {handle close window}
   inZoomIn, inZoomOut:    {cursor is in zoom box}
      {call TrackBox to handle mouse until button is released}
      IF TrackBox(thisWindow, event.where, part) THEN
         DoZoomWindow(thisWindow, part);  {handle zoom window}
   END;  {end of CASE statement}
END;     {end of DoMouseDownEvent}
```

The `DoMouseDown` procedure first calls `FindWindow` to map the location of the cursor to a part of the screen or a region of a window.

If the cursor is in the menu bar, `DoMouseDown` calls other application-defined procedures for adjusting and displaying menus and accepting menu choices.

If the cursor is in a window created by a desk accessory, `DoMouseDown` calls the `SystemClick` procedure, which handles mouse-down events for desk accessories from within applications.

If the cursor is in the content area of a window, `DoMouseDown` first checks to see whether the cursor is in the currently active window by comparing the window pointer returned by `FindWindow` with the result returned by the function `FrontWindow`. If the cursor is in an inactive window, `DoMouseDown` checks to see if the active window is a movable modal dialog box. (If the front window is an alert box or a fixed-position modal dialog box, an application does not receive mouse-down events in other windows.) If the active window is a movable modal dialog box and the cursor is in another window, `DoMouseDown` simply sounds the system alert and waits for another event. If the active window is not a movable modal dialog box, `DoMouseDown` calls `SelectWindow` to activate the window in which the cursor is located. The `SelectWindow` procedure relayers the windows as necessary, adjusts the highlighting, and sends the application a pair of activate events to deactivate the previously active window and activate the newly active window. `DoMouseDown` merely activates the window in which the cursor is located; it does not make a selection in the newly activated window in response to the first click in that window.

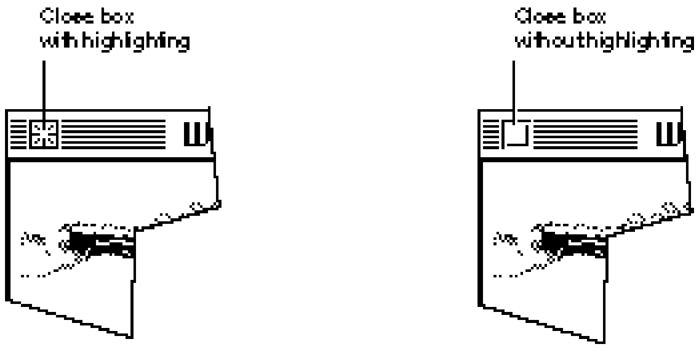
If the cursor is in the content area of the active window, the `DoMouseDown` procedure calls another application-defined procedure (`DoContentClick`) that handles mouse events in the content area.

If the cursor is in the drag region of a window, `DoMouseDown` first checks whether the drag region is in an inactive window while a movable modal dialog box is active. In that case, `DoMouseDown` merely sounds the system alert and waits for another event. In any other case, `DoMouseDown` calls the Window Manager procedure `DragWindow`, which displays an outline of the window, moves the outline as long as the user continues to drag the window, and calls `MoveWindow` to draw the window in its new location when the user releases the mouse button. After the window is drawn in its new location, it is the active window, whether or not it was active before.

If the cursor is in the size box, DoMouseDown calls another application-defined routine (DoGrowWindow, shown in Listing 4-13 on page 4-58) that resizes the window.

If the mouse press occurs in the close box, DoMouseDown calls the TrackGoAway function, which highlights the close box and tracks all mouse activity until the user releases the mouse button. As long as the user holds down the mouse button and leaves the cursor in the close box, TrackGoAway leaves the close box highlighted, as illustrated in Figure 4-19. If the user moves the cursor out of the close box, TrackGoAway removes the highlighting.

Figure 4-19 The close box with and without highlighting

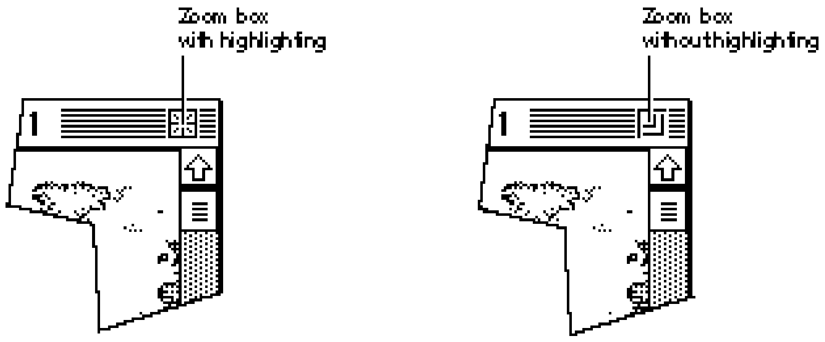


When the user releases the mouse button, TrackGoAway returns TRUE if the cursor is still in the close box and FALSE if it is not. If TrackGoAway returns TRUE, DoMouseDown calls the application-defined procedure DoCloseCmd to close the window. Listing 4-16 on page 4-60 shows the DoCloseCmd procedure.

If the mouse press occurs in the zoom box, the DoMouseDown procedure first calls TrackBox, which highlights the zoom box and tracks all mouse activity until the user releases the mouse button. As long as the user holds down the mouse button and leaves the cursor in the zoom box, TrackBox leaves the zoom box highlighted, as illustrated in Figure 4-20. If the user moves the cursor out of the zoom box, TrackBox removes the highlighting.

When the user releases the mouse button, TrackBox returns TRUE if the cursor is still in the zoom box and FALSE if it is not. If TrackBox returns TRUE, DoMouseDown calls the application-defined procedure DoZoomWindow to zoom the window. Listing 4-12 on page 4-55 shows the DoZoomWindow procedure.

Figure 4-20 The zoom box with and without highlighting



Handling Keyboard Events in Windows

Whenever your application is the foreground process, it receives key-down events for all keyboard activity, except for the three standard Command–Shift–number key sequences and any other Command–Shift–number key combinations the user has installed. (Command–Shift–1 and Command–Shift–2 eject disks, and Command–Shift–3 stores a snapshot of the screen in a TeachText document on the startup volume. Your application never receives these key combinations, which are handled by the Event Manager. For more information, see the chapter "Event Manager" in this book.)

In general, the active window is the target of keyboard activity.

When the user presses a key or a combination of keys, your application responds by inserting data into the document, changing the display, or taking other actions as defined by your application. To ensure consistent use of and response to keyboard events, follow the guidelines in *Macintosh Human Interface Guidelines.* Your application should, for example, allow the user to choose frequently used menu items by pressing a keyboard equivalent—usually a combination of the Command key and another key.

When you receive a key-down event, you first check whether the user is holding down a modifier key (Command, Shift, Control, Caps Lock, and Option, on a standard keyboard) and another key at the same time. If the Command key and a character key are held down simultaneously, for example, you adjust your menus, enabling and disabling items as appropriate, and allow the user to choose the menu item associated with the Command-key combination.

Typically, your application provides feedback for standard keystrokes by drawing the character on the screen. It should also recognize arrow keys for moving the cursor within a text display, and it might add support for function keys or other special keys available on nonstandard keyboards.

For an example of an application-defined routine for handling keyboard events, see the chapter "Event Manager" in this book.

Handling Update Events

The Event Manager sends your application an update event when part or all of your window's content region needs to be redrawn. Specifically, the Event Manager checks each window's update region every time your application calls `WaitNextEvent` or `EventAvail` (or `GetNextEvent`) and generates an update event for every window whose update region is not empty.

The Window Manager typically triggers update events when the moving and relayering of windows on the screen require that one or more windows be redrawn. If the user moves a window that covers part of an inactive window, for example, the Window Manager first calls the window definition function of the inactive window, requesting that it draw the window frame. It then adds the newly exposed area to the window's update region, which triggers an update event asking your application to update the content region. Your application can also trigger update events itself by manipulating the update region.

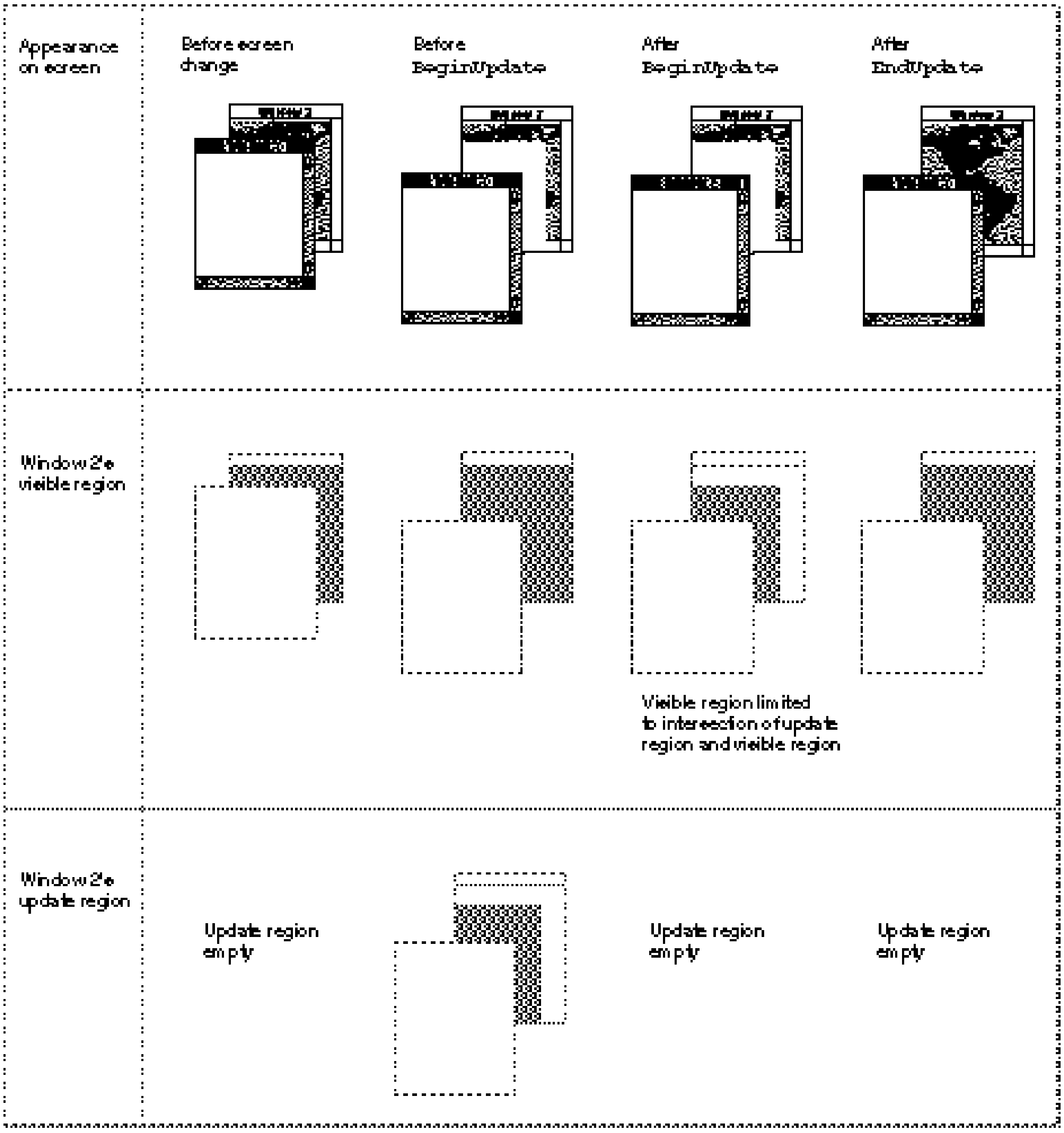
Your application can receive update events when it is in either the foreground or the background.

The Window Manager ensures that you do not accidentally draw in other windows by clipping all screen drawing to the visible region of a window's graphics port. The **visible region** is the part of the graphics port that's actually visible on the screen—that is, the part that's not covered by other windows. The Window Manager stores a handle to the visible region in the `visRgn` field of the graphics port data structure, which itself is in the window record.

In response to an update event, your application calls the `BeginUpdate` procedure, draws the window's contents, and then calls the `EndUpdate` procedure. As illustrated in Figure 4-21, `BeginUpdate` limits the visible region to the intersection of the visible region and the update region. Your application can then update either the visible region or the entire content region—because QuickDraw limits drawing to the visible region, only the parts of the window that actually need updating are drawn. The `BeginUpdate` procedure also clears the update region. After you've updated the window, you call `EndUpdate` to restore the visible region in the graphics port to the full visible region.

See *Inside Macintosh: Imaging* for more information about graphics ports and visible regions.

Figure 4-21 The effects of `BeginUpdate` and `EndUpdate` on the visible region and update region



Listing 4-10 illustrates an application-defined procedure, `DoUpdate`, that handles an update event.

Listing 4-10 Handling update events

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: LongInt;
BEGIN
   {determine type of window as defined by this application}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:           {document window}
         BEGIN
            BeginUpdate(window);
            MyDrawWindow(window);
            EndUpdate(window);
         END;
      OTHERWISE               {alert or dialog box}
         DoUpdateMyDialog(window);
      END; {of CASE}
END;
```

The `DoUpdate` procedure first determines whether the window being updated is a document window or some other application-defined window by calling the application-defined procedure `MyGetWindowType` (shown in Listing 4-1 on page 4-25). If the window is a document window, `DoUpdate` calls `BeginUpdate` to establish the temporary visible region, calls the application-defined procedure `MyDrawWindow` (shown in Listing 4-8 on page 4-39) to redraw the content region, and then calls `EndUpdate` to restore the visible region.

If the window is an alert box or a dialog box, `DoUpdate` calls the application-defined procedure `DoUpdateMyDialog`, which is not shown here.

Handling Activate Events

Your application activates and deactivates windows in response to **activate events,** which are generated by the Window Manager to inform your application that a window is becoming active or inactive. Each activate event specifies the window to be changed and the direction of the change (that is, whether it is to be activated or deactivated).

Your application often triggers activate events itself by calling the `SelectWindow` procedure. When it receives a mouse-down event in an inactive window, for example, your application calls `SelectWindow`, which brings the selected window to the front, removes the highlighting from the previously active window, and adds highlighting to the selected window. The `SelectWindow` procedure then generates two activate events: the first one tells your application to deactivate the previously active window; the second, to activate the newly active window.

When you receive the event for the previously active window, you

- hide the controls and size box
- remove or alter any highlighting of selections in the window

When you receive the event for the newly active window, you

- draw the controls and size box
- restore the content area as necessary, adding the insertion point in its former location or highlighting any previously highlighted selections

If the newly activated window also needs updating, your application also receives an update event, as described in the previous section, "Handling Update Events."

Note

A switch to one of your application's windows from a different application is handled through suspend and resume events, not activate events. See the chapter "Event Manager" in this book for a description of how your application can share processing time. u

Listing 4-11 illustrates the application-defined procedure `DoActivate`, which handles activate events.

Listing 4-11 Handling activate events

```
PROCEDURE DoActivate (window: WindowPtr; activate: Boolean;
                        event: EventRecord);
VAR
   windowType:                Integer;
   myData:                    MyDocRecHnd;
   growRect:                  Rect;
BEGIN
   {determine type of window as defined by this application}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyFindModelessDialogBox:     {modeless Find dialog box}
         DoActivateFindDBox(window, event);
                         {modeless Check Spelling dialog box}
      kMyCheckSpellingModelessDialogBox:
         DoActivateCheckSpellDBox(window, event);
      kMyDocWindow:                 {document window}
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window)); {get document record}
            HLock(Handle(myData));  {lock document record}
            WITH myData^^ DO
            IF activate THEN        {window is becoming active}
            BEGIN
               {restore selections and insert caret--if using }
               { TextEdit, for example, call TEActivate}
               TEActivate(editRec);
               MyAdjustMenus;       {adjust menus for window}
                                    {handle the controls}
               docVScroll^^.contrlVis := kControlVisible;
               docHScroll^^.contrlVis := kControlVisible;
               InvalRect(docVScroll^^.contrlRect);
               InvalRect(docHScroll^^.contrlRect);
               growRect := window^.portRect;
               WITH growRect DO     {handle the size box}
                  BEGIN             {adjust for the scroll bars}
                     top := bottom - kScrollbarAdjust;
                     left := right - kScrollbarAdjust;
                  END;
               InvalRect(growRect);
            END
            ELSE           {window is becoming inactive}
            BEGIN
               TEDeactivate(editRec);     {call TextEdit to deactivate data}
               HideControl(docVScroll);   {hide the scroll bars}
               HideControl(docHScroll);
               DrawGrowIcon(window);      {draw the size box}
            END;
         HUnLock(Handle(myData));         {unlock document record}
      END; {of kMyDocWindow statement}
   END; {of CASE statement}
END;
```

The `DoActivate` procedure first determines the general type of the window; that is, it calls an application-defined function that returns a constant identifying the type of the window: a Find dialog box, a Check Spelling dialog box, or a document window. Listing 4-1 on page 4-25 shows the `MyGetWindowType` function.

If the target of the activate event is a dialog box window, `DoActivate` calls other application-defined routines for activating and deactivating those dialog boxes. The `DoActivateFindDBox` and `DoActivateCheckSpellDBox` routines are not shown here. (The `DoActivate` procedure does not check for alert boxes and modal dialog boxes, because the Dialog Manager's `ModalDialog` procedure automatically handles activate events.)

If the target is a document window and the activate event specifies that the window is becoming active, `DoActivate` highlights any user selections in the window and draws the insertion point where appropriate. It then makes the controls visible, adds the area occupied by the scroll bars to the update region, and adds the area occupied by the size box to the update region. (Placing window area in the update region guarantees an update event. When the application receives the update event, it calls the application-defined procedure `DoUpdate` to draw the update region, which in this case includes the size box and scroll bars.)

If the target is a document window, and the activate event specifies that the window is becoming inactive, the `DoActivate` procedure calls the TextEdit procedure `TEDeactivate` to remove highlighting from user selections, calls the Control Manager procedure `HideControl` to hide the scroll bars, and calls the Window Manager procedure `DrawGrowIcon` to draw the size box and the outline of the scroll bar area.

Moving a Window

When the user drags a window by the title bar (except for the close and zoom box regions), the window should move, following the cursor as it moves on the desktop. Your application can easily let the user move the window by calling the `DragWindow` procedure.

The `DragWindow` procedure draws an outline of the window on the screen and moves the outline as the user moves the mouse. When the user releases the mouse button, `DragWindow` calls the `MoveWindow` function, which redraws the window in its new location.

For an example of moving a window, see the `inDrag` case in Listing 4-9 on page 4-44.

Zooming a Window

The zoom box allows the user to alternate quickly between two window positions and sizes: the user state and the standard state.

The **user state** is the window size and location established by the user. If your application does not supply an initial user state, the user state is simply the size and location of the window when it was created, until the user resizes it.

The **standard state** is the window size and location that your application considers most convenient, considering the function of the document and the screen space available. In a word-processing application, for example, a standard-state window might show a full page, if possible, or a page of full width and as much length as fits on the screen. If the user changes the page size through Page Setup, the application might adjust the standard state to reflect the new page size. If your application does not define a standard state, the Window Manager automatically sets the standard state to the entire gray region on the main screen, minus a three-pixel border on all sides. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.) The user cannot change a window's standard state.

The user and standard states are stored in a record whose handle appears in the dataHandle field of the window record.

```
TYPE WStateData =
   RECORD
      userState:  Rect;    {size and location established by user}
      stdState:   Rect;    {size and location established by }
                           { application}
   END;
```

The Window Manager sets the initial values of the userState and stdState fields when it fills in the window record, and it updates the userState field whenever the user resizes the window. You typically compute the standard state every time the user zooms to the standard state, to ensure that you're zooming to an appropriate location.

When the user presses the mouse button with the cursor in the zoom box, the FindWindow function specifies whether the window is in the user state or the standard state: when the window is in the standard state, FindWindow returns inZoomIn (meaning that the window is to be zoomed "in" to the user state); when the window is in the user state, FindWindow returns inZoomOut (meaning that the window is to be zoomed "out" to the standard state).

When FindWindow returns either inZoomIn or inZoomOut, your application can call the TrackBox function to handle the highlighting of the zoom box and to determine whether the cursor is inside or outside the box when the button is released. If TrackBox returns TRUE, your application can call the ZoomWindow procedure to resize the window (after computing a new standard state). If TrackBox returns FALSE, your application doesn't need to do anything. Listing 4-9 on page 4-44 illustrates the use of TrackBox in an event-handling routine.

Listing 4-12 illustrates an application-defined procedure, DoZoomWindow, which an application might call when TrackBox returns TRUE after FindWindow returns either inZoomIn or inZoomOut. Because the user might have moved the window to a different screen since it was last zoomed, the procedure first determines which screen contains the largest area of the window and then calculates the ideal window size for that screen before zooming the window.

The screen calculations in the DoZoomWindow procedure depend on the routines for handling graphics devices that were introduced at the same time as Color QuickDraw. Therefore, DoZoomWindow checks for the presence of Color QuickDraw before comparing the window to be zoomed with the graphics devices in the device list. If Color QuickDraw is not available, DoZoomWindow assumes that it's running on a computer with a single screen.

Listing 4-12 Zooming a window

```
PROCEDURE DoZoomWindow (thisWindow: windowPtr; zoomInOrOut: Integer);
VAR
   gdNthDevice, gdZoomOnThisDevice: GDHandle;
   savePort:                        GrafPtr;
   windRect, zoomRect, theSect:     Rect;
   sectArea, greatestArea:          LongInt;
   wTitleHeight:                    Integer;
   sectFlag:                        Boolean;
BEGIN
   GetPort(savePort);
   SetPort(thisWindow);
   EraseRect(thisWindow^.portRect);    {erase to avoid flicker}
   IF zoomInOrOut = inZoomOut THEN     {zooming to standard state}
   BEGIN
      IF NOT gColorQDAvailable THEN    {assume a single screen and }
      BEGIN                            { set standard state to full screen}
         zoomRect := screenBits.bounds;
         InsetRect(zoomRect, 4, 4);
         WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                             := zoomRect;
      END
      ELSE                    {locate window on available graphics devices}
      BEGIN
         windRect := thisWindow^.portRect;
         LocalToGlobal(windRect.topLeft);    {convert to global coordinates}
         LocalToGlobal(windRect.botRight);
         {calculate height of window's title bar}
         wTitleHeight := windRect.top - 1 -
                      WindowPeek(thisWindow)^.strucRgn^^.rgnBBox.top;
         windRect.top := windRect.top - wTitleHeight;
         gdNthDevice := GetDeviceList;
         greatestArea := 0;         {initialize to 0}
         {check window against all gdRects in gDevice list and remember }
         { which gdRect contains largest area of window}
         WHILE gdNthDevice <> NIL DO
         IF TestDeviceAttribute(gdNthDevice, screenDevice) THEN
            IF TestDeviceAttribute(gdNthDevice, screenActive) THEN
            BEGIN
               {The SectRect routine calculates the intersection }
               { of the window rectangle and this gDevice }
               { rectangle and returns TRUE if the rectangles intersect, }
               { FALSE if they don't.}
               sectFlag := SectRect(windRect, gdNthDevice^^.gdRect,
                                     theSect);
               {determine which screen holds greatest window area}
               {first, calculate area of rectangle on current device}
               WITH theSect DO
                  sectArea := LongInt(right - left) * (bottom - top);
               IF sectArea > greatestArea THEN
               BEGIN
                  greatestArea := sectArea;  {set greatest area so far}
                  gdZoomOnThisDevice := gdNthDevice;  {set zoom device}
               END;
               gdNthDevice := GetNextDevice(gdNthDevice);
            END;  {of WHILE}
         {if gdZoomOnThisDevice is on main device, allow for menu bar height}
         IF gdZoomOnThisDevice = GetMainDevice THEN
            wTitleHeight := wTitleHeight + GetMBarHeight;
         WITH gdZoomOnThisDevice^^.gdRect DO    {create the zoom rectangle}
         BEGIN
            {set the zoom rectangle to the full screen, minus window title }
            { height (and menu bar height if necessary), inset by 3 pixels}
            SetRect(zoomRect, left + 3, top + wTitleHeight + 3,
                   int right - 3, bottom - 3);
            {If your application has a different "most useful" standard }
            { state, then size the zoom window accordingly.}
            {set up the WStateData record for this window}
            WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                                := zoomRect;
         END;
      END;
   END; {of inZoomOut}
   {if zoomInOrOut = inZoomIn, just let ZoomWindow zoom to user state}
   {zoom the window frame}
   ZoomWindow(thisWindow, zoomInOrOut, (thisWindow = FrontWindow));
   MyResizeWindow(thisWindow);   {application-defined window-sizing routine}
   SetPort(savePort);
END; (of DoZoomWindow)
```

If the user is zooming the window to the standard state, `DoZoomWindow` calculates a new standard size and location based on the application's own considerations, the current location of the window, and the available screens. The `DoZoomWindow` procedure always places the standard state on the screen where the window is currently displayed or, if the window spans screens, on the screen containing the largest area of the window.

The bulk of the code in Listing 4-12 is devoted to determining which screen should display the window in the standard state. The sample code shown here establishes a standard state that simply occupies the gray area on the chosen screen, minus three pixels on all sides. Your application should establish a standard state appropriate to its own documents. When calculating the standard state, move the window as little as possible from the user state. If possible, anchor one corner of the standard state rectangle to one corner of the user state rectangle.

If the user is zooming the window to the user state, `DoZoomWindow` doesn't have to perform any calculations, because the user state rectangle stored in the state data record should represent a valid screen location.

After calculating the standard state, if necessary, `DoZoomWindow` calls the `ZoomWindow` procedure to redraw the window frame in the new size and location and then calls the application-defined procedure `MyResizeWindow` to redraw the window's content region. Listing 4-14 on page 4-59 shows the `MyResizeWindow` procedure.

Resizing a Window

The size box, in the lower-right corner of a window's content region, allows the user to change a window's size.

When the user positions the cursor in the size box and presses the mouse button, your application can call the Window Manager's `GrowWindow` function. This function displays a **grow image**—a gray outline of the window's frame and scroll bar areas, which expands or contracts as the user drags the size box. The grow image indicates where the window edges would be if the user released the mouse button at any given moment.

To avoid unmanageably large or small windows, you supply lower and upper size limits when you call `GrowWindow`. The `sizeRect` parameter to `GrowWindow` specifies both the lower and upper size limits in a single structure of type `Rect`. The values in the `sizeRect` structure represent window dimensions, not screen coordinates:

- You supply the minimum vertical measurement in `sizeRect.top`.
- You supply the minimum horizontal measurement in `sizeRect.left`.
- You supply the maximum vertical measurement in `sizeRect.bottom`.
- You supply the maximum horizontal measurement in `sizeRect.right`.

Most applications specify a minimum size big enough to include all parts of the structure area and the scroll bars. Because the user cannot move the cursor beyond the edges of the screen, you can safely set the maximum size to the largest possible rectangle.

When the user releases the mouse button, `GrowWindow` returns a long integer that describes the window's new height (in the high-order word) and width (in the low-order word). A value of 0 means that the window's size did not change. When `GrowWindow` returns any value other than 0, you call `SizeWindow` to resize the window.

Note
Use the utility functions `HiWord` and `LoWord` to retrieve the high-order and low-order words, respectively. u

When you change a window's size, you must erase and redraw the window's scroll bars.

Listing 4-13 illustrates the application-defined procedure `DoGrowWindow` for tracking mouse activity in the size box and resizing the window.

Listing 4-13 Resizing a window

```
PROCEDURE DoGrowWindow (thisWindow: windowPtr;
                         event: EventRecord);
VAR
   growSize:         LongInt;
   limitRect:        Rect;
   oldViewRect:      Rect;
   locUpdateRgn:     RgnHandle;
   theResult:        Boolean;
   myData:           MyDocRecHnd;
BEGIN
   {set up the limiting rectangle: kMinDocSize = 64 }
                                 { kMaxDocSize = 65535}
   SetRect(limitRect, kMinDocSize, kMinDocSize, kMaxDocSize,
            kMaxDocSize);
   {call Window Manager to let user drag size box}
   growSize := GrowWindow(thisWindow, event.where, limitRect);
   IF growSize <> 0 THEN         {if user changed size, }
   BEGIN                         { then resize window}
      myData := MyDocRecHnd(GetWRefCon(thisWindow));
      oldViewRect := myData^^.editRec^^.viewRect;
      locUpdateRgn := NewRgn;
      {save update region in local coordinates}
      MyGetLocalUpdateRgn(thisWindow, locUpdateRgn);
      {resize the window}
      SizeWindow(thisWindow, LoWord(growSize), HiWord(growSize),
                  TRUE);
      MyResizeWindow(thisWindow);
      {find intersection of old viewRect and new viewRect}
      theResult := SectRect(oldViewRect,
                            myData^^.editRec^^.viewRect,
                            oldViewRect);
      {validate the intersection (don't update)}
      ValidRect(oldViewRect);
      {invalidate any prior update region}
      InvalRgn(locUpdateRgn);
      DisposeRgn(locUpdateRgn);
   END;
END;
```

When the user presses the mouse button while the cursor is in the size box, the procedure that handles mouse-down events (DoMouseDown, shown on page 4-44) calls the application-defined DoGrowWindow procedure. The DoGrowWindow procedure calls the Window Manager function GrowWindow, which tracks mouse movement as long as the button is held down. If the user drags the size box before releasing the mouse button, GrowWindow returns a nonzero value, and DoGrowWindow prepares to resize the window. First DoGrowWindow saves the current view rectangle in the variable oldViewRect. It will use this information later, when redrawing the content region of the window in its new size. The GrowWindow procedure also saves the current update region, in local coordinates, in the region LocUpdateRgn, so that it can restore the update region after doing its own update-region maintenance. (This step is necessary only if an application allows user input to accumulate into the update region, drawing in response to update events instead of drawing into the window immediately.)

After saving the current view rectangle and the current update region, DoGrowWindow calls the Window Manager procedure SizeWindow to draw the window in its new size. The DoGrowWindow procedure then calls the application-defined procedure MyResizeWindow, which adjusts the window scroll bars and window contents to the new size. Listing 4-14 illustrates the application-defined MyResizeWindow procedure.

After calling SizeWindow, DoGrowWindow calculates the intersection of the old view rectangle and the new view rectangle. It uses this area to revalidate unchanged portions of the window (that is, to remove them from the update region), because the MyResizeWindow procedure invalidates the entire window (that is, places the entire window in the update region). This way, only the changed parts of the content area are redrawn when the application receives its next update event.

Listing 4-14 Adjusting scroll bars and content region when resizing a window

```
PROCEDURE MyResizeWindow (window: WindowPtr);
BEGIN
   WITH window^ DO
   BEGIN
      {adjust scroll bars and contents-- }
      { see the chapter “Control Manager” for implementation}
      MyAdjustScrollbars(window, TRUE);
      MyAdjustTE(window);
      {invalidate content region, forcing an update}
      InvalRect(portRect);
   END;
END; {MyResizeWindow}
```

Listing 4-15 illustrates the application-defined procedure `MyGetLocalUpdateRgn`, which supplies a window's update region in local coordinates. The `MyGetLocalUpdateRgn` procedure uses the QuickDraw routines `CopyRgn` and `OffsetRgn`, documented in *Inside Macintosh: Imaging*.

Listing 4-15 Converting a window region to local coordinates

```
PROCEDURE MyGetLocalUpdateRgn (window: WindowPtr;
                               localRgn: RgnHandle);
BEGIN
   {save old update region}
   CopyRgn(WindowPeek(window)^.updateRgn, localRgn);
   WITH window^.portBits.bounds DO
      OffsetRgn(localRgn, left, top);  {convert to local coords}
END; {MyGetLocalUpdateRgn}
```

Closing a Window

The user closes a window either by clicking the close box, in the upper-left corner of the window, or by choosing Close from the File menu.

When the user presses the mouse button while the cursor is in the close box, your application calls the `TrackGoAway` function to track the mouse until the user releases the button, as illustrated in Listing 4-9 on page 4-44. If the user releases the button while the cursor is outside the close box, `TrackGoAway` returns `FALSE`, and your application does nothing. If `TrackGoAway` returns `TRUE`, your application invokes its own procedure for closing a window.

The specific steps you take when closing a window depend on what kind of information the window contains and whether the contents need to be saved. The sample code in this chapter recognizes four kinds of windows: the modeless dialog box containing the Find dialog, the modeless dialog box containing the Spell Check dialog, a standard document window, and a window associated with a desk accessory that was launched in the application's partition.

Listing 4-16 illustrates an application-defined procedure, `DoCloseCmd`, that determines what kind of window is being closed and follows the appropriate strategy. The application calls `DoCloseCmd` when the user clicks a window's close box or chooses Close from the File menu.

Listing 4-16 Handling a close command

```
PROCEDURE DoCloseCmd;
VAR
   myWindow:   WindowPtr;     {pointer to window's record}
   myData:     MyDocRecHnd;   {handle to a document record}
   windowType: Integer;       {application-defined window type}
```

```
BEGIN
   myWindow := FrontWindow;
   windowType := MyGetWindowType(myWindow);
   CASE windowType OF
      kMyFindModelessDialog:        {for modeless dialog boxes, }
         HideWindow(myWindow);      { hide window}
      kMySpellModelessDialog:       {for modeless dialog boxes, }
         HideWindow(myWindow);      { hide window}
      kDAWindow:              {for desk accessories, close the DA}
         CloseDeskAcc(WindowPeek(myWindow)^.windowKind);
      kMyDocWindow:           {for documents, handle file first}
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(myWindow));
            MyCloseDocument(myData);
         END;
   END;     {of CASE}
END;
```

The `DoCloseCmd` procedure first determines which window is the active window and then calls the application-defined function `MyGetWindowType` to identify the window's type, as defined by the application. If the window is a modeless dialog box, `MyCloseCmd` merely hides the window, leaving the data structures in memory. For a sample routine that displays a hidden window, see Listing 4-18 on page 4-64.

If the window is associated with a desk accessory, the `DoCloseCmd` procedure calls the `CloseDeskAcc` procedure to close the desk accessory. This case is included only for compatibility; in System 7 desk accessories are seldom launched in an application's partition.

If the window is associated with a document, `DoCloseCmd` reads the document record and then calls the application-defined procedure `MyCloseDocument` to handle the closing of a document window. Listing 4-17 illustrates the `MyCloseDocument` procedure.

Listing 4-17 Closing a document

```
PROCEDURE MyCloseDocument (myData: MyDocRecHnd);
VAR
   title:      Str255;        {window/document title}
   item:       Integer;       {item in Save Alert dialog box}
   docWindow:  WindowPtr;     {pointer to window record}
   event:      EventRecord;   {dummy record for DoActivate}
   myErr:      OSErr;         {variable for error-checking}
BEGIN
   docWindow := FrontWindow;
   IF (myData^^.windowDirty) THEN   {changed since last save}
   BEGIN
      GetWTitle(docWindow, title);     {get window title}
      ParamText(title, '', '', '');    {set up dialog text}
      {deactivate window before displaying Save dialog}
      DoActivate(docWindow, FALSE, event);
      {put up Save dialog and retrieve user response}
      item := CautionAlert(kSaveAlertID, @MyEventFilter);
      IF item = kCancel THEN     {user clicked Cancel}
         Exit(MyCloseDocument);  {exit without closing}
      IF item = kSave THEN       {user clicked Save}
         DoSaveCmd;              {save the document}
      {otherwise user clicked Don't Save-- }
      { close document in either case}
      myErr := DoCloseFile(myData); {close document}
      {Add your own error handling.}
   END;
   {close window whether or not user saved}
   CloseWindow(docWindow);          {close window}
   DisposePtr(Ptr(docWindow));      {dispose of window record}
END;
```

The `MyCloseDocument` procedure checks the `windowDirty` field in the document record (described in "Managing Multiple Windows" beginning on page 4-23). If the value of `windowDirty` is `TRUE`, `MyCloseDocument` displays a dialog box giving the user a chance to save the document before closing the window. The dialog box gives the user the choices of canceling the close, saving the document before closing the window, or closing the window without saving the document. If the user cancels, `MyCloseDocument` merely exits. If the user opts to save the document, `MyCloseDocument` calls the application-defined routine `DoSaveCmd`, which is not shown here. (For a description of how to save and close a file, see the chapter "Introduction to File Management" in *Inside Macintosh: Files*.) Whether or not the user saves the document before closing the window, `MyCloseDocument` closes the document and finally removes the window from the screen and diposes of the memory allocated to the window record.

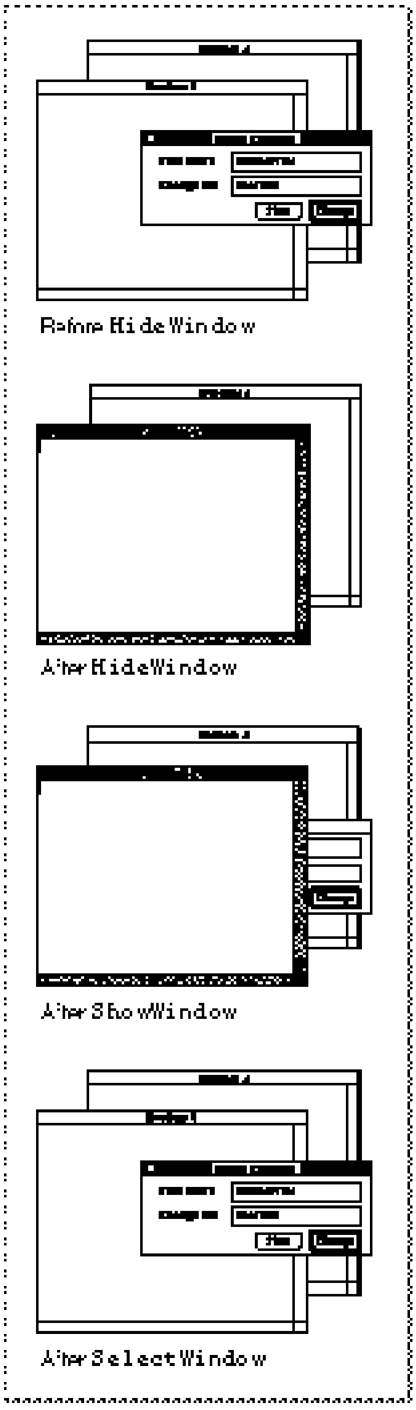
Hiding and Showing a Window

Whenever the user clicks a window's close box, you remove the window from the screen. Sometimes, however, you might find it's more efficient to merely hide the window, instead of removing its data structures.

If your application includes a Find modeless dialog box that searches for a string, for example, you might want to keep the structures in memory as long as the user is working. When the user closes the dialog box by clicking the close box, you simply hide the window by calling the `HideWindow` procedure. The next time the user chooses the Find command, your dialog box window is already available, in the same location and with the same text selected as when it was last used.

To reverse the `HideWindow` procedure, you must call both `ShowWindow`, which makes the window visible, and `SelectWindow`, which makes it the active window. Figure 4-22 illustrates how the three procedures affect the window's status on the screen.

Figure 4-22 The cumulative effects of `HideWindow`, `ShowWindow`, and `SelectWindow`



The application-defined procedure for closing a window—DoCloseCmd, described on page 4-60—hides the Find and Spell Check dialog box windows when the user closes them. Listing 4-18 illustrates a sample application-defined procedure, DoShowModelessFindDialogBox, for redisplaying the Find dialog box when the user next chooses the Find command.

Listing 4-18 Showing a hidden dialog box

```
PROCEDURE DoShowModelessFindDialogBox;
BEGIN
   IF gFindDialog = NIL THEN     {no Find dialog box exists yet}
   BEGIN
      {create Find dialog box}
      gFindDialog := GetNewDialog(rFindModelessDialog, NIL,
                                  Pointer(-1));
      IF gFindDialog = NIL THEN        {creation failed}
         Exit(DoShowModelessFindDialogBox);     {exit}
      {store value that identifies dbox in window refCon field}
      SetWRefCon(gFindDialog, LongInt(kMyFindModelessDialog))
      ShowWindow(gFindDialog);         {make dialog box visible}
   END
   ELSE              {dialog box already exists}
   BEGIN
      ShowWindow(gFindDialog);         {make it visible}
      SelectWindow(gFindDialog);       {select it}
   END;
END;
```

The DoShowModelessFindDialogBox procedure first checks whether the Find dialog box already exists. If it doesn't, then DoShowModelessFindDialogBox creates a new dialog box through the Dialog Manager. It stores the constant that represents the Find dialog box in the refCon field of the new window record, makes the window visible, and draws the dialog box contents. If the Find dialog box already exists, DoShowModelessFindDialogBox makes the dialog box window visible and selects it. When the Window Manager then generates an activate event, the application calls its own procedure to draw the contents.

Window Manager Reference

This section describes the Window Manager's data structures and routines. It also lists the resources used by the Window Manager and describes the window ('WIND') and window color table ('wctb') resources.

Data Structures

This section describes the Window Manager data structures: the window record, the color window record, the state data record, the window color table record, the auxiliary window record, and the window list.

A window record or color window record describes an individual window. It includes the record for the graphics port in which the window is displayed.

The state data record stores two rectangles, known as the user state and the standard state, which define the size and location of the window as specified by the user and by your application. Your application switches between the two states when the user clicks the zoom box.

A window color table defines the colors to be used for drawing the window's frame and highlighting selected text. Ordinarily, you use the default window color table, which produces windows in the colors selected by the user through the Color control panel. If your application has some unusual need to control the frame colors, you can set up your own window color tables.

The Window Manager uses auxiliary window records to associate a window with its window color table.

The Window Manager uses the window list to track all of the windows on the desktop.

The Color Window Record

The Window Manager maintains a window record or color window record for each window on the desktop.

The Window Manager supplies routines that let you access the window record as necessary. Your application seldom changes fields in the window record directly.

The `CWindowRecord` data type defines the window record for a color window. The `CWindowPeek` data type is a pointer to a color window record. The first field in the window record is in fact the record that describes the window's graphics port. The `CWindowPtr` data type is defined as a pointer to the window's graphics port.

When Color QuickDraw is not available, you can create monochrome windows using the parallel data types `WindowRecord`, `WindowPeek`, and `WindowPtr`, described in the next section, "The Window Record."

For compatibility, the `WindowPtr` and `WindowPeek` data types can point to either a color window record or a monochrome window record. You use the `WindowPtr` data type to specify a window in most Window Manager routines, and you can use it to specify a graphics port in QuickDraw routines that take the `GrafPtr` data type. Note that you can access only the fields of the window's graphics port, not the rest of the window record, through the `WindowPtr` and `CWindowPtr` data types. You use the `WindowPeek` and `CWindowPeek` data types in low-level Window Manager routines and in your own routines that access window record fields beyond the graphics port.

The routines that manipulate color windows get color information from the window color tables and the auxiliary window record described in the sections “The Window Color Table Record” on page 4-71 and “The Auxiliary Window Record” on page 4-73.

```
TYPE  CWindowPtr  = ^CGrafPtr;
      CWindowPeek = ^CWindowRecord;

TYPE  CWindowRecord =
   RECORD
      port:          CGrafPort;     {window's graphics port}
      windowKind:    Integer;       {class of the window}
      visible:       Boolean;       {visibility}
      hilited:       Boolean;       {highlighting}
      goAwayFlag:    Boolean;       {presence of close box}
      spareFlag:     Boolean;       {presence of zoom box}
      strucRgn:      RgnHandle;     {handle to structure region}
      contRgn:       RgnHandle;     {handle to content region}
      updateRgn:     RgnHandle;     {handle to update region}
      windowDefProc: Handle;        {handle to window definition }
                                    { function}
      dataHandle:    Handle;        {handle to window state }
                                    { data record}
      titleHandle:   StringHandle;  {handle to window title}
      titleWidth:    Integer;       {title width in pixels}
      controlList:   ControlHandle; {handle to control list}
      nextWindow:    CWindowPeek;   {pointer to next window }
                                    { record in window list}
      windowPic:     PicHandle;     {handle to optional picture}
      refCon:        LongInt;       {storage available to your }
                                    { application}
   END;
```

Field descriptions

`port`	The graphics port record that describes the graphics port in which the window is drawn. The graphics port record, which is documented in *Inside Macintosh: Imaging*, defines the rectangle in which drawing can occur, the window’s visible region, the window’s clipping region, and a collection of current drawing characteristics such as fill pattern, pen location, and pen size.
`windowKind`	The class of window—that is, how the window was created. The Window Manager fills in this field when it creates the window record. It places a negative value in `windowKind` when the window

was created by a desk accessory. (The value is the reference ID of the desk accessory.) This field can also contain one of two constants:

```
CONST
   dialogKind  = 2;     {dialog or alert window}
   userKind    = 8;     {window created by an }
                        { application}
```

The value `dialogKind` identifies all dialog or alert box windows, whether created by the system software or, indirectly through the Dialog Manager, by your application. The Dialog Manager uses this field to help it track dialog and alert box windows.

The value `userKind` represents a window created directly by your application.

`visible` A Boolean value indicating whether or not the window is visible. If the window is visible, the Window Manager sets this field to `TRUE`; if not, `FALSE`. Visibility means only whether or not the window is to be displayed, not necessarily whether you can see it on the screen. (For example, a window that is completely covered by other windows can still be visible, even if the user cannot see it on the screen.)

`hilited` A Boolean value indicating whether the window is highlighted—that is, drawn with stripes in the title bar. Only the active window is ordinarily highlighted. When the window is highlighted, the `hilited` field contains `TRUE`; when not, `FALSE`.

`goAwayFlag` A Boolean value indicating whether the window has a close box.

The Window Manager fills in this field when it creates the window according to the information in the `'WIND'` resource or the parameters passed to the function that creates the window.

If the value of `goAwayFlag` is `TRUE`, and if the window type supports a close box, the Window Manager draws a close box when the window is highlighted.

`spareFlag` A Boolean value indicating whether the window type supports zooming. The Window Manager sets this field to `TRUE` if the window's type is one that includes a zoom box (`zoomDocProc`, `zoomNoGrow`, or even `modalDBoxProc + zoomDocProc`).

`strucRgn` A handle to the structure region, which is defined in global coordinates. The structure region is the entire screen area covered by the window—that is, both the window contents and the window frame.

`contRgn` A handle to the content region, which is defined in global coordinates. The content region is the part of the window that contains the document, dialog, or other data; the window controls; and the size box.

`updateRgn` A handle to the update region, which is defined in global coordinates. The update region is the portion of the window that must be redrawn. It is maintained jointly by the Window Manager and your application. The update region excludes parts of the window that are covered by other windows.

`windowDefProc`	A handle to the definition function that controls the window. There's no need for your application to access this field directly. In Macintosh models that use only 24-bit addressing, this field contains both a handle to the window's definition function and the window's variation code. If you need to know the variation code, regardless of the addressing mode, call the `GetWVariant` function.
`dataHandle`	Usually a handle to a data area used by the window definition function. For zoomable windows, `dataHandle` contains a handle to the `WStateData` record, which contains the user state and standard state rectangles. The `WStateData` record is described in "The Window State Data Record" beginning on page 4-70. A window definition function that needs only 4 bytes of data can use the `dataHandle` field directly, instead of storing a handle to the data. The window definition function that handles rounded-corner windows, for example, stores the diameters of curvature in the `dataHandle` field.
`titleHandle`	A handle to the string that defines the title of the window.
`titleWidth`	The width, in pixels, of the window's title.
`controlList`	A handle to the window's control list, which is used by the Control Manager. (See the chapter "Control Manager" in this book for a description of control lists.)
`nextWindow`	A pointer to the next window in the window list, that is, the window behind this window on the desktop. In the window record for the last window on the desktop, the `nextWindow` field is set to `NIL`.
`windowPic`	A handle to a QuickDraw picture of the window's contents. The Window Manager initially sets the `windowPic` field to `NIL`. If you're using the window to display a stable image, you can use the `SetWindowPic` procedure to place a handle to the picture in this field. When the window's contents need updating, the Window Manager then redraws the contents itself instead of generating an update event.
`refCon`	The window's reference value field, which is simply storage space available to your application for any purpose. The sample code in this chapter uses the `refCon` field to associate a window with the data it displays by storing a window type constant in the `refCon` field of alert and dialog window records and a handle to a document record in the `refCon` field of a document window record.

Note

The close box, drag region, zoom box, and size box are not included in the window record because they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are. u

The Window Record

If Color QuickDraw is not available, you create windows with a parallel data structure, the window record. The only difference between a color window record and a window record is that a color window record points to a color graphics port, which allows full use of Macintosh computers with color capability, and a window record points to a monochrome graphics port

The data types that describe window records, `WindowRecord`, `WindowPtr`, and `WindowPeek`, are parallel to the data types that describe color window records, and the fields in the monochrome window record are identical to the fields in the color window record. For a complete description, see "The Color Window Record" beginning on page 4-65.

```
TYPE  WindowPtr   = ^GrafPtr;
      WindowPeek  = ^WindowRecord;

TYPE  WindowRecord =                {all fields have same use }
   RECORD                           { as in color window record}
      port:          GrafPort;      {window's graphics port}
      windowKind:    Integer;       {class of the window}
      visible:       Boolean;       {visibility}
      hilited:       Boolean;       {highlighting}
      goAwayFlag:    Boolean;       {presence of close box}
      spareFlag:     Boolean;       {presence of zoom box}
      strucRgn:      RgnHandle;     {handle to structure region}
      contRgn:       RgnHandle;     {handle to content region}
      updateRgn:     RgnHandle;     {handle to update region}
      windowDefProc: Handle;        {handle to window definition }
                                    { function}
      dataHandle:    Handle;        {handle to window state }
                                    { data record}
      titleHandle:   StringHandle;  {handle to window title}
      titleWidth:    Integer;       {title width in pixels}
      controlList:   ControlHandle; {handle to control list}
      nextWindow:    WindowPeek;    {pointer to next window }
                                    { record in window list}
      windowPic:     PicHandle;     {handle to optional picture}
      refCon:        LongInt;       {storage available to your }
                                    { application}
   END;
```

The Window State Data Record

The zoom box allows the user to alternate quickly between two window positions and sizes: the user state and the standard state. The Window Manager stores the user state and your application stores the standard state in the window state data record, whose handle appears in the `dataHandle` field of the window record.

The `WStateData` record data type defines the window state data record.

```
TYPE  WStateDataPtr = ^WStateData;
      WStateDataHandle = ^WStateDataPtr;

      WStateData =
      RECORD
         userState:  Rect; {size and location established by user}
         stdState:   Rect; {size and location established by app}
      END;
```

Field descriptions

`userState`	A rectangle that describes the window size and location established by the user. The Window Manager initializes the user state to the size and location of the window when it is first displayed, and then updates the `userState` field whenever the user resizes a window. Although the user state specifies both the size and location of the window, the Window Manager updates the state data record only when the user resizes a window—not when the user merely moves a window.
`stdState`	The rectangle describing the window size and location that your application considers the most convenient, considering the function of the document, the screen space available, and the position of the window in its user state. If your application does not define a standard state, the Window Manager automatically sets the standard state to the entire gray region on the main screen, minus a three-pixel border on all sides. The user cannot change a window's standard state. Your application typically calculates and sets the standard state each time the user zooms to the standard state. In a word-processing application, for example, a standard state window might show a full page, if possible, or a page of full width and as much length as fits on the screen. If the user changes the page size through Page Setup, the application might adjust the standard state to reflect the new page size. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.)

The `ZoomWindow` procedure changes the size of a window according to the values in the window state data record. The procedure changes the window to the user state when the user zooms "in" and to the standard state when the user zooms "out." For a detailed description of zooming windows, see “Zooming a Window” beginning on page 4-53. For descriptions of the routines you call when zooming windows, see “Zooming Windows” beginning on page 4-101.

The Window Color Table Record

The user controls the colors used for the window frame and text highlighting through the Color control panel. Ordinarily, your application doesn’t override the user’s color choices, which are stored in a default window color table. If you have some extraordinary need to control window colors, you can do so by defining window color tables for your application’s windows.

The Window Manager maintains window color information tables in a data structure of type `WinCTab`.

You can define your own window color table and apply it to an existing window through the `SetWinColor` procedure.

To establish the window color table for a window when you create it, you provide a window color table (`'wctb'`) resource with the same resource ID as the `'WIND'` resource that defines the window.

The `WCTabPtr` data type is a pointer to a window color table record, and the `WTabHandle` is a handle to a window color table record.

```
TYPE  WCTabPtr = ^WinCTab;
      WCTabHandle = ^WCTabPtr;
```

The `WinCTab` data type defines a window color table record.

```
TYPE  WinCTab =
      RECORD
         wCSeed:     LongInt;    {reserved}
         wCReserved: Integer;    {reserved}
         ctSize:     Integer;    {number of entries in table -1}
         ctTable:    ARRAY[0..4] OF ColorSpec;
                                 {array of color specification }
                                 { records}
      END;
```

Field descriptions

`wCSeed`	Reserved.
`wCReserved`	Reserved.
`ctSize`	The number of entries in the table, minus 1. If you’re building a color table for use with the standard window definition function, the maximum value of this field is 12. Custom window definition functions can use color tables of any size.

`ctTable` An array of `colorSpec` records.

In a window color table, each `colorSpec` record specifies a window part in the first word and an RGB value in the other three words:

```
TYPE  ColorSpec =
      RECORD
         value:   Integer;    {part identifier}
         rgb:     RGBColor;   {RGB value}
      END;
```

The `value` field of a `colorSpec` record specifies a constant that defines which part of the window the color controls. For the window color table used by the standard window definition function, you can specify these values with these meanings:

```
CONST
wContentColor     = 0;  {content region background}
wFrameColor       = 1;  {window outline}
wTextColor        = 2;  {window title and button }
                        { text}
wHiliteColor      = 3;  {reserved}
wTitleBarColor    = 4;  {reserved}
wHiliteColorLight = 5;  {lightest stripes in }
                        { title bar and lightest }
                        { dimmed text}
wHiliteColorDark  = 6;  {darkest stripes in }
                        { title bar and }
                        { darkest dimmed }
                        { text}
wTitleBarLight    = 7;  {lightest parts of }
                        { title bar background}
wTitleBarDark     = 8;  {darkest parts of }
                        { title bar background}
wDialogLight      = 9;  {lightest element }
                        { of dialog box frame}
wDialogDark       = 10; {darkest element of }
                        { dialog box frame}
wTingeLight       = 11; {lightest window tinging}
wTingeDark        = 12; {darkest window tinging}
```

Note

The part codes in System 5 and System 6 are significantly different from the part codes described here, which apply only to System 7. u

The window parts can appear in any order in the table.

The `rgb` field of a `ColorSpec` record contains three words of data that specify the red, green, and blue values of the color to be used. The `RGBColor` data type is defined in *Inside Macintosh: Imaging*.

When your application creates a window, the Window Manager first looks for a resource of type `'wctb'` with the same resource ID as the `'WIND'` resource used for the window. If it finds one, it creates a window color table for the window from the information in that resource, and then displays the window in those colors. If it doesn't find a window color table resource with the same resource ID as your window resource, the Window Manager uses the default system window color table, read into the heap during application startup.

After creating a window, you can change the entries in a window's window color table with the `SetWinColor` procedure, described on page 4-114.

See "The Window Color Table Resource" on page 4-127 for a description of the window color table resource.

The Auxiliary Window Record

The auxiliary window record specifies the color table used by a window and contains reference information used by the Dialog Manager and the Window Manager.

The Window Manager creates and maintains the information in an auxiliary window record; your application seldom, if ever, needs to access an auxiliary window record.

```
TYPE  AuxWinPtr      = ^AuxWinRec;
      AuxWinHandle   = ^AuxWinPtr;

      AuxWinRec =
      RECORD
         awNext:        AuxWinHandle;  {handle to next record}
         awOwner:       WindowPtr;     {pointer to window }
                                       { associated with this }
                                       { record}
         awCTable:      CTabHandle;    {handle to color table}
         dialogCItem:   Handle;        {storage used by }
                                       { Dialog Manager}
         awFlags:       LongInt;       {reserved}
         awReserved:    CTabHandle;    {reserved}
         awRefCon:      LongInt;       {reference constant, }
                                       { for application's use}
      END;
```

Field descriptions

`awNext`	A handle to the next record in the auxiliary window list, used by the Window Manager to maintain the auxiliary window list as a linked list. If a window is using the default auxiliary window record, this value is `NIL`.
`awOwner`	A pointer to the window that uses this record. The `awOwner` field of the default auxiliary window record is set to `NIL`.

`awCTable`	A handle to the window's color table. Unless you specify otherwise, this is a handle to the system window color table.
`dialogCItem`	Private storage for use by the Dialog Manager.
`awFlags`	Reserved.
`awReserved`	Reserved.
`awRefCon`	The reference constant, typically used by an application to associate the auxiliary window record with a document record.

Except in unusual circumstances, your application doesn't need to manipulate window color tables or the auxiliary window record.

For compatibility with other applications in the shared environment, your application should not manipulate system color tables directly but should go through the Palette Manager, documented in *Inside Macintosh: Imaging.* If your application provides its own window and control definition functions, these functions should apply the user's desktop color choices the same way the standard window and control definition functions do.

The Window List

The Window Manager maintains information about the windows on the desktop in a private structure called the *window list.* The window list contains pointers to all windows on the desktop, both visible and invisible, and contains other information that the Window Manager uses to maintain the desktop.

Your application should not directly access the information in a window list. The structure of the window list is private to the Window Manager.

The global variable `WindowList` contains a pointer to the first window in the window list.

Window Manager Routines

This section describes the complete set of routines for creating, displaying, and managing windows.

Initializing the Window Manager

Before using any other other Window Manager routines, you must initialize the Window Manager by calling the `InitWindows` procedure.

As part of initialization, `InitWindows` creates the **Window Manager port,** a graphics port that occupies all of the main screen. The Window Manager port is named `WMgrCPort` on Macintosh computers equipped with Color QuickDraw and `WMgrPort` on computers with only QuickDraw.

Ordinarily, your application does not need to know about the Window Manager port. If necessary, however, you can retrieve a pointer to it by calling the procedure `GetWMgrPort` or `GetCWMgrPort`. Your application should not draw directly into the Window Manager port, except through custom window definition functions.

The Window Manager draws your application's windows into the Window Manager port. The port rectangle of the Window Manager port is the bounding rectangle of the main screen (`screenBits.bounds`). To accommodate systems with multiple monitors, QuickDraw recognizes a port rectangle of `screenBits.bounds` as a special case and allows drawing on all parts of the desktop.

InitWindows

The procedure `InitWindows` initializes the Window Manager for your application. Before calling `InitWindows`, you must initialize QuickDraw and the Font Manager by calling the `InitGraf` and `InitFonts` procedures, documented in *Inside Macintosh: Imaging* and *Inside Macintosh: Text.*

```
PROCEDURE InitWindows;
```

DESCRIPTION

The `InitWindows` procedure initializes the Window Manager.

ASSEMBLY-LANGUAGE INFORMATION

When the desktop needs to be redrawn any time after initialization, the Window Manager checks the global variable `DeskHook`, which can be used as a pointer to an application-defined routine for drawing the desktop. This variable is ordinarily set to 0, but not until after system startup. If you're displaying windows in code that is to be executed during startup, set `DeskHook` to 0. Note that the use of the Window Manager's global variables is not guaranteed to be compatible in system software versions later than System 6.

Creating Windows

You can create windows in two ways:

- from a window resource (a resource of type `'WIND'`), with the `GetNewCWindow` and `GetNewWindow` functions
- from a collection of window characteristics passed as parameters to the `NewCWindow` and `NewWindow` functions

Creating windows from resources allows you to localize your application for different languages and to change the characteristics of your windows during application development by changing only the window resources.

All four functions, `GetNewCWindow`, `GetNewWindow`, `NewCWindow`, and `NewWindow`, can allocate space in your application's heap for the new window's window record. For more control over memory use, you can allocate the space yourself and pass a pointer when creating a window. In either case, the Window Manager fills in the data structure and returns a pointer to it.

GetNewCWindow

Use the `GetNewCWindow` function to create a color window with the properties defined in the `'WIND'` resource with a specified resource ID.

```
FUNCTION GetNewCWindow (windowID: Integer; wStorage: Ptr;
                        behind: WindowPtr): WindowPtr;
```

`windowID` The resource ID of the `'WIND'` resource that defines the properties of the window.

`wStorage` A pointer to memory space for the window record.

If you specify a value of `NIL` for `wStorage`, the `GetNewCWindow` function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the `wStorage` parameter.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(-1)`. When you place a window in front of all others, `GetNewCWindow` removes the highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

DESCRIPTION

The `GetNewCWindow` function creates a new color window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `GetNewCWindow` is unable to read the window or window definition function from the resource file, it returns `NIL`.

The `GetNewCWindow` function looks for a `'wctb'` resource with the same resource ID as that of the `'WIND'` resource. If it finds one, it uses the window color information in the `'wctb'` resource for coloring the window frame and highlighting selected text.

If the window's definition function (specified in the window resource) is not already in memory, `GetNewCWindow` reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure and content regions of the window and asks the window definition function to calculate those regions.

To create the window, `GetNewCWindow` retrieves the window characteristics from the window resource and then calls the `NewCWindow` function, passing the characteristics as parameters.

The `GetNewCWindow` function creates a window in a color graphics port. Before calling `GetNewCWindow`, verify that Color QuickDraw is available. Your application typically sets up its own global variables reflecting the system setup during initialization by calling the `Gestalt` function. See *Inside Macintosh: Overview* for more information about establishing the local configuration.

SPECIAL CONSIDERATIONS

Note that the `GetNewCWindow` function returns a value of type `WindowPtr`, not `CWindowPtr`.

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to the storage to `GetNewCWindow`, use the procedure `CloseWindow` to close the window and the procedure `DisposePtr`, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

See Listing 4-3 on page 4-28 for an example that calls `GetNewCWindow` to create a new window from a window resource.

For more information about window characteristics and the window resource, see the description of `NewCWindow` beginning on page 4-79 and the description of the `'WIND'` resource in the section "The Window Resource" beginning on page 4-124.

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*. See Listing 4-17 on page 4-61 for an example of closing a document window.

GetNewWindow

Use the GetNewWindow function to create a new window from a window resource when Color QuickDraw is not available. The GetNewWindow function takes the same parameters as GetNewCWindow and returns a value of type WindowPtr. The only difference is that it creates a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics ports are the same size and can be used interchangeably in most Window Manager routines.

```
FUNCTION GetNewWindow (windowID: Integer; wStorage: Ptr;
                        behind: WindowPtr): WindowPtr;
```

windowID	The resource ID of the 'WIND' resource that defines the properties of the window.
wStorage	A pointer to memory space for the window record. If you specify a value of NIL for wStorage, the GetNewWindow function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the wStorage parameter.
behind	A pointer to the window that appears immediately in front of the new window on the desktop. To place a new window in front of all other windows on the desktop, specify a value of Pointer(–1). When you place a window in front of all others, GetNewWindow removes the highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active). To place a new window behind all other windows, specify a value of NIL.

DESCRIPTION

Like GetNewCWindow, GetNewWindow creates a new window from a window resource, but it creates a monochrome window. The GetNewWindow function creates a new window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If GetNewWindow is unable to read the window or window definition function from the resource file, it returns NIL.

If the window's definition function (specified in the window resource) is not already in memory, GetNewWindow reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure and content regions of the window and asks the window definition function to calculate those regions.

To create the window, `GetNewWindow` retrieves the window characteristics from the window resource and then calls the function `NewWindow`, passing the characteristics as parameters.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call `DisposeWindow` to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to `GetNewWindow`, use the procedure `CloseWindow` to close the window and the procedure `DisposePtr`, documented in *Inside Macintosh: Memory,* to dispose of the window record.

SEE ALSO

For more information about window characteristics and the window resource, see the description of `NewWindow` beginning on page 4-82 and the description of the `'WIND'` resource in the section "The Window Resource" beginning on page 4-124.

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory.*

NewCWindow

You can use the `NewCWindow` function to create a window with a specified list of characteristics.

```
FUNCTION NewCWindow (wStorage: Ptr; boundsRect: Rect;
                     title: Str255; visible: Boolean;
                     procID: Integer; behind: WindowPtr;
                     goAwayFlag: Boolean;
                     refCon: LongInt): WindowPtr;
```

`wStorage`	A pointer to the window record. If you specify `NIL` as the value of `wStorage`, `NewCWindow` allocates the window record as a nonrelocatable object in the application heap. You can reduce the chances of heap fragmentation by allocating memory from a block of memory reserved for this purpose by your application and passing a pointer to it in the `wStorage` parameter.
`boundsRect`	A rectangle, in global coordinates, specifying the window's initial size and location. This parameter becomes the port rectangle of the window's graphics port. For the standard window types, the `boundsRect` field defines the content region of the window. The `NewCWindow` function places the origin of the local coordinate system at the upper-left corner of the port rectangle.

Note

The NewCWindow function actually calls the QuickDraw procedure OpenCPort to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by OpenCPort, except for the character font, which is set to the application font instead of the system font. u

title
: A string that specifies the window's title.

  If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters are truncated at the end of the title; if there's no close box, the title is centered and truncated at both ends.

  To suppress the title in a window with a title bar, pass an empty string, not NIL, in the title parameter.

visible
: A Boolean value indicating visibility status: TRUE means that the Window Manager displays the window; FALSE means it does not.

  If the value of the visible parameter is TRUE, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the goAwayFlag parameter is also TRUE and the window is frontmost (that is, if the value of the behind parameter is Pointer(–1)), the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region.

  When you create a window, you typically specify FALSE as the value of the visible parameter. When you're ready to display the window, you call the ShowWindow procedure, described on page 4-88.

procID
: The window's definition ID, which specifies both the window definition function and the variation code within that definition function.

  The Window Manager supports nine standard window types, which are handled by two window definition functions. You can create windows of the standard types by specifying one of the window definition ID constants:

```
CONST
   documentProc      = 0;  {standard document }
                           { window, no zoom box}
   dBoxProc          = 1;  {alert box or modal }
                           { dialog box}
   plainDBox         = 2;  {plain box}
   altDBoxProc       = 3;  {plain box with shadow}
   noGrowDocProc     = 4;  {movable window, }
                           { no size box or zoom box}
   movableDBoxProc   = 5;  {movable modal dialog box}
   zoomDocProc       = 8;  {standard document window}
   zoomNoGrow        = 12; {zoomable, nonresizable }
                           { window}
   rDocProc          = 16; {rounded-corner window}
```

For a description of the nine standard window types, see “Types of Windows” beginning on page 4-8.

You can control the diameter of curvature of rounded-corner windows by adding an integer to the `rDocProc` constant, as described in “The Window Resource” beginning on page 4-124.

`behind`	A pointer to the window that appears immediately in front of the new window on the desktop. To place a new window in front of all other windows on the desktop, specify a value of `Pointer(–1)`. When you place a new window in front of all others, `NewCWindow` removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application’s updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active). To place a new window behind all other windows, specify a value of `NIL`.
`goAwayFlag`	A Boolean value that determines whether the window has a close box. If the value of `goAwayFlag` is `TRUE` and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of `goAwayFlag` is `FALSE` or the window type does not support a close box, it does not.
`refCon`	The window’s reference constant, set and used only by your application. (See “Managing Multiple Windows” beginning on page 4-23 for some suggested ways to use the `refCon` parameter.)

DESCRIPTION

The `NewCWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `NewCWindow` is unable to read the window definition function from the resource file, it returns `NIL`.

The `NewCWindow` function looks for a `'wctb'` resource with the same resource ID as the `'WIND'` resource. If it finds one, it uses the window color information in the `'wctb'` resource for coloring the window frame and highlighting.

If the window’s definition function is not already in memory, `NewCWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

The `NewCWindow` function creates a window in a color graphics port. Creating color windows whenever possible ensures that your windows appear on color monitors with whatever color options the user has selected. Before calling `GetNewCWindow`, verify that Color QuickDraw is available. Your application typically sets up its own set of global variables reflecting the system setup during initialization by calling the `Gestalt` function. See the chapter *Inside Macintosh: Overview* for more information about establishing the local configuration.

Note that the function `NewCWindow` returns a value of type `WindowPtr`, not `CWindowPtr`.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call the `DisposeWindow` procedure to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewCWindow`, use the `CloseWindow` procedure to close the window and the `DisposePtr` procedure, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*.

NewWindow

Use the `NewWindow` function to create a new window with the characteristics specified by a list of parameters when Color QuickDraw is not available. The `NewWindow` function takes the same parameters as `NewCWindow` and, like `NewCWindow`, returns a `WindowPtr` as its function result. The only difference is that `NewWindow` creates a window in a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics ports are the same size and can be used interchangeably in most Window Manager routines.

```
FUNCTION NewWindow (wStorage: Ptr; boundsRect: Rect;
                     title: Str255; visible: Boolean;
                     theProc: Integer; behind: WindowPtr;
                     goAwayFlag: Boolean;
                     refCon: LongInt): WindowPtr;
```

`wStorage` A pointer to the window record. If you specify `NIL` as the value of `wStorage`, `NewWindow` allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the storage from a block of memory reserved for this purpose by your application and passing a pointer to it in the `wStorage` parameter.

`boundsRect`	A rectangle, in global coordinates, specifying the window's initial size and location. This parameter becomes the port rectangle of the window's graphics port. For the standard window types, `boundsRect` defines the content region of the window. The `NewWindow` function places the origin of the local coordinate system at the upper-left corner of the port rectangle. **Note** The `NewWindow` function actually calls the QuickDraw procedure `OpenPort` to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by `OpenPort`, except for the character font, which is set to the application font instead of the system font. The coordinates of the graphics port's port boundaries and visible region are changed along with its port rectangle. u
`title`	A string that specifies the window's title. If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters at the end of the title are truncated; if there's no close box, the title is centered and truncated at both ends. To suppress the title in a window with a title bar, pass an empty string, not `NIL`.
`visible`	A Boolean value indicating visibility status: `TRUE` means that the Window Manager displays the window; `FALSE` means it does not. If the value of the `visible` parameter is `TRUE`, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the `goAwayFlag` parameter (described below) is also `TRUE` and the window is frontmost (that is, if the value of the `behind` parameter is `Pointer(-1)`), the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region. When you create a window, you typically specify `FALSE` as the value of the `visible` parameter. When you're ready to display the window, you call the `ShowWindow` procedure, described on page 4-88.
`theProc`	The window's definition ID, which specifies both the window definition function and the variation code for that definition function. The Window Manager supports nine standard window types, which are handled by two window definition functions. You can create windows of the standard types by specifying one of the type constants:

```
CONST
   documentProc      = 0;  {standard document }
                           { window, no zoom box}
   dBoxProc          = 1;  {alert box or modal }
                           { dialog box}
   plainDBox         = 2;  {plain box}
```

```
altDBoxProc       = 3;  {plain box with shadow}
noGrowDocProc     = 4;  {movable window, }
                        { no size box or zoom box}
movableDBoxProc   = 5;  {movable modal dialog box}
zoomDocProc       = 8;  {standard document window}
zoomNoGrow        = 12; {zoomable, nonresizable }
                        { window}
rDocProc          = 16; {rounded-corner window}
```

You can control the diameter of curvature of rounded-corner windows by adding an integer to the `rDocProc` constant, as described in "The Window Resource" beginning on page 4-124.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(–1)`. When you place a new window in front of all others, `NewWindow` removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application's updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

`goAwayFlag` A Boolean value that determines whether or not the window has a close box. If the value of `goAwayFlag` is `TRUE` and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of `goAwayFlag` is `FALSE` or the window type does not support a close box, it does not.

`refCon` The window's reference constant, set and used only by your application. (See "Managing Multiple Windows" beginning on page 4-23 for some suggested ways to use the `refCon` parameter.)

DESCRIPTION

The `NewWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `NewWindow` is unable to read the window definition function from the resource file, it returns `NIL`.

If the window's definition function is not already in memory, `NewWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call the `DisposeWindow` procedure to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewCWindow`, use the `CloseWindow` procedure to close the window and the `DisposePtr` procedure, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*.

Naming Windows

This section describes the procedures that set and retrieve a window's title.

SetWTitle

Use the `SetWTitle` procedure to change a window's title.

```
PROCEDURE SetWTitle (theWindow: WindowPtr; title: Str255);
```

`theWindow`	A pointer to the window's window record.
`title`	The new window title.

DESCRIPTION

The `SetWTitle` procedure changes a window's title to the specified string, both in the window record and on the screen, and redraws the window's frame as necessary.

When the user opens a previously saved document, you typically create a new (invisible) window with the title "untitled" and then call `SetWTitle` to give the window the document's name before displaying it. You also call `SetWTitle` when the user saves a document under a new name.

To suppress the title in a window with a title bar, pass an empty string, not `NIL`.

Always use `SetWTitle` instead of directly changing the title in a window's window record.

GetWTitle

Use the `GetWTitle` procedure to retrieve a window's title.

```
PROCEDURE GetWTitle (theWindow: WindowPtr; VAR title: Str255);
```

`theWindow`	A pointer to the window record.
`title`	The window title.

DESCRIPTION

The `GetWTitle` procedure returns the title of the window in the `title` parameter.

Your application seldom needs to determine a window's title. It might need to do so, however, when presenting user dialog boxes during operations that can affect multiple files. A spell-checking command, for example, might display a dialog box that lets the user select from all currently open documents.

When you need to retrieve a window's title, you should always use `GetWTitle` instead of reading the title from a window's window record.

Displaying Windows

This section describes the Window Manager routines that change a window's display and position in the window list but not its size or location on the desktop. Note that the Window Manager automatically draws all visible windows on the screen.

Your application typically uses only a few of the routines described in this section: `DrawGrowIcon`, `SelectWindow`, `ShowWindow`, and, occasionally, `HideWindow`.

DrawGrowIcon

Use the `DrawGrowIcon` procedure to draw a window's size box.

```
PROCEDURE DrawGrowIcon (theWindow: WindowPtr);
```

`theWindow`	A pointer to the window record.

DESCRIPTION

The `DrawGrowIcon` procedure draws a window's size box or, if the window can't be sized, whatever other image is appropriate. You call `DrawGrowIcon` when drawing the content region of a window that contains a size box.

The exact appearance and location of the image depend on the window type and the window's active or inactive state. The `DrawGrowIcon` procedure automatically checks the window's type and state and draws the appropriate image.

In an active document window, `DrawGrowIcon` draws the grow image in the size box in the lower-right corner of the window's graphics port rectangle, along with the lines delimiting the size box and scroll bar areas. To draw the size box but not the scroll bar outline, set the `clipRgn` field in the window's graphics port to be a 15-by-15 pixel rectangle in the lower-right corner of the window.

The `DrawGrowIcon` procedure doesn't erase the scroll bar areas. If you use `DrawGrowIcon` to draw the size box and scroll bar outline, therefore, you should erase those areas yourself when the window size changes, even if the window doesn't contain scroll bars.

In an inactive document window, `DrawGrowIcon` draws the lines delimiting the size box and scroll bar areas and erases the size box.

SEE ALSO

See Listing 4-8 on page 4-39 for an example that draws a window's content region, including the size box. See Listing 4-11 on page 4-51 for an example that calls `DrawGrowIcon` to remove the size-box icon when a window becomes inactive.

SelectWindow

Use the `SelectWindow` procedure to make a window active. The `SelectWindow` procedure changes the active status of a window but does not affect its visibility.

```
PROCEDURE SelectWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `SelectWindow` procedure removes highlighting from the previously active window, brings the specified window to the front, highlights it, and generates the activate events to deactivate the previously active window and activate the specified window. If the specified window is already active, `SelectWindow` has no effect.

Even if the specified window is invisible, `SelectWindow` brings the window to the front, activates the window, and deactivates the previously active window. Note that in this case, no active window is visible on the screen. If you do select an invisible window, be sure to call `ShowWindow` immediately to make the window visible (and accessible to the user).

Call `SelectWindow` when the user presses the mouse button while the cursor is in the content region of an inactive window.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `SelectWindow` to change the active window when the user presses the mouse button while the cursor is in an inactive window.

See Listing 4-18 on page 4-64 for an example that uses `SelectWindow` and `ShowWindow` together to restore a window's active, visible status after it has been made invisible with `HideWindow`.

ShowWindow

Use the `ShowWindow` procedure to make an invisible window visible.

```
PROCEDURE ShowWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record of the window.

DESCRIPTION

The `ShowWindow` procedure makes an invisible window visible. If the specified window is already visible, `ShowWindow` has no effect. Your application typically creates a new window in an invisible state, performs any necessary setup of the content region, and then calls `ShowWindow` to make the window visible.

When you display a previously invisible window by calling `ShowWindow`, the Window Manager draws the window frame and then generates an update event to trigger your application's drawing of the content region.

If the newly visible window is the frontmost window, `ShowWindow` highlights it if it's not already highlighted and generates an activate event to make it active. The `ShowWindow` procedure does not activate a window that is not frontmost on the desktop.

Note

Because `ShowWindow` does not change the front-to-back ordering of windows, it is not the inverse of `HideWindow`. If you make the frontmost window invisible with `HideWindow`, and `HideWindow` has activated another window, you must call both `ShowWindow` and `SelectWindow` to bring the original window back to the front. u

SEE ALSO

See Listing 4-16 on page 4-60 for an example that temporarily hides a dialog box window when the user closes it. See Listing 4-18 on page 4-64 for the example that calls `ShowWindow` to display the window again later.

HideWindow

Use the `HideWindow` procedure to make a window invisible.

```
PROCEDURE HideWindow (theWindow: WindowPtr);
```

`theWindow`	A pointer to the window's window record.

DESCRIPTION

The `HideWindow` procedure make a visible window invisible. If you hide the frontmost window, `HideWindow` removes the highlighting, brings the window behind it to the front, highlights the new frontmost window, and generates the appropriate activate events.

To reverse the actions of `HideWindow`, you must call both `ShowWindow`, to make the window visible, and `SelectWindow`, to select it.

SEE ALSO

See Listing 4-16 on page 4-60 for an example that calls `HideWindow` to temporarily hide a dialog box window when the user closes it. See Listing 4-18 on page 4-64 for the companion example that redisplays the window later.

ShowHide

Use the `ShowHide` procedure to set a window's visibility status.

```
PROCEDURE ShowHide (theWindow: WindowPtr; showFlag: Boolean);
```

`theWindow`	A pointer to the window's window record.
`showFlag`	A Boolean value that determines visibility status: `TRUE` makes a window visible; `FALSE` makes it invisible.

DESCRIPTION

The `ShowHide` procedure sets a window's visibility to the status specified by the `showFlag` parameter. If the value of `showFlag` is `TRUE`, `ShowHide` makes the window visible if it's not already visible and has no effect if it's already visible. If the value of `showFlag` is `FALSE`, `ShowHide` makes the window invisible if it's not already invisible and has no effect if it's already invisible.

The `ShowHide` procedure never changes the highlighting or front-to-back ordering of windows and generates no activate events.

s **WARNING**
Use this procedure carefully and only in special circumstances where you need more control than that provided by `HideWindow` and `ShowWindow`. Do not, for example, use `ShowHide` to hide the active window without making another window active. s

HiliteWindow

Use the `HiliteWindow` procedure to set a window's highlighting status.

```
PROCEDURE HiliteWindow (theWindow: WindowPtr; fHilite: Boolean);
```

`theWindow`	A pointer to the window's window record.
`fHilite`	A Boolean value that determines the highlighting status: `TRUE` highlights a window; `FALSE` removes highlighting.

DESCRIPTION

The `HiliteWindow` procedure sets a window's highlighting status to the specified state. If the value of the `fHilite` parameter is `TRUE`, `HiliteWindow` highlights the specified window; if the specified window is already highlighted, the procedure has no effect. If the value of `fHilite` is `FALSE`, `HiliteWindow` removes highlighting from the specified window; if the window is not already highlighted, the procedure has no effect.

Your application doesn't normally need to call `HiliteWindow`. To make a window active, you can call `SelectWindow`, which handles highlighting for you.

BringToFront

Use the `BringToFront` procedure to bring a window to the front.

```
PROCEDURE BringToFront (theWindow: WindowPtr);
```

`theWindow`	A pointer to the window's window record.

DESCRIPTION

The `BringToFront` procedure puts the specified window at the beginning of the window list and redraws the window in front of all others on the screen. It does not change the window's highlighting or make it active.

Your application does not ordinarily call `BringToFront`. The user interface guidelines specify that the frontmost window should be the active window. To bring a window to the front and make it active, call the `SelectWindow` procedure.

SendBehind

Use the `SendBehind` procedure to move one window behind another.

```
PROCEDURE SendBehind (theWindow, behindWindow: WindowPtr);
```

`theWindow`	A pointer to the window to be moved.
`behindWindow`	A pointer to the window that is to be in front of the moved window.

DESCRIPTION

The `SendBehind` procedure moves the window pointed to by the parameter `theWindow` behind the window pointed to by the parameter `behindWindow`. If the move exposes previously obscured windows or parts of windows, `SendBehind` redraws the frames as necessary and generates the appropriate update events to have any newly exposed content areas redrawn.

If the value of `behindWindow` is `NIL`, `SendBehind` sends the window to be moved behind all other windows on the desktop. If the window to be moved is the active window,`SendBehind` removes its highlighting, highlights the newly exposed frontmost window, and generates the appropriate activate events.

Note
Do not use `SendBehind` to deactivate a window after you've made a new window active with the `SelectWindow` procedure. The `SelectWindow` procedure automatically deactivates the previously active window. u

Retrieving Window Information

This section describes

- the `FindWindow` function, which maps the cursor location of a mouse-down event to parts of the screen or regions of a window
- the `FrontWindow` function, which tells your application which window is active

FindWindow

When your application receives a mouse-down event, call the `FindWindow` function to map the location of the cursor to a part of the screen or a region of a window.

```
FUNCTION FindWindow (thePoint: Point;
                     VAR theWindow: WindowPtr): Integer;
```

`thePoint`	The point, in global coordinates, where the mouse-down event occurred. Your application retrieves this information from the `where` field of the event record.
`theWindow`	A parameter in which `FindWindow` returns a pointer to the window in which the mouse-down event occurred, if it occurred in a window. If it didn't occur in a window, `FindWindow` sets `theWindow` to `NIL`.

DESCRIPTION

The `FindWindow` function returns an integer that specifies where the cursor was when the user pressed the mouse button. You typically call `FindWindow` whenever you receive a mouse-down event. The `FindWindow` function helps you dispatch the event by reporting whether the cursor was in the menu bar or in a window when the mouse button was pressed and, if it was in a window, which window and which region of the window. If the mouse-down event occurred in a window, `FindWindow` places a pointer to the window in the parameter `theWindow`.

The `FindWindow` function returns an integer that specifies one of nine regions:

```
CONST inDesk      = 0;  {none of the following}
      inMenuBar   = 1;  {in menu bar}
      inSysWindow = 2;  {in desk accessory window}
      inContent   = 3;  {anywhere in content region except size }
                        { box if window is active, }
                        { anywhere including size box if window }
                        { is inactive}
      inDrag      = 4;  {in drag (title bar) region}
      inGrow      = 5;  {in size box (active window only)}
      inGoAway    = 6;  {in close box}
      inZoomIn    = 7;  {in zoom box (window in standard state)}
      inZoomOut   = 8;  {in zoom box (window in user state)}
```

The `FindWindow` function returns `inDesk` if the cursor is not in the menu bar, a desk accessory window, or any window that belongs to your application. The `FindWindow` function might return this value if, for example, the user presses the mouse button while the cursor is on the window frame but not in the title bar, close box, or zoom box. When `FindWindow` returns `inDesk`, your application doesn't need to do anything. In System 7, when the user presses the mouse button while the cursor is on the desktop or in a window that belongs to another application, the Event Manager sends your application a suspend event and switches to the Finder or another application.

The `FindWindow` function returns `inMenuBar` when the user presses the mouse button with the cursor in the menu bar. Your application typically adjusts its menus and then calls the Menu Manager's function `MenuSelect` to let the user choose menu items.

The FindWindow function returns `inSysWindow` when the user presses the mouse button while the cursor is in a window belonging to a desk accessory that was launched in your application's partition. This situation seldom arises in System 7. When the user clicks in a window belonging to a desk accessory launched independently, the Event Manager sends your application a suspend event and switches to the desk accessory.

If `FindWindow` does return `inSysWindow`, your application calls the `SystemClick` procedure, documented in the chapter "Event Manager" in this book. The `SystemClick` procedure routes the event to the desk accessory. If the user presses the mouse button with the cursor in the content region of an inactive desk accessory window, `SystemClick` makes the window active by sending your application and the desk accessory the appropriate activate events.

The `FindWindow` function returns `inContent` when the user presses the mouse button with the cursor in the content area (excluding the size box in an active window) of one of your application's windows. Your application then calls its routine for handling clicks in the content region.

The `FindWindow` function returns `inDrag` when the user presses the mouse button with the cursor in the drag region of a window (that is, the title bar, excluding the close box and zoom box). Your application then calls the Window Manager's `DragWindow` procedure to let the user drag the window to a new location.

The `FindWindow` function returns `inGrow` when the user presses the mouse button with the cursor in an active window's size box. Your application then calls its own routine for resizing a window.

The `FindWindow` function returns `inGoAway` when the user presses the mouse button with the cursor in an active window's close box. Your application calls the `TrackGoAway` function to track mouse activity while the button is down and then calls its own routine for closing a window if the user releases the button while the cursor is in the close box.

The `FindWindow` function returns `inZoomIn` or `inZoomOut` when the user presses the mouse button with the cursor in an active window's zoom box. Your application calls the `TrackBox` function to track mouse activity while the button is down and then calls its own routine for zooming a window if the user releases the button while the cursor is in the zoom box.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `FindWindow` to determine the location of the cursor and then dispatches the mouse-down event depending on the results.

FrontWindow

Use the `FrontWindow` function to find out which window is active.

```
FUNCTION FrontWindow: WindowPtr;
```

DESCRIPTION

The `FrontWindow` function returns a pointer to the first visible window in the window list (that is, the active window). If there are no visible windows, `FrontWindow` returns `NIL`.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `FrontWindow` to determine whether an event occurred in the active window.

See Listing 4-12 on page 4-55 for an example that calls `FrontWindow` to determine whether to display a window in front of other windows after changing its size.

See Listing 4-16 on page 4-60 and Listing 4-17 on page 4-61 for examples that call `FrontWindow` to determine which window is affected by a user command directed at the active window.

Moving Windows

This section describes the procedures that move windows on the desktop.

To move a window, your application ordinarily needs to call only the `DragWindow` procedure, which itself calls the `DragGrayRgn` function, and the `MoveWindow` procedure. The `DragGrayRgn` function drags a dotted outline of the window on the screen, following the motion of the cursor, as long as the user holds down the mouse button. The `DragGrayRgn` function itself calls the `PinRect` function to contain the point where the cursor was when the mouse button was first pressed inside the available desktop area. When the user releases the mouse button, `DragWindow` calls `MoveWindow`, which moves the window to a new location.

DragWindow

When the user drags a window by its title bar, use the `DragWindow` procedure to move the window on the screen.

```
PROCEDURE DragWindow (theWindow: WindowPtr;
                      startPt: Point; boundsRect: Rect);
```

`theWindow`	A pointer to the window record of the window to be dragged.
`startPt`	The location, in global coordinates, of the cursor at the time the user pressed the mouse button. Your application retrieves this point from the `where` field of the event record.

`boundsRect` A rectangle, in global coordinates, that limits the region to which a window can be dragged. If the mouse button is released when the cursor is outside the limits of `boundsRect`, `DragWindow` returns without moving the window (or, if it was inactive, without making it the active window).

Because the user cannot ordinarily move the cursor off the desktop, you can safely set `boundsRect` to the largest available rectangle (the bounding box of the desktop region pointed to by the global variable `GrayRgn`) when you're using `DragWindow` to track mouse movements. Don't set the bounding rectangle to the size of the immediate screen (`screenBits.bounds`), because the user wouldn't be able to move the window to a different screen on a system equipped with multiple monitors.

DESCRIPTION

The `DragWindow` procedure moves a dotted outline of the specified window around the screen, following the movement of the cursor until the user releases the mouse button. When the button is released, `DragWindow` calls `MoveWindow` to move the window to its new location. If the specified window isn't the active window (and the Command key wasn't down when the mouse button was pressed), `DragWindow` makes it the active window by setting the `front` parameter to `TRUE` when calling `MoveWindow`. If the Command key was down when the mouse button was pressed, `DragWindow` moves the window without making it active.

SEE ALSO

The `DragWindow` procedure calls both `MoveWindow` and `DragGrayRgn`, which are described in this section.

See Listing 4-9 on page 4-44 for an example that calls `DragWindow` when the user presses the mouse button while the cursor is in the drag region.

MoveWindow

Use the `MoveWindow` procedure to move a window on the desktop.

```
PROCEDURE MoveWindow (theWindow: WindowPtr;
                      hGlobal, vGlobal: Integer;
                      front: Boolean);
```

`theWindow`	A pointer to the window record of the window being moved.
`hGlobal`	The new location, in global coordinates, of the left edge of the window's port rectangle.
`vGlobal`	The new location, in global coordinates, of the top edge of the window's port rectangle.

`front`	A Boolean value specifying whether the window is to become the frontmost, active window. If the value of the front parameter is `FALSE`, `MoveWindow` does not change its plane or status. If the value of the front parameter is `TRUE` and the window isn't active, `MoveWindow` makes it active by calling the `SelectWindow` procedure.

DESCRIPTION

The `MoveWindow` procedure moves the specified window to the location specified by the `hGlobal` and `vGlobal` parameters, without changing the window's size. The upper-left corner of the window's port rectangle is placed at the point (`vGlobal`,`hGlobal`). The local coordinates of the upper-left corner are unaffected.

Your application doesn't normally call `MoveWindow`. When the user drags a window by dragging its title bar, you can call `DragWindow`, which in turn calls `MoveWindow` when the user releases the mouse button.

DragGrayRgn

The `DragWindow` function calls the `DragGrayRgn` function to move an outline of a window around the screen as the user drags a window.

```
FUNCTION DragGrayRgn (theRgn: RgnHandle; startPt: Point;
                        limitRect, slopRect: Rect; axis: Integer;
                        actionProc: ProcPtr): LongInt;
```

`theRgn`	A handle to the region to be dragged.
`startPt`	The location, in the local coordinates of the current graphics port, of the cursor when the mouse button was pressed.
`limitRect`	A rectangle, in the local coordinates of the current graphics port, that limits where the region can be dragged. This parameter works in conjunction with the `slopRect` parameter, as illustrated in Figure 4-23 on page 4-98.
`slopRect`	A rectangle, in the local coordinates of the current graphics port, that gives the user some leeway in moving the mouse without violating the limits of the `limitRect` parameter, as illustrated in Figure 4-23 on page 4-98. The `slopRect` rectangle should be larger than the `limitRect` rectangle.
`axis`	A constant that constrains the region's motion. The `axis` parameter can have one of these values:

```
CONST noConstraint   = 0;  {no constraints}
      hAxisOnly      = 1;  {move on horizontal axis }
                           { only}
      vAxisOnly      = 2;  {move on vertical axis }
                           { only}
```

	If an axis constraint is in effect, the outline follows the cursor's movements along only the specified axis, ignoring motion along the other axis. With or without an axis constraint, the outline appears only when the mouse is inside the `slopRect` rectangle.
`actionProc`	A pointer to a procedure that defines an action to be performed repeatedly as long as the user holds down the mouse button. The procedure can have no parameters. If the value of `actionProc` is `NIL`, `DragGrayRgn` simply retains control until the mouse button is released.

DESCRIPTION

The `DragGrayRgn` function moves a gray outline of a region on the screen, following the movements of the cursor, until the mouse button is released. It returns the difference between the point where the mouse button was pressed and the **offset point**—that is, the point in the region whose horizontal and vertical offsets from the upper-left corner of the region's enclosing rectangle are the same as the offsets of the starting point when the user pressed the mouse button. The `DragGrayRgn` function stores the vertical difference between the starting point and the offset point in the high-order word of the return value and the horizontal difference in the low-order word.

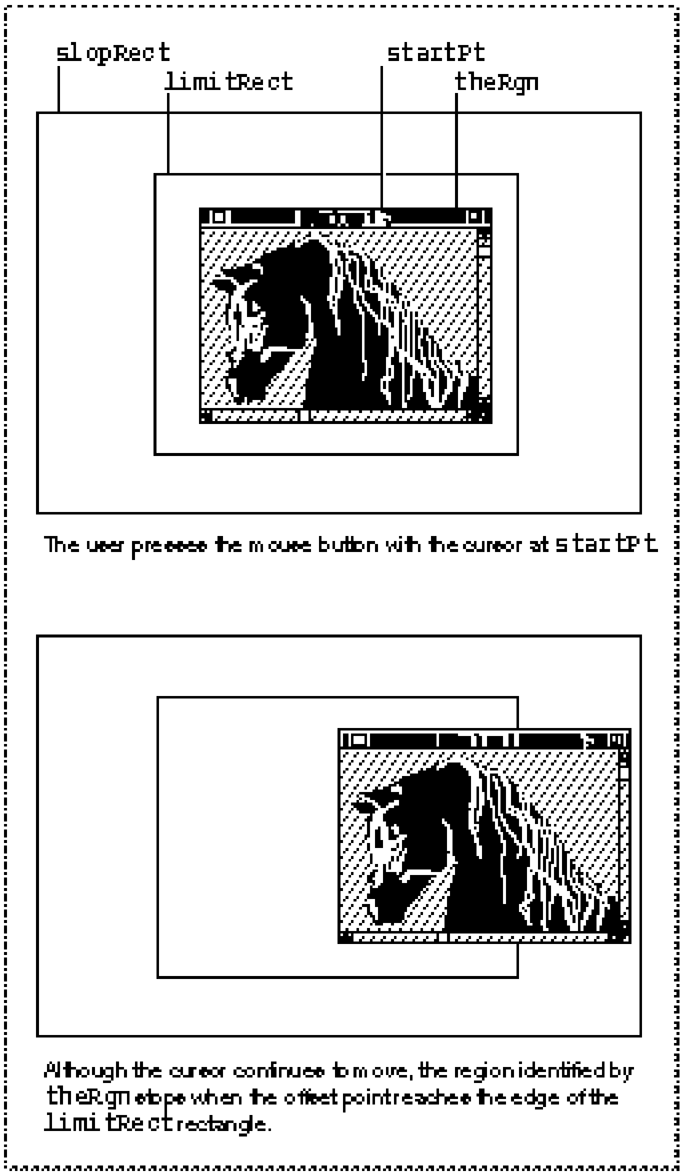
The `DragGrayRgn` function limits the movement of the region according to the constraints set by the `limitRect` and `slopRect` parameters:

- As long as the cursor is inside the `limitRect` rectangle, the region's outline follows it normally. If the mouse button is released while the cursor is within this rectangle, the return value reflects the simple distance that the cursor moved in each dimension.
- When the cursor moves outside the `limitRect` rectangle, the offset point stops at the edge of the `limitRect` rectangle. If the mouse button is released while the cursor is outside the `limitRect` rectangle but inside the `slopRect` rectangle, the return value reflects only the difference between the starting point and the offset point, regardless of how far outside of the `limitRect` rectangle the cursor may have moved. (Note that part of the region can fall outside the `limitRect` rectangle, but not the offset point.)
- When the cursor moves outside the `slopRect` rectangle, the region's outline disappears from the screen. The `DragGrayRgn` function continues to track the cursor, however, and if the cursor moves back into the `slopRect` rectangle, the outline reappears. If the mouse button is released while the cursor is outside the `slopRect` rectangle, both words of the return value are set to $8000. In this case, the Window Manager does not move the window from its original location.

Figure 4-23 on page 4-98 illustrates how the region stops moving when the offset point reaches the edge of the `limitRect` rectangle. The cursor continues to move, but the region does not.

If the mouse button is released while the cursor is anywhere inside the `slopRect` rectangle, the Window Manager redraws the window in its new location, which is calculated from the value returned by `DragGrayRgn`.

Figure 4-23 Limiting rectangle used by `DragGrayRgn`



The user presses the mouse button with the cursor at startPt

Although the cursor continues to move, the region identified by theRgn stops when the offset point reaches the edge of the limitRect rectangle.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `DragGrayRgn` as long as the mouse button is held down. (If there's an `actionProc` procedure, it is called first.) If you want `DragGrayRgn` to draw the region's outline in a pattern other than gray, you can store the pattern in the global variable `DragPattern` and then invoke the macro `_DragTheRgn`. Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

PinRect

The `DragGrayRgn` function uses the `PinRect` function to contain a point within a specified rectangle.

```
FUNCTION PinRect (theRect: Rect; thePt: Point): LongInt;
```

`theRect`	The rectangle in which the point is to be contained.
`thePt`	The point to be contained.

DESCRIPTION

The `PinRect` function returns a point within the specified rectangle that is as close as possible to the specified point. (The high-order word of the returned long integer is the vertical coordinate; the low-order word is the horizontal coordinate.)

If the specified point is within the rectangle, `PinRect` returns the point itself. If not, then

- if the horizontal position is to the left of the rectangle, `PinRect` returns the left edge as the horizontal coordinate
- if the horizontal position is to the right of the rectangle, `PinRect` returns the right edge minus 1 as the horizontal coordinate
- if the vertical position is above the rectangle, `PinRect` returns the top edge as the vertical coordinate
- if the vertical position is below the rectangle, `PinRect` returns the bottom edge minus 1 as the vertical coordinate

Note

The 1 is subtracted when the point is below or to the right of the rectangle so that a pixel drawn at that point lies within the rectangle. If the point is exactly on the bottom or the right edge of the rectangle, however, 1 should be subtracted but isn't. u

Resizing Windows

This section describes the procedures you can use to track the cursor while the user resizes a window and to draw the window in a new size.

GrowWindow

Use the `GrowWindow` function to allow the user to change the size of a window. The `GrowWindow` function displays an outline (grow image) of the window as the user moves the cursor to make the window larger or smaller; it handles all user interaction until the user releases the mouse button. After calling `GrowWindow`, you call the `SizeWindow` procedure to change the size of the window.

```
FUNCTION GrowWindow (theWindow: WindowPtr;
                     startPt: Point; sizeRect: Rect): LongInt;
```

`theWindow`	A pointer to the window record of the window to drag.
`startPt`	The location of the cursor at the time the mouse button was first pressed, in global coordinates. Your application retrieves this point from the `where` field of the event record.
`sizeRect`	The limits on the vertical and horizontal measurements of the port rectangle, in pixels. Although the `sizeRect` parameter is in the form of the `Rect` data type, the four numbers in the structure represent lengths, not screen coordinates. The `top`, `left`, `bottom`, and `right` fields of the `sizeRect` parameter specify the minimum vertical measurement (`top`), the minimum horizontal measurement (`left`), the maximum vertical measurement (`bottom`), and the maximum horizontal measurement (`right`). The minimum measurements must be large enough to allow a manageable rectangle; 64 pixels on a side is typical. Because the user cannot ordinarily move the cursor off the screen, you can safely set the upper bounds to the largest possible length (65,535 pixels) when you're using `GrowWindow` to follow cursor movements.

DESCRIPTION

The `GrowWindow` function moves a dotted-line image of the window's right and lower edges around the screen, following the movements of the cursor until the mouse button is released. It returns the new dimensions, in pixels, of the resulting window: the height in the high-order word of the returned long-integer value and the width in the low-order word. You can use the functions `HiWord` and `LoWord` to retrieve only the high-order and low-order words, respectively.

A return value of 0 means that the new size is the same as the size of the current port rectangle.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `GrowWindow` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that calls `GrowWindow` when the user presses the mouse button while the cursor is in the size box.

SizeWindow

Use the `SizeWindow` procedure to set the size of a window.

```
PROCEDURE SizeWindow (theWindow: WindowPtr; w, h: Integer;
                        fUpdate: Boolean);
```

`theWindow`	A pointer to the window record of the window to be sized.
`w`	The new window width, in pixels.
`h`	The new window height, in pixels.
`fUpdate`	A Boolean value that specifies whether any newly created area of the content region is to be accumulated into the update region (`TRUE`) or not (`FALSE`). You ordinarily pass a value of `TRUE` to ensure that the area is updated. If you pass `FALSE`, you're responsible for maintaining the update region yourself. For more information on adding rectangles to and removing rectangles from the update region, see the description of `InvalRect` on page 4-107 and `ValidRect` on page 4-108.

DESCRIPTION

The `SizeWindow` procedure changes the size of the window's graphics port rectangle to the dimensions specified by the `w` and `h` parameters, or does nothing if the values of `w` and `h` are 0. The Window Manager redraws the window in the new size, recentering the title and truncating it if necessary. Your application calls `SizeWindow` immediately after calling `GrowWindow`, to adjust the window to any changes made by the user through the size box.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that calls `SizeWindow` to resize a window based on the return value of `GrowWindow`.

Zooming Windows

This section describes the procedures you can use to track mouse activity in the zoom box and to zoom windows.

TrackBox

Use the `TrackBox` function to track the cursor when the user presses the mouse button while the cursor is in the zoom box.

```
FUNCTION TrackBox (theWindow: WindowPtr; thePt: Point;
                   partCode: Integer): Boolean;
```

`theWindow`	A pointer to the window record of the window in which the mouse button was pressed.
`thePt`	The location of the cursor when the mouse button was pressed. Your application receives this point from the `where` field in the event record.
`partCode`	The part code (either `inZoomIn` or `inZoomOut`) returned by the `FindWindow` function.

DESCRIPTION

The `TrackBox` function tracks the cursor when the user presses the mouse button while the cursor is in the zoom box, retaining control until the mouse button is released. While the button is down, `TrackBox` highlights the zoom box while the cursor is in the zoom region, as illustrated in Figure 4-20 on page 4-47.

When the mouse button is released, `TrackBox` removes the highlighting from the zoom box and returns `TRUE` if the cursor is within the zoom region and `FALSE` if it is not.

Your application calls the `TrackBox` function when it receives a result code of either `inZoomIn` or `inZoomOut` from the `FindWindow` function. If `TrackBox` returns `TRUE`, your application calculates the standard state, if necessary, and calls the `ZoomWindow` procedure to zoom the window. If `TrackBox` returns `FALSE`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `TrackBox` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-12 on page 4-55 for an example that calls `TrackBox` to track cursor activity when the user presses the mouse button while the cursor is in the zoom box.

ZoomWindow

Use the `ZoomWindow` procedure to zoom the window when the user has pressed and released the mouse button with the cursor in the zoom box.

```
PROCEDURE ZoomWindow (theWindow: WindowPtr;
                     partCode: Integer; front: Boolean);
```

`theWindow`	A pointer to the window record of the window to be zoomed.
`partCode`	The result (either `inZoomIn` or `inZoomOut`) returned by the `FindWindow` function.

`front`	A Boolean value that determines whether the window is to be brought to the front. If the value of `front` is `TRUE`, the window necessarily becomes the frontmost, active window. If the value of `front` is `FALSE`, the window's position in the window list does not change. Note that if a window was active before it was zoomed, it remains active even if the value of `front` is `FALSE`.

DESCRIPTION

The `ZoomWindow` procedure zooms a window in or out, depending on the value of the `partCode` parameter. Your application calls `ZoomWindow`, passing it the part code returned by `FindWindow`, when it receives a result of `TRUE` from `TrackBox`. The `ZoomWindow` procedure then changes the window's port rectangle to either the user state (if the part code is `inZoomIn`) or the standard state (if the part code is `inZoomOut`), as stored in the window state data record, described in the section "Zooming a Window" beginning on page 4-53.

If the part code is `inZoomOut`, your application ordinarily calculates and sets the standard state before calling `ZoomWindow`.

For best results, call the QuickDraw procedure `EraseRect`, passing the window's graphics port as the port rectangle, before calling `ZoomWindow`.

SEE ALSO

See Listing 4-12 on page 4-55 for an example that calculates and sets the standard state and then calls `ZoomWindow` to zoom a window.

Closing and Deallocating Windows

This section describes the procedures that track user activity in the close box and that close and dispose of windows.

When you no longer need a window, call the `CloseWindow` procedure if you allocated the memory for the window record or the `DisposeWindow` procedure if you did not.

TrackGoAway

Use the `TrackGoAway` function to track the cursor when the user presses the mouse button while the cursor is in the close box.

```
FUNCTION TrackGoAway (theWindow: WindowPtr;
                      thePt: Point): Boolean;
```

`theWindow`	A pointer to the window record of the window in which the mouse-down event occurred.
`thePt`	The location of the cursor at the time the mouse button was pressed. Your application receives this point from the `where` field of the event record.

DESCRIPTION

The `TrackGoAway` function tracks cursor activity when the user presses the mouse button while the cursor is in the close box, retaining control until the user releases the mouse button. While the button is down, `TrackGoAway` highlights the close box as long as the cursor is in the close region, as illustrated in Figure 4-19 on page 4-46.

When the mouse button is released, `TrackGoAway` removes the highlighting from the close box and returns `TRUE` if the cursor is within the close region and `FALSE` if it is not.

Your application calls the `TrackGoAway` function when it receives a result code of `inGoAway` from the `FindWindow` function. If `TrackGoAway` returns `TRUE`, your application calls its own procedure for closing a window, which can call either the `CloseWindow` procedure or the `DisposeWindow` procedure to remove the window from the screen. (Before removing a document window, your application ordinarily checks whether the document has changed since the associated file was last saved. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for a general discusion of handling files.) If `TrackGoAway` returns `FALSE`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `TrackGoAway` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `TrackGoAway` to track cursor activity when the user presses the mouse button while the cursor is in the close box.

CloseWindow

Use the `CloseWindow` procedure to remove a window if you allocated memory yourself for the window's window record.

```
PROCEDURE CloseWindow (theWindow: WindowPtr);
```

`theWindow`	A pointer to the window record of the window to be closed.

DESCRIPTION

The `CloseWindow` procedure removes the specified window from the screen and deletes it from the window list. It releases the memory occupied by all data structures associated with the window except the window record itself.

If you allocated memory for the window record and passed a pointer to it as one of the parameters to the functions that create windows, call `CloseWindow` when you're done with the window. You must then call the Memory Manager procedure `DisposePtr` to release the memory occupied by the window record.

s **WARNING**

If your application allocated any other memory for use with a window, you must release it before calling `CloseWindow`. The Window Manager releases only the data structures it created.

Also, `CloseWindow` assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the QuickDraw procedure `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` procedure and set the `windowPic` field to `NIL` before closing the window. s

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

SEE ALSO

See Listing 4-17 on page 4-61 for an example that calls `CloseWindow` to remove a window from the screen.

See Listing 4-3 on page 4-28 for an example that calls `CloseWindow` to clean up memory when an attempt to create a new window fails.

DisposeWindow

Use the `DisposeWindow` procedure to remove a window if you let the Window Manager allocate memory for the window record.

```
PROCEDURE DisposeWindow (theWindow: WindowPtr);
```

`theWindow`	A pointer to the window record of the window to be closed.

DESCRIPTION

The `DisposeWindow` procedure removes a window from the screen, deletes it from the window list, and releases the memory occupied by all structures associated with the window, including the window record. (`DisposeWindow` calls `CloseWindow` and then releases the memory occupied by the window record.)

s **WARNING**
If your application allocated any other memory for use with a window, you must release it before calling `DisposeWindow`. The Window Manager releases only the data structures it created.

The `DisposeWindow` procedure assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the QuickDraw procedure `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` procedure and set the `windowPic` field to `NIL` before closing the window. s

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

Maintaining the Update Region

This section describes the routines you use to update your windows and to maintain window update regions.

BeginUpdate

Use the `BeginUpdate` procedure to start updating a window when you receive an update event for that window.

```
PROCEDURE BeginUpdate (theWindow: WindowPtr);
```

`theWindow`	A pointer to the window's window record. Your application gets this information from the `message` field in the update event record.

DESCRIPTION

The `BeginUpdate` procedure limits the visible region of the window's graphics port to the intersection of the visible region and the update region; it then sets the window's update region to an empty region. After calling `BeginUpdate`, your application redraws either the entire content region or only the visible region. In either case, only the parts of the window that require updating are actually redrawn on the screen.

Every call to `BeginUpdate` must be matched with a subsequent call to `EndUpdate` after your application redraws the content region.

Note
In Pascal, `BeginUpdate` and `EndUpdate` can't be nested. That is, you must call `EndUpdate` before the next call to `BeginUpdate`.

You can nest `BeginUpdate` and `EndUpdate` calls in assembly language if you save and restore the copy of the `visRgn`, a copy of which is stored, in global coordinates, in the global variable `SaveVisRgn`. u

SPECIAL CONSIDERATIONS

If you don't clear the update region when you receive an update event, the Event Manager continues to send update events until you do.

SEE ALSO

See Figure 4-21 on page 4-49 for an illustration of how `BeginUpdate` and `EndUpdate` affect the visible region and update region. See Listing 4-10 on page 4-50 for an example that updates a window.

EndUpdate

Use the `EndUpdate` procedure to finish updating a window.

```
PROCEDURE EndUpdate (theWindow: WindowPtr);
```

`theWindow`	A pointer to the window's window record.

DESCRIPTION

The `EndUpdate` procedure restores the normal visible region of a window's graphics port. When you receive an update event for a window, you call `BeginUpdate`, redraw the update region, and then call `EndUpdate`. Each call to `BeginUpdate` must be balanced by a subsequent call to `EndUpdate`.

SEE ALSO

See Figure 4-21 on page 4-49 for an illustration of how `BeginUpdate` and `EndUpdate` affect the visible region and update region. See Listing 4-10 on page 4-50 for an example that updates a window.

InvalRect

Use the `InvalRect` procedure to add a rectangle to a window's update region.

```
PROCEDURE InvalRect (badRect: Rect);
```

`badRect`	A rectangle, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalRect` procedure adds a specified rectangle to the update region of the window whose graphics port is the current port. Specify the rectangle in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Both your application and the Window Manager use the `InvalRect` procedure. When the user enlarges a window, for example, the Window Manager uses `InvalRect` to add the newly created content region to the update region. Your application uses `InvalRect` to add the two rectangles formerly occupied by the scroll bars in the smaller content area.

InvalRgn

Use the `InvalRgn` procedure to add a region to a window's update region.

```
PROCEDURE InvalRgn (badRgn: RgnHandle);
```

`badRgn`	The region, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalRgn` procedure adds a specified region to the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that uses `InvalRgn` to add part of the window's content region to the update region.

ValidRect

Use the `ValidRect` procedure to remove a rectangle from a window's update region.

```
PROCEDURE ValidRect (goodRect: Rect);
```

`goodRect`	A rectangle, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRect` procedure removes a specified rectangle from the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Your application uses `ValidRect` to tell the Window Manager that it has already drawn a rectangle and to cancel any updates accumulated for that area. You can thereby improve response time by reducing redundant redrawing.

Suppose, for example, that you've resized a window that contains a size box and scroll bars. Depending on the dimensions of the newly sized window, the new size box and scroll bar areas may or may not have been accumulated into the window's update region. After calling `SizeWindow`, you can redraw the size box or scroll bars immediately and then call `ValidRect` for the areas they occupy. If they were in fact accumulated into the update region, `ValidRect` removes them so that you do not have to redraw them with the next update event.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that uses `ValidRect` to remove part of the window's content region from the update region.

ValidRgn

Use the `ValidRgn` procedure to remove a specified region from a window's update region.

```
PROCEDURE ValidRgn (goodRgn: RgnHandle);
```

`goodRgn`	A region, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRgn` procedure removes a specified region from the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Setting and Retrieving Other Window Characteristics

This section describes the routines that let you set and retrieve less commonly used fields in the window record.

SetWindowPic

Use the `SetWindowPic` procedure to establish a picture that the Window Manager can draw in a window's content region.

```
PROCEDURE SetWindowPic (theWindow: WindowPtr;
                        Pic: PicHandle);
```

`theWindow`	A pointer to a window's window record.
`Pic`	A handle to the picture to be drawn in the window.

DESCRIPTION

The `SetWindowPic` procedure stores in a window's window record a handle to a picture to be drawn in the window. When the window's content region must be updated, the Window Manager then draws the picture or part of the picture, as necessary, instead of generating an update event.

Note

The `CloseWindow` and `DisposeWindow` procedures assume that any picture pointed to by the window record field `windowPic` is stored as data, not as a resource. If your application uses a picture stored as a resource, you must release the memory it occupies by calling the Resource Manager's `ReleaseResource` procedure and set the `WindowPic` field to `NIL` before you close the window. u

GetWindowPic

Use the `GetWindowPic` function to retrieve a handle to a window's picture.

```
FUNCTION GetWindowPic (theWindow: WindowPtr): PicHandle;
```

`theWindow`	A pointer to the window's window record.

DESCRIPTION

The `GetWindowPic` function returns a handle to the picture to be drawn in a specified window's content region. The handle must have been stored previously with the `SetWindowPic` procedure.

SetWRefCon

Use the `SetWRefCon` procedure to set the `refCon` field of a window record.

```
PROCEDURE SetWRefCon (theWindow: WindowPtr; data: LongInt);
```

`theWindow`	A pointer to the window's window record.
`data`	The data to be placed in the `refCon` field.

DESCRIPTION

The `SetWRefCon` procedure places the specified data in the `refCon` field of the specified window record. The `refCon` field is available to your application for any window-related data it needs to store.

SEE ALSO

See Listing 4-3 on page 4-28 for an example that sets the `refCon` field. See Listing 4-16 on page 4-60 for an example that uses the contents of the `refCon` field.

GetWRefCon

Use the `GetWRefCon` function to retrieve the reference constant from a window's window record.

```
FUNCTION GetWRefCon (theWindow: WindowPtr): LongInt;
```

`theWindow`	A pointer to the window's window record.

DESCRIPTION

The `GetWRefCon` function returns the long integer data stored in the `refCon` field of the specified window record.

SEE ALSO

See the section "Managing Multiple Windows" beginning on page 4-23 for suggested ways to use the `refCon` field. See Listing 4-1 on page 4-25 for an example of an application-defined routine that gets the `refCon` field.

GetWVariant

Use the `GetWVariant` function to retrieve a window's variation code.

```
FUNCTION GetWVariant (theWindow: WindowPtr): Integer;
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWVariant` function returns the variation code of the specified window. Depending on the window's window definition function, the result of `GetWVariant` can represent one of the standard window types listed in the section "Creating a Window" beginning on page 4-25 or a variation code defined by your own window definition function.

SEE ALSO

See "Types of Windows" beginning on page 4-8 for a definition of variation codes. See "The Window Definition Function" beginning on page 4-120 for a detailed description of variation codes.

Manipulating the Desktop

This section describes the routines that let your application retrieve information about the desktop and set the desktop pattern. Ordinarily, your application doesn't need to manipulate any part of the desktop outside of its own windows.

SetDeskCPat

Use the `SetDeskCPat` procedure to set the desktop pattern on a computer that supports Color QuickDraw.

```
PROCEDURE SetDeskCPat (deskPixPat: PixPatHandle);
```

`deskPixPat` A handle to a pixel pattern.

DESCRIPTION

The `SetDeskCPat` procedure sets the desktop pattern to a specified pixel pattern, which can be drawn in more than two colors. After a call to `SetDeskCPat`, the desktop is automatically redrawn in the new pattern. If the specified pattern is a binary pattern (with a pattern type of 0), it is drawn is the current foreground and background colors. If the value of the `deskPixPat` parameter is `NIL`, `SetDeskCPat` uses the standard binary desk pattern (that is, the `'ppat'` resource with resource ID 16).

Note
For compatibility with other Macintosh applications and the system software, applications should ordinarily not change the desktop pattern. u

The Window Manager's desktop-painting routines can paint the desktop either in the binary pattern stored in the global variable `DeskPattern` or in a new pixel pattern. The desktop pattern used at startup is determined by the value of the parameter-RAM bit flag called `pCDeskPat`. If the value of `pCDeskPat` is 0, the Window Manager uses the new pixel pattern; if not, it uses the binary pattern stored in `DeskPattern`. The user can change the color pattern through the General Controls panel, which changes the value of `pCDeskPat`.

GetGrayRgn

Use the `GetGrayRgn` function to retrieve a handle to the current desktop region.

```
FUNCTION GetGrayRgn: RgnHandle;
```

DESCRIPTION

The `GetGrayRgn` function returns a handle to the current desktop region from the global variable `GrayRgn`.

The desktop region represents all available screen space, that is, the desktop area displayed by all monitors attached to the computer. Ordinarily, your application doesn't need to access the desktop region directly.

When your application calls `DragWindow` to let the user drag a window, it can use `GetGrayRgn` to set the limiting rectangle to the entire desktop area.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that uses `GetGrayRgn` to specify the limiting rectangle when calling `DragWindow` to let the user move a window.

GetCWMgrPort

Use the `GetCWMgrPort` procedure to retrieve a pointer to the Window Manager port on a system that supports Color QuickDraw.

```
PROCEDURE GetCWMgrPort (VAR wMgrCPort: CGrafPtr);
```

`wMgrCPort`	A parameter in which `GetCWMgrPort` returns a pointer to the Window Manager port.

DESCRIPTION

The `GetCWMgrPort` procedure places a pointer to the color Window Manager port in the parameter `wMgrCPort`. The `GetCWMgrPort` procedure is available only on computers with Color QuickDraw.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note
Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. u

GetWMgrPort

Use the `GetWMgrPort` procedure to retrieve a pointer to the Window Manager port on a system with only the original monochrome QuickDraw.

```
PROCEDURE GetWMgrPort (VAR wPort: GrafPtr);
```

`wPort`	A parameter in which `GetWMgrPort` returns a pointer to the Window Manager port.

DESCRIPTION

The `GetWMgrPort` procedure places a pointer to the Window Manager port in the parameter `wPort`.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note
Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. u

Manipulating Window Color Information

This section describes the routines you use for setting and retrieving window color information. Your application does not normally change window color information.

SetWinColor

Use the `SetWinColor` procedure to set a window's window color table.

```
PROCEDURE SetWinColor (theWindow: WindowPtr;
                        newColorTable: WCTabHandle);
```

`theWindow` A pointer to the window's window record.

`newColorTable`
A handle to a window color table record, which defines the colors for the window's new color table.

DESCRIPTION

The `SetWinColor` procedure sets a window's color table. If the window has no auxiliary window record, it creates a new one with the specified window color table and adds it to the auxiliary window list. If the window already has an auxiliary record, its window color table is replaced. The Window Manager then redraws the window frame and highlighted text in the new colors and sets the window's background color to the new content color.

If the new color table has the same entries as the default color table, `SetWinColor` changes the auxiliary window record so that it points to the default color table.

Window color table resources (resources of type `'wctb'`) should not be purgeable.

If you specify a value of `NIL` for the parameter `theWindow`, `SetWinColor` changes the default color table in memory. Your application shouldn't, however, change the default color table.

SEE ALSO

For a description of a window color table, see "The Window Color Table Record" on page 4-71. For a description of the auxiliary window record, see "The Auxiliary Window Record" on page 4-73. For a description of the `'wctb'` resource, see "The Window Color Table Resource" on page 4-127.

GetAuxWin

Use the `GetAuxWin` function to retrieve a handle to a window's auxiliary window record.

```
FUNCTION GetAuxWin (theWindow: WindowPtr;
                    VAR awHndl: AuxWinHandle): Boolean;
```

`theWindow` A pointer to the window's window record.

`awHndl` A handle to the window's auxiliary window record.

DESCRIPTION

The `GetAuxWin` function returns a Boolean value that reports whether or not the window has an auxiliary window record, and it sets the variable parameter `awHndl` to the window's auxiliary window record.

If the window has no auxiliary window record, `GetAuxWin` places the default window color table in `awHndl` and returns a value of `FALSE`.

SEE ALSO

For a description of the auxiliary window record, see “The Auxiliary Window Record” on page 4-73.

Low-Level Routines

This section describes the low-level routines that are called by higher-level Window Manager routines. Ordinarily, you won’t need to use these routines.

CheckUpdate

The Event Manager uses the `CheckUpdate` function to scan the window list for windows that need updating.

```
FUNCTION CheckUpdate (VAR theEvent: EventRecord): Boolean;
```

`theEvent`	An event record to be filled in if a window needs updating.

DESCRIPTION

The `CheckUpdate` function scans the window list from front to back, checking for a visible window that needs updating (that is, a visible window whose update region is not empty). If it finds one whose window record contains a picture handle, it redraws the window itself and continues through the list. If it finds a window record whose update region is not empty and whose window record does not contain a picture handle, it stores an update event in the parameter `theEvent` and returns `TRUE`. If it finds no such window, it returns `FALSE`.

The Event Manager is the only software that ordinarily calls `CheckUpdate`.

ClipAbove

The Window Manager uses the `ClipAbove` procedure to determine the clip region of the Window Manager port for displaying a window.

```
PROCEDURE ClipAbove (window: WindowPeek);
```

`window`	A pointer to the window’s complete window record.

DESCRIPTION

The `ClipAbove` procedure sets the clip region of the Window Manager port to be the area of the desktop that intersects the current clip region, minus the structure regions of all the windows in front of the specified window.

The `ClipAbove` procedure retrieves the desktop region from the global variable `GrayRgn`.

SaveOld

The Window Manager uses the `SaveOld` procedure to save a window's current structure and content regions preparatory to updating the window.

```
PROCEDURE SaveOld (window: WindowPeek);
```

`window`	A pointer to the window's complete window record.

DESCRIPTION

The `SaveOld` procedure saves the specified window's current structure region and content region for the `DrawNew` procedure. Each call to `SaveOld` must be balanced by a subsequent call to `DrawNew`.

DrawNew

The Window Manager uses the `DrawNew` procedure to erase and update changed window regions.

```
PROCEDURE DrawNew (window: WindowPeek; update: Boolean);
```

`window`	A pointer to the window's complete window record.
`update`	A Boolean value that determines whether the regions are updated.

DESCRIPTION

The `DrawNew` procedure erases the parts of a window's structure and content regions that are part of the window's former state and part of its new state but not both. That is,

```
(OldStructure XOR NewStructure) UNION (OldContent XOR NewContent)
```

If the update parameter is set to `TRUE`, `DrawNew` also updates the erased regions.

s **WARNING**
In Pascal, `SaveOld` and `DrawNew` are not nestable. s

ASSEMBLY-LANGUAGE INFORMATION

In assembly language, you can nest `SaveOld` and `DrawNew` if you save and restore the values of the global variables `OldStructure` and `OldContent`.

PaintOne

The Window Manager uses the `PaintOne` procedure to redraw the invalid, exposed portions of one window on the desktop.

```
PROCEDURE PaintOne (window: WindowPeek; clobberedRgn: RgnHandle);
```

`window`	A pointer to the window's complete window record.
`clobberedRgn`	A handle to the region that has become invalid.

DESCRIPTION

The `PaintOne` procedure "paints" the invalid portion of the specified window and all windows above it. It draws as much of the window frame as is in `clobberedRgn` and, if some content region is exposed, erases the exposed area (paints it with the background pattern) and adds it to the window's update region. If the value of the `window` parameter is `NIL`, the window is the desktop, and `PaintOne` paints it with the desktop pattern.

ASSEMBLY-LANGUAGE INFORMATION

The global variables `SaveUpdate` and `PaintWhite` are flags used by `PaintOne`. Normally both flags are set. Clearing `SaveUpdate` prevents `clobberedRgn` from being added to the window's update region. Clearing `PaintWhite` prevents `clobberedRgn` from being erased before being added to the update region (this is useful, for example, if the background pattern of the window isn't the background pattern of the desktop). The Window Manager sets both flags periodically, so you should clear the appropriate flags each time you need them to be clear.

PaintBehind

The Window Manager uses the `PaintBehind` procedure to redraw a series of windows in the window list.

```
PROCEDURE PaintBehind (startWindow: WindowPeek;
                        clobberedRgn: RgnHandle);
```

`startWindow`	A pointer to the window's complete window record.
`clobberedRgn`	A handle to the region that has become invalid.

DESCRIPTION

The `PaintBehind` procedure calls `PaintOne` for `startWindow` and all the windows behind `startWindow`, clipped to `clobberedRgn`.

ASSEMBLY-LANGUAGE INFORMATION

Because `PaintBehind` clears the global variable `PaintWhite` before calling `PaintOne`, `clobberedRgn` isn't erased. The `PaintWhite` global variable is reset after the call to `PaintOne`.

CalcVis

The Window Manager uses the `CalcVis` procedure to calculate the visible region of a window.

```
PROCEDURE CalcVis (window: WindowPeek);
```

`window`	A pointer to the window's complete window record.

DESCRIPTION

The `CalcVis` procedure calculates the visible region of the specified window by starting with its content region and subtracting the structure region of each window in front of it.

CalcVisBehind

The Window Manager uses the `CalcVisBehind` procedure to calculate the visible regions of a series of windows.

```
PROCEDURE CalcVisBehind (startWindow: WindowPeek;
                         clobberedRgn: RgnHandle);
```

`startWindow`	A pointer to a window's window record.
`clobberedRgn`	A handle to the desktop region that has become invalid.

DESCRIPTION

The `CalcVisBehind` procedure calculates the visible regions of the window specified by the `startWindow` parameter and all windows behind `startWindow` that intersect `clobberedRgn`. It is called after `PaintBehind`.

Application-Defined Routine

This section describes the window definition function. The Window Manager supplies window definition functions that handle the standard window types described in “Types of Windows” beginning on page 4-8.

The Window Definition Function

If your application defines its own window types, you must supply your own window definition function to handle them. Store your definition function as a resource of type `'WDEF'` with an ID from 128 through 4096. (Window definition function resource IDs 0 and 1 are the default window definition functions; resource IDs 2 through 127 are reserved by Apple Computer, Inc.)

Your window definition function can support up to 16 variation codes, which are identified by integers 0 through 15. To invoke your own window type, you specify the window’s definition ID, which contains the resource ID of the window’s definition function in the upper 12 bits and the variation code in the lower 4 bits. Thus, for a given resource ID and variation code, the window definition ID is

(16 * resource ID) + (variation code)

When you create a window, the Window Manager calls the Resource Manager to access the window definition function. The Resource Manager reads the window definition function into memory and returns a handle to it. The Window Manager stores this handle in the `windowDefProc` field of the window record. (If 24-bit addressing is in effect, the Window Manager stores the variation code in the lower 4 bits of the `windowDefProc` field; if 32-bit addressing is in effect, the Window Manager stores the variation code elsewhere.) Later, when it needs to perform a type-dependent action on the window, the Window Manager calls the window definition function and passes it the variation code as a parameter.

MyWindow

The window definition function is responsible for drawing the window frame, reporting the region where mouse-down events occur, calculating the window’s structure region and content region, drawing the size box, resizing the window frame when the user drags the size box, and performing any customized initialization or disposal tasks.

You can give your window definition function any name you wish. It takes four parameters and returns a result code:

```
FUNCTION MyWindow (varCode: Integer; theWindow: WindowPtr;
                     message: Integer; param: LongInt): LongInt;
```

`varCode`	The window’s variation code.
`theWindow`	A pointer to the window’s window record.

`message` A code for the task to be performed. The `message` parameter has one of these values:

```
CONST
   wDraw       = 0;  {draw window frame}
   wHit        = 1;  {report where mouse-down event }
                     { occurred}
   wCalcRgns   = 2;  {calculate strucRgn and contRgn}
   wNew        = 3;  {perform additional }
                     { initialization}
   wDispose    = 4;  {perform additional disposal }
                     { tasks}
   wGrow       = 5;  {draw grow image during resizing}
   wDrawGIcon  = 6;  {draw size box and scroll bar }
                     { outline}
```

The subsections that follow explain each of these tasks in detail.

`param` Data associated with the task specified by the `message` parameter. If the task requires no data, this parameter is ignored.

Your window definition function performs whatever task is specified by the `message` parameter and returns a function result if appropriate. If the task performed requires no result code, return 0.

The function's entry point must be at the beginning of the function.

You can set up the various tasks as subroutines inside the window definition function, but you're not required to do so.

Drawing the Window Frame

When you receive a `wDraw` message, draw the window frame in the current graphics port, which is the Window Manager port.

You must make certain checks to determine exactly how to draw the frame. If the value of the `visible` field in the window record is `FALSE`, you should do nothing; otherwise, you should examine the `param` parameter and the status flags in the window record:

- If the value of `param` is 0, draw the entire window frame.
- If the value of `param` is 0 and the `hilited` field in the window record is `TRUE`, highlight the frame to show that the window is active.
  - If the value of the `goAwayFlag` field in the window record is also `TRUE`, draw a close box in the window frame.
  - If the value of the `spareFlag` field in the window record is also `TRUE`, draw a zoom box in the window frame.
- If the value of the `param` parameter is `wInGoAway`, add highlighting to, or remove it from, the window's close box. Figure 4-19 on page 4-46 illustrates the close box with and without highlighting as drawn by the Window Manager's window definition function.

- n If the value of the `param` parameter is `wInZoom`, add highlighting to, or remove it from, the window's zoom box. Figure 4-20 on page 4-47 illustrates the zoom box with and without highlighting as drawn by the Window Manager's window definition function.

Note

When the Window Manager calls a window definition function with a message of `wDraw`, it stores a value of type `Integer` in the `param` parameter without clearing the high-order word. When processing the `wDraw` message, use only the low-order word of the `param` parameter. u

The window frame typically but not necessarily includes the window's title, which should be displayed in the system font and system font size. The Window Manager port is already set to use the system font and system font size.

When designing a title bar that includes the window title, allow at least 16 pixels vertically to support localization for script systems in which the system font can be no smaller than 12 points.

Note

Nothing drawn outside the window's structure region is visible. u

Returning the Region of a Mouse-Down Event

When you receive a `wHit` message, you must determine where the cursor was when the mouse button was pressed. The `wHit` message is accompanied by the mouse location, in global coordinates, in the `param` parameter. The vertical coordinate is in the high-order word of the parameter, and the horizontal coordinate is in the low-order word. You return one of these constants:

```
CONST
   wNoHit      = 0;  {none of the following}
   wInContent  = 1;  {in content region (except grow, if active)}
   wInDrag     = 2;  {in drag region}
   wInGrow     = 3;  {in grow region (active window only)}
   wInGoAway   = 4;  {in go-away region (active window only)}
   wInZoomIn   = 5;  {in zoom box for zooming in (active window }
                     { only)}
   wInZoomOut  = 6;  {in zoom box for zooming out (active window }
                     { only)}
```

The return value `wNoHit` might mean (but not necessarily) that the point isn't in the window. The standard window definition functions, for example, return `wNoHit` if the point is in the window frame but not in the title bar.

Return the constants `wInGrow`, `wInGoAway`, `wInZoomIn`, and `wInZoomOut` only if the window is active—by convention, the size box, close box, and zoom box aren't drawn if the window is inactive. In an inactive document window, for example, a mouse-down event in the part of the title bar that would contain the close box if the window were active is reported as `wInDrag`.

Calculating Regions

When you receive the `wCalcRgns` message, you calculate the window's structure and content regions based on the current graphics port's port rectangle. These regions, whose handles are in the `strucRgn` and `contRgn` fields of the window record, are in global coordinates. The Window Manager requests this operation only if the window is visible.

s **WARNING**
When you calculate regions for your own type of window, do not alter the clip region or the visible region of the window's graphics port. The Window Manager and QuickDraw take care of this for you. Altering the clip region or visible region may damage other windows. s

Initializing a New Window

When you receive the `wNew` message, you can perform any type-specific initialization that may be required. If the content region has an unusual shape, for example, you might allocate memory for the region and store the region handle in the `dataHandle` field of the window record. The initialization routine for a standard document window creates the `wStateData` record for storing zooming data.

Disposing of a Window

When you receive the `wDispose` message, you can perform any additional tasks necessary for disposing of a window. You might, for example, release memory that was allocated by the initialization routine. The dispose routine for a standard document window disposes of the `wStateData` record.

Resizing a Window

When you receive the `wGrow` message, draw a grow image of the window. With the `wGrow` message you receive a pointer to a rectangle, in global coordinates, whose upper-left corner is aligned with the port rectangle of the window's graphics port. Your grow image should fit inside the rectangle. As the user drags the mouse, the Window Manager sends repeated `wGrow` messages, so that you can change your grow image to match the changing mouse location.

Draw the grow image in the current graphics port, which is the Window Manager port, in the current pen pattern and pen mode. These are set up (as `gray` and `notPatXor`) to conform to the Macintosh user interface guidelines.

The grow routine for a standard document window draws a dotted (gray) outline of the window and also the lines delimiting the title bar, size box, and scroll bar areas.

Drawing the Size Box

When you receive the `wDrawGIcon` message, you draw the size box in the content region if the window is active—if the window is inactive, draw whatever is appropriate to show that the window cannot currently be sized.

Note
If the size box is located in the window frame instead of the content region, do nothing in response to the `wDrawGIcon` message, instead drawing the size box in response to the `wDraw` message. u

The routine that draws a size box for an active document window draws the size box in the lower-right corner of the port rectangle of the window's graphics port. It also draws lines delimiting the size box and scroll bar areas. For an inactive document window, it erases the size box and draws the delimiting lines.

Resources

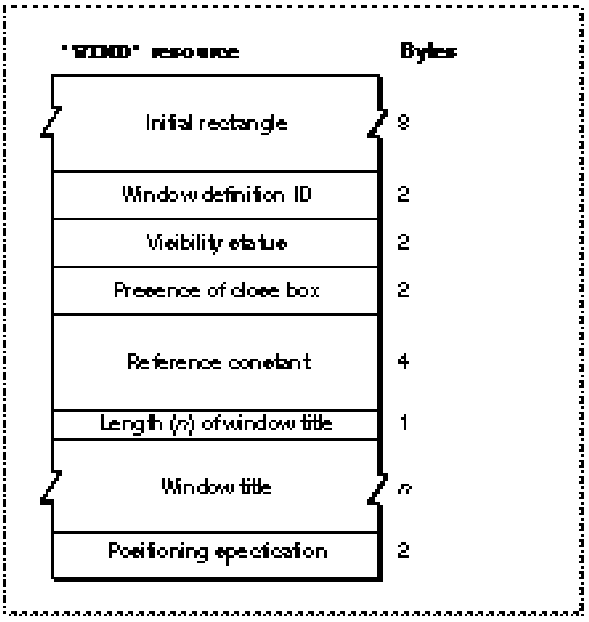
This section describes the resources used by the Window Manager:

- the `'WIND'` resource, used for describing the characteristics of windows
- the `'WDEF'` resource, which holds a window definition function
- the `'wctb'` resource, which defines the colors to be used for a window's frame and highlighting

The Window Resource

You typically define a window resource for each type of window that your application creates. Figure 4-24 illustrates a compiled `'WIND'` resource.

Figure 4-24 Structure of a compiled window (`'WIND'`) resource



A compiled version of a window resource contains the vollowing elements:

- The upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional positioning code described later in this section.
- The window's definition ID, which incorporates both the resource ID of the window definition function that will handle the window and an optional variation code. Together, the window definition function resource ID and the variation code define a window type. Place the resource ID of the window definition function in the upper 12 bits of the definition ID. Window definition functions with IDs 0 through 127 are reserved for use by Apple Computer, Inc. Place the optional variation code in the lower 4 bits of the definition ID.

  If you're using one of the standard window types (described in "Types of Windows" beginning on page 4-8), the definition ID is one of the window-type constants:

```
CONST
documentProc      = 0;  {movable, sizable window, }
                        { no zoom box}
dBoxProc          = 1;  {alert box or modal dialog box}
plainDBox         = 2;  {plain box}
altDBoxProc       = 3;  {plain box with shadow}
noGrowDocProc     = 4;  {movable window, no size box or }
                        { zoom box}
movableDBoxProc   = 5;  {movable modal dialog box}
zoomDocProc       = 8;  {standard document window}
zoomNoGrow        = 12; {zoomable, nonresizable window}
rDocProc          = 16; {rounded-corner window}
```

  You can also add a zoom box to a movable modal dialog box by specifying the sum of two constants: `movableDBoxProc + zoomDocProc`, but a zoom box is not recommended on any dialog box.

  You can control the angle of curvature on a rounded-corner window (window type `rDocProc`) by adding one of these integers:

Window definition ID	**Diameters of curvature**
`rDocProc`	16, 16
`rDocProc + 2`	4, 4
`rDocProc + 4`	6, 6
`rDocProc + 6`	10, 10

- A specification that determines whether the window is visible or invisible. This characteristic controls only whether the Window Manager displays the window, not necessarily whether the window can be seen on the screen. (A visible window entirely covered by other windows, for example, is "visible" even though the user cannot see it.) You typically create a new window in an invisible state, build the content area of the window, and then display the completed window.

- A specification that determines whether or not the window has a close box. The Window Manager draws the close box when it draws the window frame. The window type specified in the second field determines whether a window can support a close box; this field determines whether the close box is present.
- A reference constant, which your application can use for whatever data it needs to store. When it builds a new window record, the Window Manager stores, in the `refCon` field, whatever value you specify in the fifth element of the window resource. You can also put a placeholder here and then set the `refCon` field yourself with the `SetWRefCon` procedure.
- A string that specifies the window title. The first byte of the string specifies the length of the string (that is, the number of characters in the title plus 1 byte for the length), in bytes.
- An optional positioning specification that overrides the window position established by the rectangle in the first field. The positioning value can be one of the integers defined by the constants listed here. In these constant names, the terms have the following meanings:

`center`	Centered both horizontally and vertically, relative either to a screen or to another window (if a window to be centered relative to another window is wider than the window that preceded it, it is pinned to the left edge; a narrower window is centered)
`stagger`	Located 10 pixels to the right and 10 pixels below the upper-left corner of the last window (in the case of staggering relative to a screen, the first window is placed just below the menu bar at the left edge of the screen, and subsequent windows are placed on that screen relative to the first window)
`alert position`	Centered horizontally and placed in the "alert position" vertically, that is, with about one-fifth of the window or screen above the new window and the rest below
`parent window`	The window in which the user was last working

The seventh element of the resource can contain one of the values specified by these constants:

```
CONST noAutoCenter            = 0x0000;{use initial }
                                       { location}
      centerMainScreen        = 0x280A;{center on main }
                                       { screen}
      alertPositionMainScreen = 0x300A;{place in alert }
                                       { position on main }
                                       { screen}
      staggerMainScreen       = 0x380A;{stagger on main }
                                       { screen}
      centerParentWindow      = 0xA80A;{center on parent }
                                       { window}
```

```
        alertPositionParentWindow  = 0xB00A;{place in alert }
                                            { position on }
                                            { parent window}
        staggerParentWindow        = 0xB80A;{stagger relative }
                                            { to parent window}
        centerParentWindowScreen   = 0x680A;{center on parent }
                                            { window screen}
        alertPositionParentWindowScreen
                                   = 0x700A;{place in alert }
                                            { position on }
                                            { parent window }
                                            { screen}
        staggerParentWindowScreen  = 0x780A;{stagger on parent }
                                            { window screen}
```

The positioning constants are convenient when the user is creating new documents or when you are handling your own dialog boxes and alert boxes. When you are creating a new window to display a previously saved document, however, you should display the new window in the same rectangle as the previous window (that is, the window the document occupied when it was last saved). For more information, see “Positioning a Document Window on the Desktop” beginning on page 4-30.

Use the `GetNewCWindow` or `GetNewWindow` function to read a `'WIND'` resource. Both functions create a new window record and fill it in according to the values specified in a `'WIND'` resource.

The Window Definition Function Resource

Window definition functions are stored as resources of type `'WDEF'`. The `'WDEF'` resource is simply the executable code for the window definition function.

The two standard window definition functions supplied by the Window Manager use resource IDs 0 and 1.

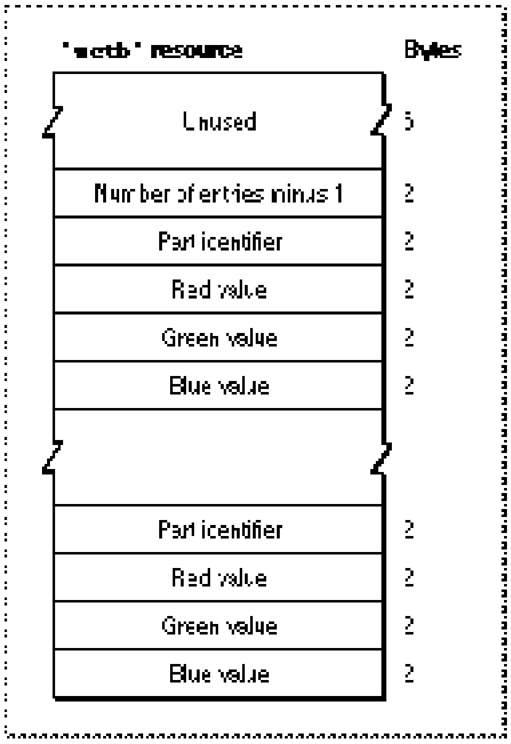
The Window Color Table Resource

You can specify your own window color tables as resources of type `'wctb'`.

Ordinarily, you should not define your own window color tables, unless you have some extraordinary need to control the color of a window’s frame or text highlighting. To assign a table to a window when you create the window, provide a window color table (`'wctb'`) resource with the same resource ID as the `'WIND'` resource from which you create the window.

The window color table resource is an exact image of the window color table data structure. Figure 4-25 illustrates the contents of a compiled `'wctb'` resource.

Figure 4-25 Structure of a compiled window color table (`'wctb'`) resource



A compiled version of a window resource contains the following elements:

- An unused field 6 bytes long.
- An integer that specifies the number of entries in the resource (that is, the number of color specification records) minus 1.
- A series of color specification records, each of which consists of a 2-byte part identifier and three 2-byte color values. The part identifier is an integer specified by one of these constants:

```
CONST wContentColor     = 0;  {content region background}
      wFrameColor       = 1;  {window frame}
      wTextColor        = 2;  {window title and button text}
      wHiliteColor      = 3;  {reserved}
      wTitleBarColor    = 4;  {reserved}
      wHiliteColorLight = 5;  {lightest stripes in title bar }
                              { and lightest dimmed text}
      wHiliteColorDark  = 6;  {darkest stripes in title bar }
                              { and darkest dimmed text}
      wTitleBarLight    = 7;  {lightest parts of title bar }
                              { background}
```

```
wTitleBarDark     = 8;  {darkest parts of title bar }
                        { background}
wDialogLight      = 9;  {lightest element of dialog box }
                        { frame}
wDialogDark       = 10; {darkest element of dialog box }
                        { frame}
wTingeLight       = 11; {lightest window tinging}
wTingeDark        = 12; {darkest window tinging}
```

The color values are simply the intensity of the red, green, and blue in each window part (see *Inside Macintosh: Imaging* for a description of RGB color).

Summary of the Window Manager

Pascal Summary

Constants

```
CONST
   {window types}
   documentProc       = 0;  {movable, sizable window, no zoom box}
   dBoxProc           = 1;  {alert box or modal dialog box}
   plainDBox          = 2;  {plain box}
   altDBoxProc        = 3;  {plain box with shadow}
   noGrowDocProc      = 4;  {movable window, no size box or }
                            { zoom box}
   movableDBoxProc    = 5;  {movable modal dialog box}
   zoomDocProc        = 8;  {standard document window}
   zoomNoGrow         = 12; {zoomable, nonresizable window}
   rDocProc           = 16; {rounded-corner window}

   {window kinds}
   dialogKind         = 2;  {dialog or alert box window}
   userKind           = 8;  {window created by the application}

   {part codes returned by FindWindow}
   inDesk             = 0;  {none of the following}
   inMenuBar          = 1;  {in menu bar}
   inSysWindow        = 2;  {in desk accessory window}
   inContent          = 3;  {anywhere in content region except size }
                            { box if window is active, }
                            { anywhere including size box if window }
                            { is inactive}
   inDrag             = 4;  {in drag (title bar) region}
   inGrow             = 5;  {in size box (active window only)}
   inGoAway           = 6;  {in close box}
   inZoomIn           = 7;  {in zoom box (window in standard state)}
   inZoomOut          = 8;  {in zoom box (window in user state)}

   {axis constraints on DragGrayRgn}
   noConstraint       = 0;  {no constraints}
   hAxisOnly          = 1;  {move on horizontal axis only}
   vAxisOnly          = 2;  {move on vertical axis only}
```

```
{window definition function task codes}
wDraw       = 0;  {draw window frame}
wHit        = 1;  {report where mouse-down occurred}
wCalcRgns   = 2;  {calculate strucRgn and contRgn}
wNew        = 3;  {perform additional initialization}
wDispose    = 4;  {perform additional disposal tasks}
wGrow       = 5;  {draw grow image during resizing}
wDrawGIcon  = 6;  {draw size box and scroll bar outline}

{window definition function wHit return codes}
wNoHit      = 0;  {none of the following}
wInContent  = 1;  {anywhere in content region except size }
                  { box if window is active, }
                  { anywhere including size box if window }
                  { is inactive}
wInDrag     = 2;  {in drag (title bar) region}
wInGrow     = 3;  {in size box (active window only)}
wInGoAway   = 4;  {in close box}
wInZoomIn   = 5;  {in zoom box (window in standard state)}
wInZoomOut  = 6;  {in zoom box (window in user state)}

{window color information table part codes}
wContentColor     = 0;     {content region background}
wFrameColor       = 1;     {window outline}
wTextColor        = 2;     {window title and button text}
wHiliteColor      = 3;     {reserved}
wTitleBarColor    = 4;     {reserved}
wHiliteColorLight = 5;     {lightest stripes in title bar }
                           { and lightest dimmed text}
wHiliteColorDark  = 6;     {darkest stripes in title bar }
                           { and darkest dimmed text}
wTitleBarLight    = 7;     {lightest parts of title bar background}
wTitleBarDark     = 8;     {darkest parts of title bar background}
wDialogLight      = 9;     {lightest element of dialog box frame}
wDialogDark       = 10;    {darkest element of dialog box frame}
wTingeLight       = 11;    {lightest window tinging}
wTingeDark        = 12;    {darkest window tinging}

{resource ID of desktop pattern}
deskPatID         = 16;
```

Data Types

```
TYPE  CWindowPtr  = CGrafPtr;
      CWindowPeek = ^CWindowRecord;

CWindowRecord =
RECORD
   port:          CGrafPort;     {window's graphics port}
   windowKind:    Integer;       {class of window}
   visible:       Boolean;       {visibility}
   hilited:       Boolean;       {highlighting}
   goAwayFlag:    Boolean;       {presence of close box}
   spareFlag:     Boolean;       {presence of zoom box}
   strucRgn:      RgnHandle;     {handle to structure region}
   contRgn:       RgnHandle;     {handle to content region}
   updateRgn:     RgnHandle;     {handle to update region}
   windowDefProc: Handle;        {handle to window definition function}
   dataHandle:    Handle;        {handle to window state data record}
   titleHandle:   StringHandle;  {handle to window title}
   titleWidth:    Integer;       {title width in pixels}
   controlList:   ControlHandle; {handle to control list}
   nextWindow:    CWindowPeek;   {pointer to next window record in }
                                 { window list}
   windowPic:     PicHandle;     {handle to optional picture}
   refCon:        LongInt;       {storage available to your application}
END;

WindowPtr   = GrafPtr;
WindowPeek  = ^WindowRecord;

WindowRecord =
RECORD                           {all fields have same use as }
                                 { in color window record}
   port:          GrafPort;      {window's graphics port}
   windowKind:    Integer;       {class of window}
   visible:       Boolean;       {visibility}
   hilited:       Boolean;       {highlighting}
   goAwayFlag:    Boolean;       {presence of close box}
   spareFlag:     Boolean;       {presence of zoom box}
   strucRgn:      RgnHandle;     {handle to structure region}
   contRgn:       RgnHandle;     {handle to content region}
   updateRgn:     RgnHandle;     {handle to update region}
   windowDefProc: Handle;        {handle to window definition function}
   dataHandle:    Handle;        {handle to window state data record}
   titleHandle:   StringHandle;  {handle to window title}
   titleWidth:    Integer;       {title width in pixels}
   controlList:   ControlHandle; {handle to control list}
   nextWindow:    WindowPeek;    {pointer to next window record in }
                                 { window list}
   windowPic:     PicHandle;     {handle to optional picture}
   refCon:        LongInt;       {storage available to your application}
END;

WStateDataPtr = ^WStateData;
WStateDataHandle = ^WStateDataPtr;

WStateData =            {zoom state data record}
RECORD
   userState:  Rect;    {size and location established by user}
   stdState:   Rect;    {size and location established by application}
END;

WCTabPtr = ^WinCTab;
WCTabHandle = ^WCTabPtr;

WinCTab =                        {window color information table}
RECORD
   wCSeed:     LongInt;          {reserved}
   wCReserved: Integer;          {reserved}
   ctSize:     Integer;          {number of entries in table -1}
   ctTable:    ARRAY [0..4] OF ColorSpec;
                                 {array of color specification records}
END;

ColorSpec   =
RECORD
   value:      Integer;          {part identifier}
   rgb:        RGBColor;         {RGB value}
END;

AuxWinHandle= ^AuxWinPtr;
AuxWinPtr   = ^AuxWinRec;

AuxWinRec   =                    {auxiliary window record}
RECORD
   awNext:        AuxWinHandle;  {handle to next record}
   awOwner:       WindowPtr;     {pointer to window}
   awCTable:      CTabHandle;    {handle to color table}
   dialogCItem:   Handle;        {storage used by Dialog Manager}
   awFlags:       LongInt;       {reserved}
   awReserved:    CTabHandle;    {reserved}
   awRefCon:      LongInt;       {reference constant, for }
                                 { use by application}
END;
```

Window Manager Routines

Initializing the Window Manager

```
PROCEDURE InitWindows;
```

Creating Windows

```
FUNCTION GetNewCWindow      (windowID: Integer; wStorage: Ptr;
                             behind: WindowPtr): WindowPtr;
FUNCTION GetNewWindow       (windowID: Integer; wStorage: Ptr;
                             behind: WindowPtr): WindowPtr;
FUNCTION NewCWindow         (wStorage: Ptr; boundsRect: Rect;
                             title: Str255; visible: Boolean;
                             procID: Integer; behind: WindowPtr;
                             goAwayFlag: Boolean;
                             refCon: LongInt): WindowPtr;
FUNCTION NewWindow          (wStorage: Ptr; boundsRect: Rect;
                             title: Str255; visible: Boolean;
                             theProc: Integer; behind: WindowPtr;
                             goAwayFlag: Boolean;
                             refCon: LongInt): WindowPtr;
```

Naming Windows

```
PROCEDURE SetWTitle         (theWindow: WindowPtr; title: Str255);
PROCEDURE GetWTitle         (theWindow: WindowPtr; VAR title: Str255);
```

Displaying Windows

```
PROCEDURE DrawGrowIcon      (theWindow: WindowPtr);
PROCEDURE SelectWindow      (theWindow: WindowPtr);
PROCEDURE ShowWindow        (theWindow: WindowPtr);
PROCEDURE HideWindow        (theWindow: WindowPtr);
PROCEDURE ShowHide          (theWindow: WindowPtr; showFlag: Boolean);
PROCEDURE HiliteWindow      (theWindow: WindowPtr; fHilite: Boolean);
PROCEDURE BringToFront      (theWindow: WindowPtr);
PROCEDURE SendBehind        (theWindow, behindWindow: WindowPtr);
```

Retrieving Window Information

```
FUNCTION FindWindow          (thePoint: Point;
                              VAR theWindow: WindowPtr): Integer;
FUNCTION FrontWindow         : WindowPtr;
```

Moving Windows

```
PROCEDURE DragWindow         (theWindow: WindowPtr;
                              startPt: Point; boundsRect: Rect);
PROCEDURE MoveWindow         (theWindow: WindowPtr;
                              hGlobal, vGlobal: Integer; front: Boolean);
FUNCTION DragGrayRgn         (theRgn: RgnHandle; startPt: Point;
                              limitRect, slopRect: Rect; axis: Integer;
                              actionProc: ProcPtr): LongInt;
FUNCTION PinRect             (theRect: Rect; thePt: Point): LongInt;
```

Resizing Windows

```
FUNCTION GrowWindow          (theWindow: WindowPtr;
                              startPt: Point; sizeRect: Rect): LongInt;
PROCEDURE SizeWindow         (theWindow: WindowPtr; w, h: Integer;
                              fUpdate: Boolean);
```

Zooming Windows

```
FUNCTION TrackBox            (theWindow: WindowPtr; thePt: Point;
                              partCode: Integer): Boolean;
PROCEDURE ZoomWindow         (theWindow: WindowPtr;
                              partCode: Integer; front: Boolean);
```

Closing and Deallocating Windows

```
FUNCTION TrackGoAway         (theWindow: WindowPtr; thePt: Point): Boolean;
PROCEDURE CloseWindow        (theWindow: WindowPtr);
PROCEDURE DisposeWindow      (theWindow: WindowPtr);
```

Maintaining the Update Region

```
PROCEDURE BeginUpdate        (theWindow: WindowPtr);
PROCEDURE EndUpdate          (theWindow: WindowPtr);
PROCEDURE InvalRect          (badRect: Rect);
PROCEDURE InvalRgn           (badRgn: RgnHandle);
PROCEDURE ValidRect          (goodRect: Rect);
PROCEDURE ValidRgn           (goodRgn: RgnHandle);
```

Setting and Retrieving Other Window Characteristics

```
PROCEDURE SetWindowPic      (theWindow: WindowPtr; Pic: PicHandle);
FUNCTION GetWindowPic       (theWindow: WindowPtr): PicHandle;
PROCEDURE SetWRefCon        (theWindow: WindowPtr; data: LongInt);
FUNCTION GetWRefCon         (theWindow: WindowPtr): LongInt;
FUNCTION GetWVariant        (theWindow: WindowPtr): Integer;
```

Manipulating the Desktop

```
PROCEDURE SetDeskCPat       (deskPixPat: PixPatHandle);
FUNCTION GetGrayRgn         : RgnHandle;
PROCEDURE GetCWMgrPort      (VAR wMgrCPort: CGrafPtr);
PROCEDURE GetWMgrPort       (VAR wPort: GrafPtr);
```

Manipulating Window Color Information

```
PROCEDURE SetWinColor       (theWindow: WindowPtr;
                             newColorTable: WCTabHandle);
FUNCTION GetAuxWin          (theWindow: WindowPtr;
                             VAR awHndl: AuxWinHandle): Boolean;
```

Low-Level Routines

```
FUNCTION CheckUpdate        (VAR theEvent: EventRecord): Boolean;
PROCEDURE ClipAbove         (window: WindowPeek);
PROCEDURE SaveOld           (window: WindowPeek);
PROCEDURE DrawNew           (window: WindowPeek; update: Boolean);
PROCEDURE PaintOne          (window: WindowPeek; clobberedRgn: RgnHandle);
PROCEDURE PaintBehind       (startWindow: WindowPeek;
                             clobberedRgn: RgnHandle);
PROCEDURE CalcVis           (window: WindowPeek);
PROCEDURE CalcVisBehind     (startWindow: WindowPeek;
                             clobberedRgn: RgnHandle);
```

Application-Defined Routine

The Window Definition Function

```
FUNCTION MyWindow           (varCode: Integer; theWindow: WindowPtr;
                             message: Integer; param: LongInt): LongInt;
```

C Summary

Constants

```
enum {
   /*window types*/
   documentProc      = 0,  /*movable, sizable window, no zoom box*/
   dBoxProc          = 1,  /*alert box or modal dialog box*/
   plainDBox         = 2,  /*plain box*/
   altDBoxProc       = 3,  /*plain box with shadow*/
   noGrowDocProc     = 4,  /*movable window, no size box or zoom box*/
   movableDBoxProc   = 5,  /*movable modal dialog box*/
   zoomDocProc       = 8,  /*standard document window*/
   zoomNoGrow        = 9,  /*zoomable, nonresizable window*/
   rDocProc          = 16, /*rounded-corner window*/

   /*window kinds*/
   dialogKind        = 2,  /*dialog or alert box window*/
   userKind          = 8,  /*window created by the application*/

   /*part codes returned by FindWindow*/
   inDesk            = 0,  /*none of the following*/
   inMenuBar         = 1,  /*in menu bar*/
   inSysWindow       = 2,  /*in desk accessory window*/
   inContent         = 3,  /*anywhere in content region except size box if*/
                           /* window is active, anywhere including */
                           /* size box if window is inactive*/
   inDrag            = 4,  /*in drag (title bar) region*/
   inGrow            = 5,  /*in size box (active window only)*/
   inGoAway          = 6,  /*in close box*/
   inZoomIn          = 7,  /*in zoom box (window in standard state)*/
   inZoomOut         = 8   /*in zoom box (window in user state)*/
};

enum {
   /*axis constraints on DragGrayRgn*/
   noConstraint      = 0,  /*no constraints*/
   hAxisOnly         = 1,  /*move on horizontal axis only*/
   vAxisOnly         = 2   /*move on vertical axis only*/
};
```

```
enum {
   /*window definition function task codes*/
   wDraw       = 0,  /*draw window frame*/
   wHit        = 1,  /*report where mouse-down occurred*/
   wCalcRgns   = 2,  /*calculate strucRgn and contRgn*/
   wNew        = 3,  /*perform additional initialization*/
   wDispose    = 4,  /*perform additional disposal tasks*/
   wGrow       = 5,  /*draw grow image during resizing*/
   wDrawGIcon  = 6,  /*draw size box and scroll bar outline*/

   /*window definition function wHit return codes*/
   wNoHit      = 0,  /*none of the following*/
   wInContent  = 1,  /*in content region (except grow, if active)*/
   wInDrag     = 2,  /*in drag region*/
   wInGrow     = 3,  /*in grow region (active window only)*/
   wInGoAway   = 4,  /*in go-away region (active window only)*/
   wInZoomIn   = 5,  /*in zoom box for zooming in (active window */
                     /* only)*/
   wInZoomOut  = 6,  /*in zoom box for zooming out (active window */
                     /* only)*/
   deskPatID   = 16, /*resource ID of desktop pattern*/

   /*window color information table part codes*/
   wContentColor     = 0,     /*the background of the window's */
                              /* content region*/
   wFrameColor       = 1,     /*the window outline*/
   wTextColor        = 2,     /*window title and text in buttons*/
   wHiliteColor      = 3,     /*reserved*/
   wTitleBarColor    = 4,     /*reserved*/
   wHiliteColorLight = 5,     /*lightest stripes in title bar */
                              /* and lightest dimmed text*/
   wHiliteColorDark  = 6,     /*darkest stripes in title bar */
                              /* and darkest dimmed text*/
   wTitleBarLight    = 7,     /*lightest parts of title bar background*/
   wTitleBarDark     = 8,     /*darkest parts of title bar background*/
   wDialogLight      = 9,     /*lightest element of dialog box frame*/
   wDialogDark       = 10,    /*darkest element of dialog box frame*/
   wTingeLight       = 11,    /*lightest window tinging*/
   wTingeDark        = 12     /*darkest window tinging*/
};
```

Data Types

```
struct CWindowRecord {
   CGrafPort            port;          /*window's graphics port*/
   short                windowKind;    /*class of the window*/
   Boolean              visible;       /*visibility*/
   Boolean              hilited;       /*highlighting*/
   Boolean              goAwayFlag;    /*presence of close box*/
   Boolean              spareFlag;     /*presence of zoom box*/
   RgnHandle            strucRgn;      /*handle to structure region*/
   RgnHandle            contRgn;       /*handle to content region*/
   RgnHandle            updateRgn;     /*handle to update region*/
   Handle               windowDefProc; /*handle to window definition */
                                       /* function*/
   Handle               dataHandle;    /*handle to window state data record*/
   StringHandle         titleHandle;   /*handle to window title*/
   short                titleWidth;    /*title width in pixels*/
   ControlHandle        controlList;   /*handle to control list*/
   struct CWindowRecord *nextWindow;   /*next window in window list*/
   PicHandle            windowPic;     /*handle to optional picture*/
   long                 refCon;        /*storage available to your */
                                       /* application*/
};

typedef struct CWindowRecord CWindowRecord;
typedef CWindowRecord *CWindowPeek;

struct WindowRecord {
   GrafPort             port;          /*window's graphics port*/
   short                windowKind;    /*class of the window*/
   Boolean              visible;       /*visibility*/
   Boolean              hilited;       /*highlighting*/
   Boolean              goAwayFlag;    /*presence of close box*/
   Boolean              spareFlag;     /*presence of zoom box*/
   RgnHandle            strucRgn;      /*handle to structure region*/
   RgnHandle            contRgn;       /*handle to content region*/
   RgnHandle            updateRgn;     /*handle to update region*/
   Handle               windowDefProc; /*handle to window definition */
                                       /* function*/
   Handle               dataHandle;    /*handle to window state data record*/
   StringHandle         titleHandle;   /*handle to window title*/
   short                titleWidth;    /*title width in pixels*/
   ControlHandle        controlList;   /*handle to window's control list*/
   struct WindowRecord  *nextWindow;   /*next window in window list*/
   PicHandle            windowPic;     /*handle to optional picture*/
   long                 refCon;        /*reference constant*/
};

typedef struct WindowRecord WindowRecord;
typedef WindowRecord *WindowPeek;

struct WStateData {
   Rect  userState;  /*user state*/
   Rect  stdState;   /*standard state*/
};

typedef struct WStateData WStateData;
typedef WStateData *WStateDataPtr, **WStateDataHandle;

struct AuxWinRec {
    struct AuxWinRec **awNext;      /*handle to next record*/
    WindowPtr        awOwner;       /*pointer to window */
    CTabHandle       awCTable;      /*handle to color table*/
    Handle           dialogCItem;   /*storage used by Dialog Manager*/
    long             awFlags;       /*reserved*/
    CTabHandle       awReserved;    /*reserved*/
    long             awRefCon;      /*reference constant, for use by */
                                    /* application*/
};

typedef struct AuxWinRec AuxWinRec;
typedef AuxWinRec *AuxWinPtr, **AuxWinHandle;

struct WinCTab {
   long        wCSeed;           /*reserved*/
   short       wCReserved;       /*reserved*/
   short       ctSize;           /*number of entries in table –1*/
   ColorSpec   ctTable[5];       /*array of color specification records*/
};

typedef struct WinCTab WinCTab;
typedef WinCTab *WCTabPtr, **WCTabHandle;
```

Window Manager Routines

Initializing the Window Manager

```
pascal void InitWindows(void);
```

Creating Windows

```
pascal WindowPtr GetNewCWindow
                            (short windowID, void *wStorage,
                             WindowPtr behind);
pascal WindowPtr GetNewWindow
                            (short windowID, void *wStorage,
                             WindowPtr behind);
pascal WindowPtr NewCWindow (void *wStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short procID, WindowPtr behind,
                             Boolean goAwayFlag, long refCon);
pascal WindowPtr NewWindow  (void *wStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short theProc, WindowPtr behind,
                             Boolean goAwayFlag, long refCon);
```

Naming Windows

```
pascal void SetWTitle       (WindowPtr theWindow, ConstStr255Param title);
pascal void GetWTitle       (WindowPtr theWindow, Str255 title);
```

Displaying Windows

```
pascal void DrawGrowIcon    (WindowPtr theWindow);
pascal void SelectWindow    (WindowPtr theWindow);
pascal void ShowWindow      (WindowPtr theWindow);
pascal void HideWindow      (WindowPtr theWindow);
pascal void ShowHide        (WindowPtr theWindow, Boolean showFlag);
pascal void HiliteWindow    (WindowPtr theWindow, Boolean fHilite);
pascal void BringToFront    (WindowPtr theWindow);
pascal void SendBehind      (WindowPtr theWindow, WindowPtr behindWindow);
```

Retrieving Mouse Information

```
pascal short FindWindow     (Point thePoint, WindowPtr *theWindow);
pascal WindowPtr FrontWindow(void);
```

Moving Windows

```
pascal void DragWindow      (WindowPtr theWindow, Point startPt,
                             const Rect *boundsRect);
pascal void MoveWindow      (WindowPtr theWindow, short hGlobal,
                             short vGlobal, Boolean front);
```

```
pascal long DragGrayRgn      (RgnHandle theRgn, Point startPt,
                              const Rect *boundsRect,
                              const Rect *slopRect,
                              short axis, DragGrayRgnProcPtr actionProc);
pascal long PinRect          (const Rect *theRect, Point *thePt);
```

Resizing Windows

```
pascal long GrowWindow       (WindowPtr theWindow, Point startPt,
                              const Rect *bBox);
pascal void SizeWindow       (WindowPtr theWindow, short w, short h,
                              Boolean fUpdate);
```

Zooming Windows

```
pascal Boolean TrackBox      (WindowPtr theWindow, Point thePt,
                              short partCode);
pascal void ZoomWindow       (WindowPtr theWindow, short partCode,
                              Boolean front);
```

Closing and Deallocating Windows

```
pascal Boolean TrackGoAway   (WindowPtr theWindow, Point thePt);
pascal void CloseWindow      (WindowPtr theWindow);
pascal void DisposeWindow    (WindowPtr theWindow);
```

Maintaining the Update Region

```
pascal void BeginUpdate      (WindowPtr theWindow);
pascal void EndUpdate        (WindowPtr theWindow);
pascal void InvalRect        (const Rect *badRect);
pascal void InvalRgn         (RgnHandle badRgn);
pascal void ValidRect        (const Rect *goodRect);
pascal void ValidRgn         (RgnHandle goodRgn);
```

Setting and Retrieving Other Window Characteristics

```
pascal void SetWindowPic     (WindowPtr theWindow, PicHandle pic);
pascal PicHandle GetWindowPic
                             (WindowPtr theWindow);
pascal void SetWRefCon       (WindowPtr theWindow, long data);
pascal long GetWRefCon       (WindowPtr theWindow);
pascal short GetWVariant     (WindowPtr theWindow);
```

Manipulating the Desktop

```
pascal void SetDeskCPat      (PixPatHandle deskPixPat);
#define GetGrayRgn()         (* (RgnHandle* 0X09EE))
pascal void GetCWMgrPort     (CGrafPtr *wMgrCPort);
pascal void GetWMgrPort      (GrafPtr *wPort);
```

Manipulating Window Color Information

```
pascal void SetWinColor      (WindowPtr theWindow,
                              WCTabHandle newColorTable);
pascal Boolean GetAuxWin     (WindowPtr theWindow, AuxWinHandle *awHndl);
```

Low-Level Routines

```
pascal Boolean CheckUpdate   (EventRecord *theEvent);
pascal void ClipAbove        (WindowPeek window;)
pascal void SaveOld          (WindowPeek window);
pascal void DrawNew          (WindowPeek window, Boolean update);
pascal void PaintOne         (WindowPeek window, RgnHandle clobberedRgn);
pascal void PaintBehind      (WindowPeek startWindow,
                              RgnHandle clobberedRgn);
pascal void CalcVis          (WindowPeek window);
pascal void CalcVisBehind    (WindowPeek startWindow,
                              RgnHandle clobberedRgn);
```

Application-Defined Routine

The Window Definition Function

```
pascal long MyWindow         (short varCode, WindowPtr theWindow,
                              short message, long param);
```

Assembly-Language Summary

Data Types

Window Record and Color Window Record Data Structure

0	`windowPort`	108 bytes	window's graphics port
108	`windowKind`	word	how window was created
110	`wVisible`	byte	visibility status
111	`wHilited`	byte	highlighted status
112	`wGoAway`	byte	presence of close box
113	`wZoom`	byte	presence of zoom box
114	`structRgn`	long	handle to structure region
118	`contRgn`	long	handle to content region
122	`updateRgn`	long	handle to update region
126	`windowDef`	long	handle to window definition function
130	`wDataHandle`	long	handle to window state data record
134	`wTitleHandle`	long	handle to window's title
138	`wTitleWidth`	word	title width in pixels
140	`wControlList`	long	handle to window's control list
144	`nextWindow`	long	pointer to next window in window list
148	`windowPic`	long	handle to picture for updates
152	`wRefCon`	long	reference constant field

Window State Data Structure

0	`userState`	8 bytes	user state rectangle
8	`stdState`	8 bytes	standard state rectangle

Window Color Information Table Data Structure

0	`ctSeed`	long	ID number for table
4	`ctFlags`	word	flags word
6	`ctSize`	word	number of entries minus 1
8	`ctTable`	variable	a series of color specification records (8 bytes each)

Auxiliary Window Record Data Structure

0	`awNext`	long	handle to next window in chain
4	`awOwner`	long	pointer to associated window record
8	`awCTable`	long	handle to window color information table
12	`dialogCItem`	long	handle to dialog color structures
16	`awFlags`	long	handle for QuickDraw
20	`awResrv`	long	reserved
24	`awRefCon`	long	user constant

Global Variables

`AuxWinHead`	Handle to beginning of auxiliary window list.
`CurActivate`	Pointer to window to receive activate event.
`CurDeactive`	Pointer to window to receive deactivate event.
`DeskHook`	Address of procedure for painting desktop.
`DeskPattern`	Pattern in which desktop is painted (8 bytes).
`DragHook`	Address of optional procedure to execute during `TrackGoAway`, `TrackBox`, `DragWindow`, `GrowWindow`, and `DragGrayRgn`.
`DragPattern`	Pattern of dragged region's outline (8 bytes).
`GrayRgn`	Handle to desktop region.
`OldContent`	Handle to saved content region.
`OldStructure`	Handle to saved structure region.
`PaintWhite`	Flag indicating whether to paint window white before update event (2 bytes).
`SaveUpdate`	Flag indicating whether to generate update events (2 bytes).
`SaveVisRgn`	Handle to saved visible region.
`WindowList`	Pointer to first window in window list.
`WMgrPort`	Pointer to Window Manager port.

Control Manager

Contents

This chapter describes how your application can use the Control Manager to create and manage controls. **Controls** are onscreen objects that the user can manipulate with the mouse. By manipulating controls, the user can take an immediate action or change settings to modify a future action. For example, a scroll bar control allows a user to immediately change the portion of the document that your application displays, whereas a pop-up menu control for baud rate might allow the user to change the rate by which your application handles subsequent data transmissions.

Read this chapter to learn how and when to implement controls. Virtually all applications need to implement controls, at least in the form of scroll bars for document windows. You use Control Manager routines, resources, and data structures to implement scroll bars in your application's document windows.

The other standard Macintosh controls are buttons, checkboxes, radio buttons, and pop-up menus. You can use the Control Manager to create and manage these controls, too. Alternatively, you can use the Dialog Manager to implement these controls in alert boxes and dialog boxes more easily. (You typically use an alert box to warn a user of an unusual situation, and you typically use a dialog box to ask the user for information necessary to carry out a command.) The chapter "Dialog Manager" in this book describes in detail how to implement controls in alert and dialog boxes. However, in certain situations—for instance, when you need to implement highly complex dialog boxes—you may want to use Control Manager routines to manage these types of controls directly; read this chapter for information on how to do so.

For scrolling lists of graphic or textual information (similar to the list of files that system software presents after the user chooses the Open command from the File menu), your application can use the List Manager to implement the scroll bars. See the chapter "List Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information.

The Control Manager offers routines for automatically handling user-generated mouse events in controls and redrawing controls in response to update events. For further information about events and event handling, see the chapter "Event Manager" in this book.

You typically use a control resource—a resource of type `'CNTL'`—to specify the type, size, location, and other attributes of a control. See the chapter "Introduction to the Macintosh Toolbox" in this book for general information about resources; detailed information about the Resource Manager and its routines is provided in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Every control you create must be associated with a particular window. All of the controls for a window are stored in a control list referenced by the window's window record. See the chapter "Window Manager" in this book for general information about windows. (When you use the Dialog Manager to implement a control, the Dialog Manager associates it with its respective dialog box or alert box, as described in the chapter "Dialog Manager.")

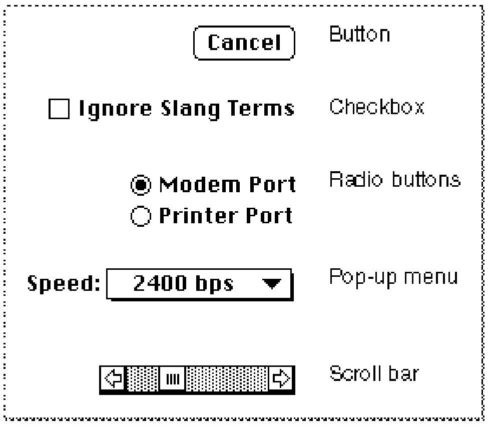
This chapter provides an introduction to the use of controls, and then discusses how you can

- create and display controls
- determine whether mouse-down events have occurred in controls
- respond to mouse-down events in controls
- change the settings in controls
- use scroll bars to move a document in a window
- move and resize controls for a window
- define your own control definition function to create nonstandard controls

Introduction to Controls

The Control Manager provides several standard controls. Figure 5-1 illustrates these standard controls: buttons, checkboxes, radio buttons, pop-up menus, and scroll bars. You can also design and implement your own custom controls.

Figure 5-1 Standard controls provided by the Control Manager



Buttons, checkboxes, and radio buttons are the simplest controls. They consist of only a title and an outline shape, and they respond to only mouse clicks. A pop-up menu is slightly more complex. This control has a menu attached to its title, and it must respond when the user drags the cursor across the menu. A scroll bar, because it consists of different parts that behave differently, is the most complex of the standard controls. Even though a scroll bar has several parts, it is still only one control.

The Control Manager displays these standard controls in colors that provide aesthetic consistency across all monitors, from black-and-white displays to 8-bit color displays. To ensure consistency across applications, you generally shouldn't change the default colors of controls, although the Control Manager does allow you to do so with the `SetControlColor` procedure (described on page 5-101) or the control color table resource (described on page 5-121).

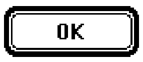
Standard controls and common custom controls are described in the next several sections.

Buttons

Buttons appear on the screen as rounded rectangles with a title centered inside. When the user clicks a button, your application should perform the action described by the button title. Typically, buttons allow the user to perform actions instantaneously—for example, completing the operations defined by a dialog box or acknowledging an error message in an alert box.

Make your buttons large enough to surround their titles. In every window or dialog box in which you display buttons, you should designate one button as the default button by drawing a thick black outline around it (as shown in Figure 5-2). Your application should respond to key-down events involving the Enter and Return keys as if the user had clicked the default button. (In your alert boxes, the Dialog Manager automatically outlines the default button; you must outline the default button in your dialog boxes.)

Figure 5-2 A default button

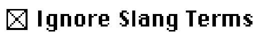


You normally use buttons in alert boxes and dialog boxes. See the chapter “Dialog Manager” for additional details about where to display buttons, what to title them, how to respond to events involving them, and how to draw an outline around them.

Checkboxes

Checkboxes provide alternative choices. Typically you use checkboxes in dialog boxes so that users can specify information necessary for completing a command. Checkboxes act like toggle switches, turning a setting either off or on. Use checkboxes to indicate one or more options that must be either off or on. A checkbox appears as a small square with a title alongside it; use the Control Manager procedure `SetControlValue` to place an X in the box when the user selects it by clicking it on and to remove the X when the user deselects it by clicking it off. Figure 5-3 shows a selected checkbox.

Figure 5-3 A selected checkbox



When you design a dialog box, you can include any number of checkboxes—including only one. Checkboxes are independent of each other, even when they offer related options. Within a dialog box, it's a good idea to group sets of related checkboxes and to provide some visual demarcation between different groups.

Each checkbox has a title. It can be very difficult to title the option in an unambiguous way. The title should reflect two clearly opposite states. For example, in a Finder's Info window, a checkbox provides the option to lock a file. The checkbox is titled simply Locked. The clearly opposite state, when the option is off, is unlocked.

If you can't devise a checkbox title that clearly implies an opposite state, you might be better off using two radio buttons. With two radio buttons, you can use two titles, thereby clarifying the states.

Checkboxes are frequently used in dialog boxes to set or modify future actions instead of specifying actions to be taken immediately. See the chapter "Dialog Manager" in this book for a detailed discussion of how and where to display checkboxes in dialog boxes.

Radio Buttons

Like checkboxes, **radio buttons** retain and display an on-or-off setting. You organize radio buttons in a group to offer a choice among several alternatives—typically, inside a dialog box. Radio buttons are small circles; when the user clicks a radio button to turn it on, use the Control Manager procedure `SetControlValue` to fill the radio button with a small black dot. The user can have only one radio button setting in effect at one time. In other words, radio buttons are mutually exclusive. However, the Control Manager cannot determine how your radio buttons are grouped; therefore, when the user turns on one radio button, it is up to your application to use `SetControlValue` to turn off the others in that group.

A set of radio buttons normally has two to seven items; each set must always have at least two radio buttons. Each set of radio buttons must have a label that identifies the kind of choices the group offers. Also, each button must have a title that identifies what the radio button does. This title can be a few words or a phrase. A set of radio buttons is *never* dynamic—that is, its contents should never change according to the context. (If you need to display more than seven items, or if the items change as the context changes, you should use a pop-up menu instead.)

Radio buttons represent choices that are related but not necessarily opposite. For example, a pair of radio buttons may provide a choice between using the modem port or the printer port, as shown in Figure 5-1 on page 5-4. If more than one set of radio buttons is visible at one time, you need to demarcate the sets from one another. For example, you can draw a dotted line around a set of radio buttons to separate it from other elements in a dialog box.

Pop-Up Menus

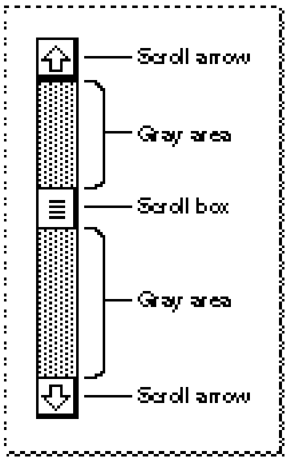
Pop-up menus, introduced in the chapter "Menu Manager" in this book, provide the user with a simple way to choose from among a list of choices without having to move the cursor to the menu bar. As an alternative to a group of radio buttons, a pop-up menu is particularly useful for specifying a group of settings or values that number five or more, or whose settings or values might change. Like the items in a set of radio buttons, the items in a pop-up menu are mutually exclusive—that is, only one choice from the menu can be in effect at any time. Figure 5-8 on page 5-12 illustrates the choices available in a pop-up menu that has been selected by the user.

Never use a pop-up menu as a way to provide the user with commands. Pop-up menus should not list actions (that is, verbs); instead, they should list attributes (that is, adjectives) or settings from which the user can choose one option.

Scroll Bars

Scroll bars change what portion of a document the user can view within the document's window. A scroll bar is a light gray rectangle with scroll arrows at each end. Inside the scroll bar is a square called the **scroll box.** The rest of the scroll bar is called the **gray area.** Windows can have a horizontal scroll bar, a vertical scroll bar, or both. A vertical scroll bar lies along the right side of a window. A horizontal scroll bar runs along the bottom of a window. Figure 5-4 shows the parts of a scroll bar.

Figure 5-4 A vertical scroll bar



If the user drags the scroll box, clicks a scroll arrow, or clicks anywhere in the gray area, your application "moves" the document accordingly; use Control Manager routines as appropriate to move the scroll box. Figure 5-5 illustrates, and the next few sections explain, several key behaviors of a scroll bar.

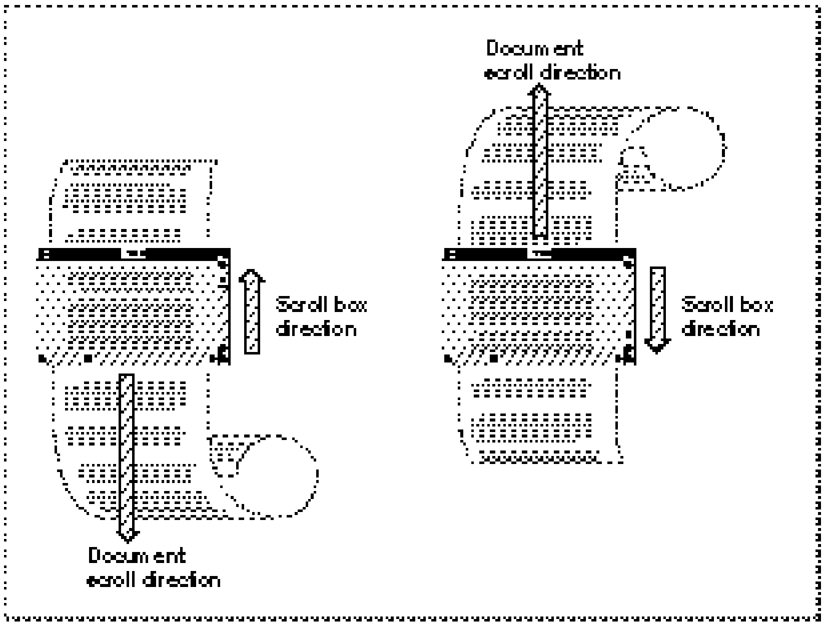
A scroll bar represents the entire document in one dimension, top to bottom or right to left. The scroll box shows the position, relative to the whole document, of the visible portion of the document. If the scroll box is halfway between the top and bottom of the scroll bar, then what the user sees should be about halfway through the document. Use the `SetControlValue` or `SetControlMaximum` procedure to move the scroll box whenever your application resizes a window and whenever it scrolls through a document for any reason other than responding to the user dragging the scroll box.

After the user drags the scroll box, the Control Manager redraws the scroll box in its new position. You then use the `GetControlValue` function to determine the position of the scroll box, and you display the appropriate portion of the document. By dragging the scroll box, the user can move quickly through the document. For example, to see the beginning of the document, the user drags the scroll box to the top of the scroll bar. Your application then scrolls to the top of the document.

At either end of the scroll bar are **scroll arrows** that indicate the direction of movement through the document. For instance, when the user clicks the top scroll arrow, your application needs to move toward the beginning of the document. Thus, the document moves down, seemingly in the opposite direction. By clicking the scroll arrow, the user tells your application, “Show me more of the document that’s hidden in this direction.”

Your application uses the `SetControlValue` procedure to move the scroll box in the direction of the arrow being clicked. In this way, the scroll box continues to represent the approximate position of the visible part of the document in relation to the whole document. For example, when the user clicks the top scroll arrow, you move the document down to bring more of the top of the document into view, and you move the scroll box up, as illustrated in Figure 5-5.

Figure 5-5 Using the scroll box and scroll arrows



Each click of a scroll arrow should move the document a distance of one unit in the chosen direction. Your application determines what one unit equals. For example, a word processor should move one line of text for each click in the arrow. A spreadsheet should move one row or one column, depending on the direction of the arrow. To ensure smooth scrolling effects, it's usually best to specify the same size units within a document. When the user holds down the mouse button while the cursor is in a scroll arrow, your application should continuously scroll through the document in the indicated direction until the user releases the mouse button or your application has scrolled as far as possible.

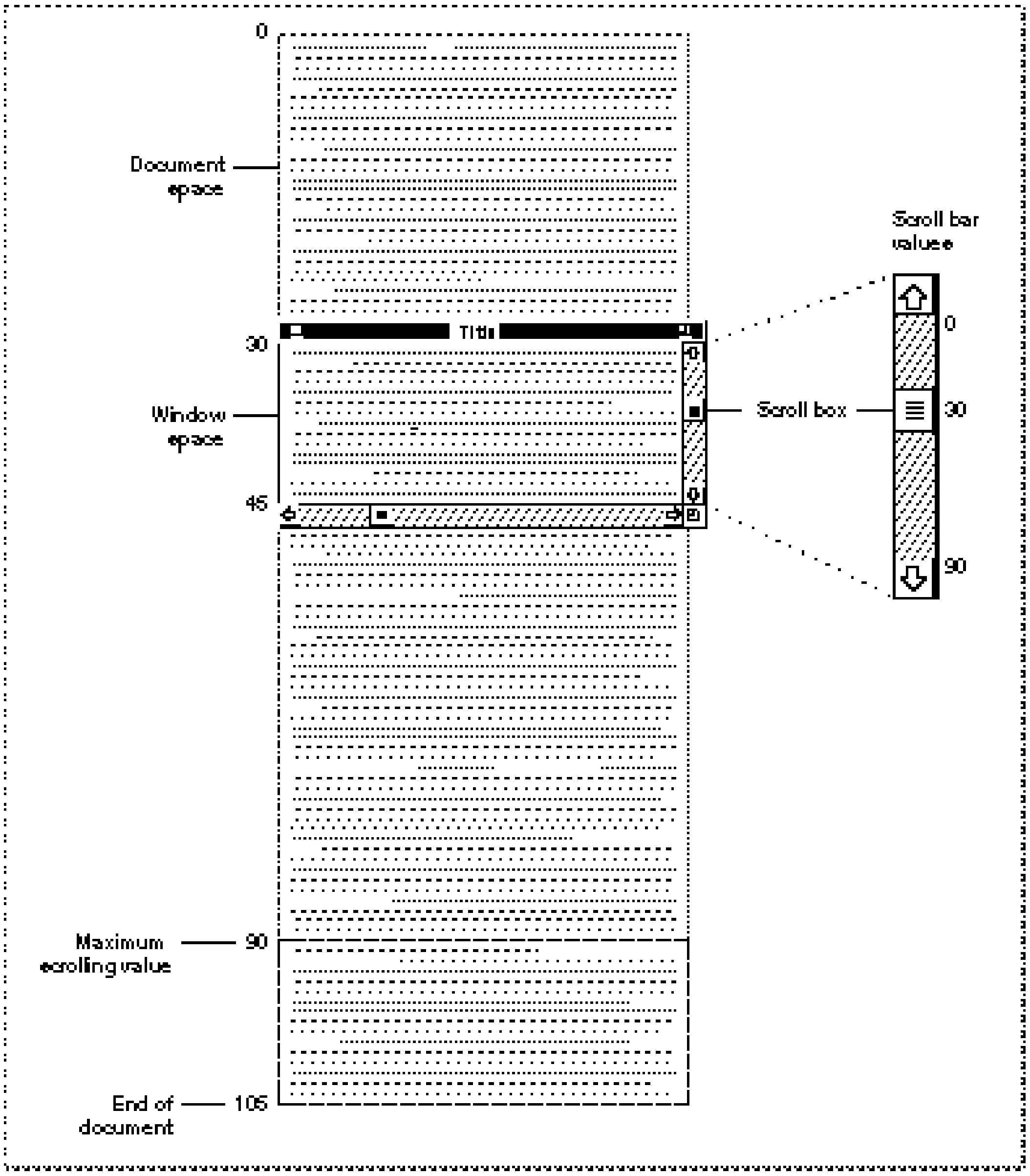
The rest of the area within the scroll bar—excluding the scroll box and the scroll arrows—is called the gray area. When the user clicks the gray area of a scroll bar, your application should move the displayed area of the document by an entire window of information minus one scroll unit. For example, if the window displays 15 lines of text and the user clicks the gray area below the scroll box, your application should move the document up 14 lines so that the bottom line of the previous view appears at the top of the new view. (This retained line helps the user see the newly displayed material in context.) You must also move the scroll box an appropriate distance in that direction. For example, when the user clicks the gray area below the scroll box, move the document view by one window toward the bottom of the document and use `SetControlValue` to move the scroll box accordingly.

When your application scrolls through a document—for example, when the user manipulates a scroll bar—your application must move the document's coordinate space in relation to the window's coordinate space. Your application uses the scroll box to indicate the location of the top of the displayed portion of the document relative to the rest of the document.

For example, if a text window contains 15 lines of text and the user scrolls 30 lines from the top of the document, the scroll box should be set to a value of 30. The window displays all of the lines between line 30 and line 45, as shown in Figure 5-6 on the next page. The scroll box always indicates the displacement between the beginning of the document and the top of the displayed portion of the document.

To prevent the user from scrolling past the edge of the document and seeing a blank window, you should—for a vertical scroll bar—allow the document to scroll no farther than the length of the document minus the height of the window, excluding the 15-pixel-deep region for the horizontal scroll bar at the bottom edge of the window. Likewise, for a horizontal scroll bar, you should allow the document to scroll no farther than the width of the document minus the width of the window—here, too, excluding the 15-pixel-wide region for the vertical scroll bar at the right edge of the window.

Figure 5-6 Spatial relations between a document and a window, and their representation by a scroll bar



For example, the document shown in Figure 5-6 is 105 lines long. So that the last 15 lines will fill the window when the user scrolls to the end of the document, the application does not scroll beyond 90 lines. Because the user has scrolled to line 30 of a maximum 90 lines, the scroll box appears a third of the way down the scroll bar.

“Scrolling Through a Document” beginning on page 5-43 describes in detail how to scroll through a document in a window.

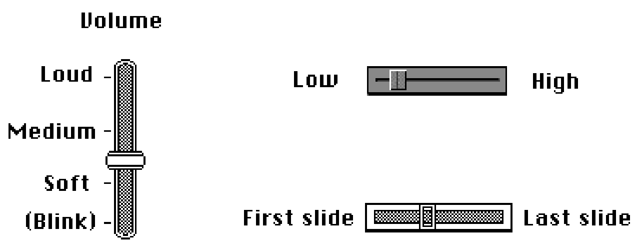
Other Controls

If you need controls other than the standard ones provided by the Control Manager, you can design and implement your own. Typically, the only types of controls you might need to implement are sliders or dials. **Sliders** and **dials** (which differ only in appearance) are similar to scroll bars in that they graphically represent a range of values that a user can set. Use an **indicator**—such as a sliding switch or a dial needle—to indicate the current setting for the control and to let the user set its value. (For scroll bars, the scroll box is the indicator.)

If you want to display a value not under the user's direct control (for example, the amount of free space remaining on a disk), you should use a status bar or other type of graphic instead of a slider or dial.

Figure 5-7 illustrates several custom controls, which are used for purposes such as setting the speaker volume, the gray-scale saturation level, and the relative position of a slide within a presentation. As in this figure, be sure to include meaningful labels that indicate the range and the direction of your control's indicator.

Figure 5-7 Custom slider controls



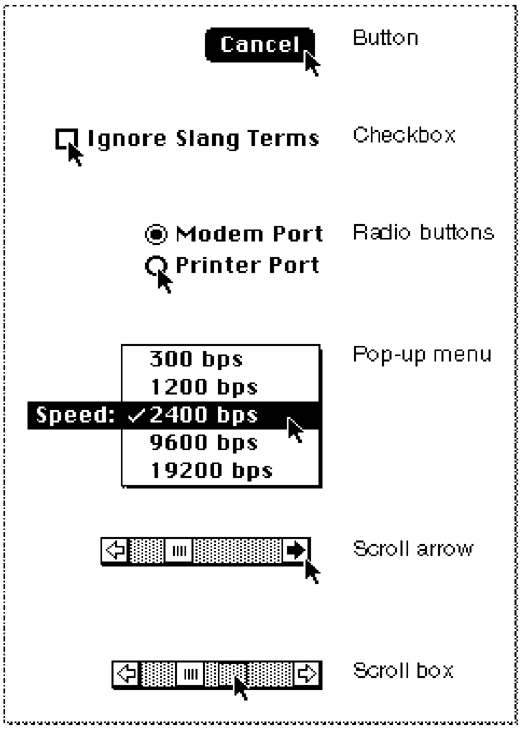
A scroll bar is a slider representing the entire contents of a window, and the user uses the scroll box to move to a specific location in that content. Don't use scroll bars to represent any other concept (for instance, changing a setting). Otherwise, your departure from the consistent Macintosh interface might confuse the user.

Active and Inactive Controls

You can make a control become either active or inactive. Figure 5-8 on the next page shows how the `TrackControl` function (which you use in response to a mouse-down event in a control) gives visual feedback when the user moves the cursor to an **active control** and presses the mouse button. In particular, `TrackControl` responds to mouse-down events in active controls by

- displaying buttons in inverse video
- drawing checkboxes and radio buttons with heavier lines
- highlighting the titles of and displaying the items in pop-up menus
- highlighting scroll arrows
- moving outlines of scroll boxes when users drag them

Figure 5-8 Visual feedback for user selection of active controls



Your application, in turn, should respond appropriately to mouse events involving active controls. Most often, your application waits until the user releases the mouse button before taking any action; as long as the user holds down the mouse button when the cursor is over a control, you typically let `TrackControl` react to the mouse-down event; `TrackControl` then informs your application the moment the user releases the mouse button when the cursor is over an active control.

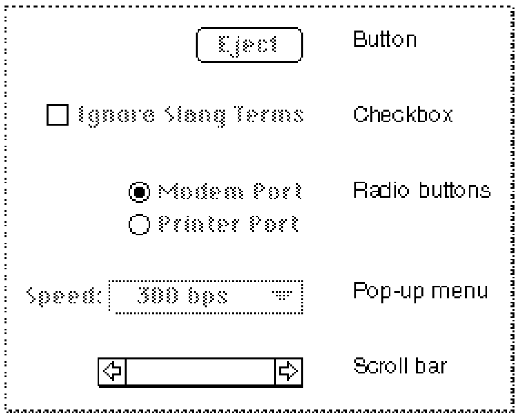
As soon as the user releases the mouse button, your application should

- perform the task identified by the button title when the cursor is over an active button
- toggle the value of the checkbox when the cursor is over an active checkbox (The Control Manager then draws or removes the checkmark, as appropriate.)
- turn on the radio button and turn off all other radio buttons in the group when the cursor is over an active radio button
- use the new setting chosen by the user when the cursor is over an active pop-up menu
- show more of the document in the direction of the scroll arrow when the cursor is over the scroll arrow or gray area of an active scroll bar, and move the scroll box accordingly
- determine where the user has dragged the scroll box when the cursor is over the scroll box and then display the corresponding portion of the document

Sometimes your application should respond even before the user releases the mouse button—that is, your application should undertake some continuous action as long as the user holds down the mouse button when the cursor is in an active control. Most typically, when the user moves the cursor to a scroll arrow or gray area and then holds down the mouse button, your application should continuously scroll through the document until the user releases the mouse button or until the user can't scroll any farther. To perform this kind of action, you define an **action procedure** and specify it to `TrackControl`; `TrackControl` calls your action procedure as long as the user holds down the mouse button.

Whenever it is inappropriate for your application to a respond to a mouse-down event in a control, you should make it inactive. An **inactive control** is one that the user can't use because it has no meaning or effect in the current context—for example, the scroll bars in an empty window. The Control Manager continues to display an inactive control so that it remains visible, but in a manner that indicates its state to the user. As shown in Figure 5-9, the Control Manager dims inactive buttons, checkboxes, radio buttons, and pop-up menus, and it lightens the gray area and removes the scroll box from inactive scroll bars.

Figure 5-9 Inactive controls



You can use the `HiliteControl` procedure to make any control inactive and then active again. Except for scroll bars (which you should hide using the `HideControl` procedure), you should use `HiliteControl` to make all other controls inactive when their windows are not frontmost. You typically use controls other than scroll bars in dialog boxes. See the chapter "Dialog Manager" in this book for a discussion of how to make buttons, radio buttons, checkboxes, and pop-up menus inactive and active.

You make scroll bars inactive when the document is smaller than the window in which you display it. To make a scroll bar inactive, you typically use the `SetControlMaximum` procedure to make the scroll bar's maximum value equal to its minimum value, in which case the Control Manager automatically makes the scroll bar inactive. To make it active again, you typically use `SetControlMaximum` to make its maximum value larger than its minimum value.

The Control Definition Function

A **control definition function** determines how a control generally looks and behaves. Various Control Manager routines call a control definition function whenever they need to perform some control-dependent action, such as drawing the control on the screen.

Control definition functions are stored as resources of type `'CDEF'`. The System file includes three **standard control definition functions,** stored with resource IDs of 0, 1, and 63. The `'CDEF'` resource with resource ID 0 defines the look and behavior of buttons, checkboxes, and radio buttons; the `'CDEF'` resource with resource ID 1 defines the look and behavior of scroll bars; and the `'CDEF'` resource with resource ID 63 defines the look and behavior of pop-up menus. (If you want to define nonstandard controls, you'll have to write control definition functions for them, as described in "Defining Your Own Control Definition Function" beginning on page 5-109.)

Just as a window definition function can describe variations of the same basic window, a control definition function can use a **variation code** to describe variations of the same basic control. You specify a particular control with a control definition ID. The **control definition ID** is an integer that contains the resource ID of the control definition function in its upper 12 bits and a variation code in its lower 4 bits. For a given resource ID and variation code, the control definition ID is derived as follows:

control definition ID = 16 * (`'CDEF'` resource ID) + variation code

For example, buttons, checkboxes, and radio buttons all use the standard control definition function with resource ID 0; because they have variation codes of 0, 1, and 2, respectively, their respective control definition IDs are 0, 1, and 2.

You can use these constants to define the controls provided by the standard control definition functions:

Constant	Control definition ID	Control
`pushButProc`	0	Button
`checkBoxProc`	1	Checkbox
`radioButProc`	2	Radio button
`scrollBarProc`	16	Scroll bar
`popupMenuProc`	1008	Pop-up menu

The control definition function for scroll bars figures out whether a scroll bar is vertical or horizontal from a rectangle you specify when you create the control.

About the Control Manager

You can use the Control Manager to

- create and dispose of controls
- display, update, and hide controls

- change the size, location, and appearance of controls
- monitor and respond to the user's operation of a control
- determine and change the settings and other attributes of a control

Your application performs these actions by calling the appropriate Control Manager routines. The Control Manager carries out the actual operations, but it's up to you to decide when, where, and how to carry these out.

Using the Control Manager

To implement a control, you generally

- use a control resource (that is, a resource of type `'CNTL'`) to describe the control
- create and display the control
- determine when the user presses, clicks, or holds down the mouse button while the cursor is in the control
- respond as appropriate to events involving the control—for example, by displaying a different portion of the document when the user manipulates a scroll bar
- respond as appropriate to other events in windows that include controls—for example, by moving and resizing a scroll bar when the user resizes a window, or by hiding one window's scroll bars when the user makes a different window active

These tasks are explained in greater detail in the rest of this chapter.

Before using the Control Manager, you must initialize QuickDraw, the Font Manager, and the Window Manager, in that order, by using the `InitGraf`, `InitFonts`, and `InitWindows` procedures. (See *Inside Macintosh: Imaging* for information about `InitGraf` and `InitFonts`; see the chapter "Window Manager" in this book for information about `InitWindows`.)

Creating and Displaying a Control

To create a control in one of your application's windows, use the `GetNewControl` or `NewControl` function. You should usually use `GetNewControl`, which takes information about the control from a control resource (that is, a `'CNTL'` resource) in a resource file. Like window resources, control resources isolate descriptive information from your application code for ease of modification—especially for translation to other languages. The rest of this section describes how to use `GetNewControl`. Although it's generally not recommended, you can also use the `NewControl` function and pass it the necessary descriptive information in individual parameters instead of using a control resource. The `NewControl` function is described on page 5-82.

When you use `GetNewControl`, you pass it the resource ID of the control resource, and you pass it a pointer to a window. The function then creates a data structure (called a **control record**) of type `ControlRecord` from the information in the control resource, adds the control record to the control list for your window, and returns as its function result a handle to the control. (You use a control's handle when referring to the control in most other Control Manager routines; when you create scroll bars or pop-up menus for a window, you should store their handles in one of your application's own data structures for later reference.)

When you specify in the control resource that a control is initially visible and you use the `GetNewControl` function, the Control Manager uses the control's control definition function to draw the control inside its window. The Control Manager draws the control immediately, without using your window's standard updating mechanism. If you specify that a control is invisible, you can use the `ShowControl` procedure when you want to draw the control. Again, the Control Manager draws the control without using your window's standard updating mechanism. (Of course, even when the Control Manager draws the control, it might be completely or partially obscured from the user by overlapping windows or other objects.)

When your application receives an update event for a window that contains controls, you use the `UpdateControls` procedure in your application's standard window-updating code to redraw all the controls in the update region of the window.

Note

When you use the Dialog Manager to implement buttons, radio buttons, checkboxes, or pop-up menus in alert boxes and dialog boxes, Dialog Manager routines automatically use Control Manager routines to create and update these controls for you. If you implement any controls other than buttons, radio buttons, checkboxes, and pop-up menus in alert or dialog boxes—and whenever you implement *any* controls (scroll bars, for example) in your application's windows—you must explicitly use either the `GetNewControl` or the `NewControl` function to create the controls. You must always use the `UpdateControls` procedure to update controls you put in your own windows. u

When you use the Window Manager procedure `DisposeWindow` or `CloseWindow` to remove a window, either procedure automatically removes all controls associated with the window and releases the memory they occupy.

When you no longer need a control in a window that you want to keep, you can use the `DisposeControl` procedure, described on page 5-108, to remove it from the screen, delete it from its window's control list, and release the control record and all other associated data structures from memory. You can use the `KillControls` procedure, described on page 5-108, to dispose of all of a window's controls at once.

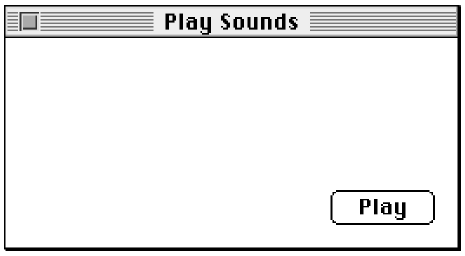
The next section, "Creating a Button, Checkbox, or Radio Button," provides a general discussion of the control resource as well as a more detailed description of the use of the control resource to specify buttons, checkboxes, and radio buttons in your application's windows. The two following sections, "Creating Scroll Bars" (beginning on page 5-21) and "Creating a Pop-Up Menu" (beginning on page 5-25), describe those elements of the control resource that differ from the control resources for buttons, checkboxes, and radio buttons. "Updating a Control" beginning on page 5-29 then offers an example of how you can use the `UpdateControls` procedure within your window-updating code.

Note
For the Control Manager to draw a control properly inside a window, the window must have its upper-left corner at local coordinates (0,0). If you use the QuickDraw procedure `SetOrigin` to change a window's local coordinate system, be sure to change it back—so that the upper-left corner is again at (0,0)—before drawing any of its controls. Because many Control Manager routines can (at least potentially) redraw a control, the safest policy after changing a window's local coordinate system is to change the coordinate system back before calling any Control Manager routine. u

Creating a Button, Checkbox, or Radio Button

Figure 5-10 shows a simple example of a button placed in a window of type `noGrowDocProc`—which you normally use to create a modeless dialog box. Although you usually use the Dialog Manager to create dialog boxes and their buttons, sometimes you might use the Window Manager and the Control Manager instead. The chapter "Dialog Manager" in this book explains why the use of the Window and Control Managers is sometimes preferable for this purpose.

Figure 5-10 A button in a simple window



Listing 5-1 shows an application-defined routine, `MyCreatePlaySoundsWindow`, that uses the `GetNewControl` function to create the button shown in Figure 5-10.

Listing 5-1 Creating a button for a window

```
FUNCTION MyCreatePlaySoundsWindow: OSErr;
VAR
   myWindow: WindowPtr;
BEGIN
   MyCreatePlaySoundsWindow := noErr;
   myWindow := GetNewWindow(rPlaySoundsModelessWindow, NIL, POINTER(-1));
   IF myWindow <> NIL THEN
   BEGIN
      {use the window's refCon to identify this window}
      SetWRefCon(myWindow, LongInt(kMyPlaySoundsWindow));
      SetPort(myWindow);
      gMyPlayButtonCtlHandle := GetNewControl(rPlayButton, myWindow);
      IF (gMyPlayButtonCtlHandle = NIL) THEN
         MyCreatePlaySoundsWindow := kControlErr;
   END
   ELSE
      MyCreatePlaySoundsWindow := kNoSoundWindow;
END;
```

The `MyCreatePlaySoundsWindow` routine begins by using the Window Manager function `GetNewWindow` to create a window; a pointer to that window is passed to `GetNewControl`. Note that, as explained in the chapter “Dialog Manager” in this book, you could create a modeless dialog box more easily by using the Dialog Manager function `GetNewDialog` and specifying its controls in an item list (`'DITL'`) resource.

For the resource ID of a control resource, the `MyCreatePlaySoundsWindow` routine defines an `rPlayButton` constant, which it passes to the `GetNewControl` function. Listing 5-2 shows how this control resource appears in Rez input format.

Listing 5-2 Rez input for a control resource

```
resource 'CNTL' (rPlayButton, preload, purgeable) {
   {87, 187, 107, 247},    /*rectangle*/
   0,                      /*initial setting*/
   visible,                /*make control visible*/
   1,                      /*maximum setting*/
   0,                      /*minimum setting*/
   pushButProc,            /*control definition ID*/
   0,                      /*reference value*/
   "Play"                  /*title*/
};
```

You supply the following information in the control resource for a button, checkbox, radio button, or scroll bar:

- a rectangle, specified by coordinates local to the window, that determines the control’s size and location
- the initial setting for the control
- a constant (either `visible` or `invisible`) that specifies whether the control should be drawn on the screen immediately
- the maximum setting for the control
- the minimum setting for the control
- the control definition ID
- a reference value, which your application may use for any purpose
- the title of the control; or, for scroll bars, an empty string

As explained in “Creating a Pop-Up Menu” beginning on page 5-25, the values you supply in a control resource for a pop-up menu differ from those you specify for other buttons, checkboxes, radio buttons, and scroll bars.

Buttons are drawn to fit the rectangle exactly. To allow for the tallest characters in the system font, there should be at least a 20-point difference between the top and bottom coordinates of the rectangle. Listing 5-2 uses a rectangle with coordinates (87,187,107,247) to describe the size and location of the control within the window. Remember that the Control Manager will not draw controls properly unless the upper-left corner of the window coincides with the coordinates (0,0).

In Listing 5-2, the initial and minimum settings for the button are 0 and the maximum setting is 1. In control resources for buttons, checkboxes, and radio buttons, supply these values as the initial settings:

- For buttons, which don’t retain a setting, specify a value of 0 for the initial and minimum settings and 1 for the maximum setting.
- For checkboxes and radio buttons, which retain an on-or-off setting, specify a value of 0 when you want to the control to be initially off. To turn a checkbox or radio button on, assign it an initial setting of 1. In response, the Control Manager places an X in a checkbox or a black dot in a radio button.

Because the `visible` identifier is specified in this example, the control is drawn immediately in its window. If you use the `invisible` identifier, your control is not drawn until your application uses the `ShowControl` procedure. When you want to make a visible control invisible, you can use the `HideControl` procedure.

In Listing 5-2, the maximum setting for the button is 1, which you, too, should specify in your control resources as the maximum setting for buttons, checkboxes, and radio buttons. In Listing 5-2, the minimum setting for the button is 0, which you, too, should specify in your control resources as the minimum setting for buttons, checkboxes, and radio buttons.

In Listing 5-2, the `pushButProc` constant is used to specify the control definition ID. Use the `checkBoxProc` constant to specify a checkbox and the `radioButProc` constant to specify a radio button.

Listing 5-2 specifies a reference value of 0. Your application can use this value for any purpose (except when you add the `popupUseAddResMenu` variation code to the `popupMenuProc` control definition function, as described in “Creating a Pop-Up Menu” beginning on page 5-25).

Finally, the string `"Play"` is specified as the title of the control. Buttons, checkboxes, and radio buttons require a title that communicates their purpose to the user. (The chapter “Dialog Manager” in this book offers extensive guidelines on appropriate titles for buttons.)

When specifying a title, make sure it fits in the control’s rectangle; otherwise, the Control Manager truncates the title. For example, it truncates the titles of checkboxes and radio buttons on the right in Roman scripts, and it centers and truncates both ends of button titles.

If you localize your application for use with worldwide versions of system software, the titles may become longer or shorter. Translated text is often 50 percent longer than U.S. English text. You may need to resize your controls to accommodate the translated text.

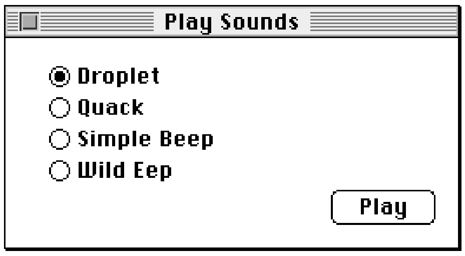
By default, the Control Manager displays control titles in the system font. To make it easier to localize your application for use with worldwide versions of system software, you should not change the font. Do not use a smaller font, such as 9-point Geneva; some script systems, such as KanjiTalk, require 12-point fonts. You can spare yourself future localization effort by leaving all control titles in the system font.

Follow book-title style when you capitalize control titles. In general, capitalize one-word titles and capitalize nouns, adjectives, verbs, and prepositions of four or more letters in multiple-word titles. You usually don't capitalize words such as *in, an,* or *and.* For capitalization rules, see the *Apple Publications Style Guide,* available from APDA.

The Control Manager allows button, checkbox, and radio button titles of multiple lines. When specifying a multiple-line title, end each line with the ASCII character code $0D (carriage return). If the control is a button, each line is horizontally centered, and the font leading is inserted between lines. (The height of each line is equal to the distance from the ascent line to the descent line plus the leading of the font used. Be sure to make the total height of the rectangle greater than the number of lines times this height.) If the control is a checkbox or a radio button, the text is justified as appropriate for the user's current script system, and the checkbox or button is vertically centered within its rectangle.

Figure 5-11 shows the Play Sounds window with four additional controls: radio buttons titled Droplet, Quack, Simple Beep, and Wild Eep.

Figure 5-11 Radio buttons in a simple window



Only one of these radio buttons can be on at a time. Listing 5-3 initially sets the Droplet radio button to 1, turning it on by default. This listing also shows the control resources for the other buttons, all initially set to 0 to turn them off.

For a checkbox or a radio button, always allow at least a 16-point difference between the top and bottom coordinates of its rectangle to accommodate the tallest characters in the system font.

Listing 5-3 Rez input for the control resources of radio buttons

```
resource 'CNTL' (cDroplet, preload, purgeable) {
   {13, 23, 31, 142},/*rectangle of control*/
   1,                /*initial setting*/
   visible,          /*make control visible*/
   1,                /*maximum setting*/
   0,                /*minimum setting*/
   radioButProc,     /*control definition ID*/
   0,                /*reference value*/
   "Droplet"         /*control title*/
};
resource 'CNTL' (cQuack, preload, purgeable) {
   {31, 23, 49, 142},/*rectangle of control*/
   0,                /*initial setting*/
   visible, 1, 0, radioButProc, 0, "Quack"};

resource 'CNTL' (cSimpleBeep, preload, purgeable) {
   {49, 23, 67, 142},/*rectangle of control*/
   0,                /*initial setting*/
   visible, 1, 0, radioButProc, 0, "Simple Beep"};

resource 'CNTL' (cWildEep, preload, purgeable) {
   {67, 23, 85, 142},/*rectangle of control*/
   0,                /*initial setting*/
   visible, 1, 0, radioButProc, 0, "Wild Eep"};
```

Creating Scroll Bars

When you define the control resource for a scroll bar, specify the `scrollBarProc` constant for the control definition ID. Typically, you make the scroll bar invisible and specify an initial value of 0, a minimum value of 0, and a maximum value of 0, and you supply an empty string for the title.

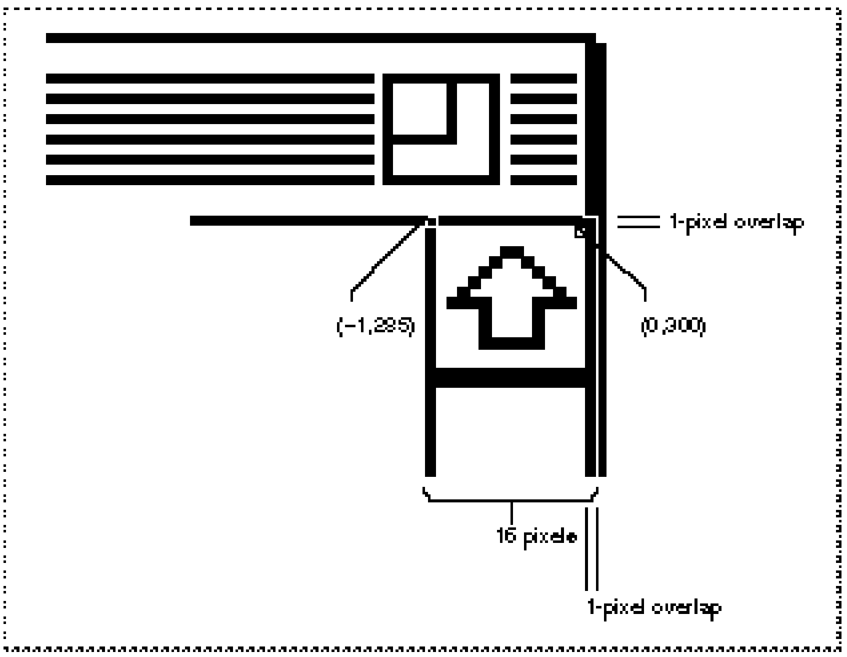
After you create a window, use the `GetNewControl` function to create the scroll bar you've defined in the control resource and to attach that scroll bar to the window. Use the `MoveControl`, `SizeControl`, `SetControlMaximum`, and `SetControlValue` procedures to adjust the location, size, and settings of the scroll bars, and then use the `ShowControl` procedure to display the scroll bars.

In your window-handling code, make the maximum setting the maximum area you want to allow the user to scroll. Most applications allow the user to drag the size box and click the zoom box to change the size of windows, and they allow the user to add information to and remove it from documents. To allow users to perform these actions, your application needs to calculate a changing maximum setting based upon the document's current size and its window's current size. For new documents that have no content to scroll to, assign an initial value of 0 as the maximum setting in the control resource; the control definition function automatically makes a scroll bar inactive when its minimum and maximum settings are identical. Thereafter, your window-handling routines should set and maintain the maximum setting, as described in "Determining and Changing Control Settings" beginning on page 5-37.

By convention, a scroll bar is 16 pixels wide, so there should be a 16-point difference between the left and right coordinates of a vertical scroll bar's rectangle and between the top and bottom coordinates of a horizontal scroll bar's rectangle. (If you don't provide a 16-pixel width, the Control Manager scales the scroll bar to fit the width you specify.) A standard scroll bar should be at least 48 pixels long, to allow room for the scroll arrows and scroll box.

The Control Manager draws lines that are 1 pixel wide for the rectangle enclosing the scroll bar. As shown in Figure 5-12, the outside lines of a scroll bar should overlap the lines that the Window Manager draws for the window frame.

Figure 5-12 How a scroll bar should overlap the window frame



To determine the rectangle for a *vertical* scroll bar, perform the following calculations and use their results in your control resource. (Do not include the area of the title bar in your calculations.)

- top coordinate = combined height of any items above the scroll bar – 1
- left coordinate = width of window – 15
- bottom coordinate = height of window – 14
- right coordinate = width of window + 1

To determine the rectangle for a *horizontal* scroll bar, perform the following calculations and use their results in your control resource.

- top coordinate = height of window – 15
- left coordinate = combined width of any items to the left of the scroll bar – 1
- bottom coordinate = height of window + 1
- right coordinate = width of window – 14

The top coordinate of a vertical scroll bar is –1, and the left coordinate of a horizontal scroll bar is –1, unless your application uses part of the window's typical scroll bar areas (in particular, those areas opposite the size box) for displaying information or specifying additional controls. For example, your application may choose to display the current page number of a document in the lower-left corner of the window—that is, in a small area to the left of its window's horizontal scroll bar. See *Macintosh Human Interface Guidelines* for a discussion of appropriate uses of a window's scroll bar areas for additional items and controls.

Just as the maximum settings of a window's scroll bars change when the user resizes the document's window, so too do the scroll bars' coordinate locations change when the user resizes the window. Although you must specify an initial maximum setting and location in the control resource for a scroll bar, your application must be able to change them dynamically—typically, by storing handles to each scroll bar in a document record when you create a window, and then by using Control Manager routines to change control settings (as described in "Determining and Changing Control Settings" beginning on page 5-37) and sizes and locations of controls (as described in "Moving and Resizing Scroll Bars" beginning on page 5-65).

Listing 5-4 shows a window resource (described in the chapter "Window Manager" in this book) for creating a window, and two control resources for creating the window's vertical and horizontal scroll bars. The rectangle for the initial size and shape of the window is specified in global coordinates, of course, and the rectangles for the two scroll bars are specified in coordinates local to the window.

Listing 5-4 Rez input for resources for a window and its scroll bars

```
                           /*initial window*/
resource 'WIND' (rDocWindow, preload, purgeable) {
   {64, 60, 314, 460},     /*initial rectangle for window*/
   zoomDocProc, invisible, goAway, 0x0, "untitled"
};
                           /*initial vertical scroll bar*/
resource 'CNTL' (rVScroll, preload, purgeable) {
   {-1, 385, 236, 401},    /*initial rectangle for control*/
      /*initial setting, visibility, max, min, ID, refcon, title*/
   0, invisible, 0, 0, scrollBarProc, 0, ""
};
```

```
                           /*initial horizontal scroll bar*/
resource 'CNTL' (rHScroll, preload, purgeable) {
   {235, -1, 251, 386},    /*initial rectangle for control*/
      /*initial setting, visibility, max, min, ID, refcon, title*/
   0, invisible, 0, 0, scrollBarProc, 0, ""
};
```

Listing 5-5 shows an application-defined procedure called `DoNew` that uses the `GetNewWindow` and `GetNewControl` functions to create a window and its scroll bars from the resources in Listing 5-4.

Listing 5-5 Creating a document window with scroll bars

```
PROCEDURE DoNew (newDocument: Boolean; VAR window: WindowPtr);
VAR
   good:                Boolean;
   windStorage:         Ptr;
   myData:              MyDocRecHnd;
BEGIN
   {use GetNewWindow or GetNewCWindow to create the window here}
   myData := MyDocRecHnd(NewHandle(SIZEOF(MyDocRec))); {create document rec}
   {test for errors along the way; if there are none, create the scroll }
   { bars and save their handles in myData}
   IF good THEN
   BEGIN    {create the vertical scroll bar and save its handle}
      myData^^.vScrollBar := GetNewControl(rVScroll, window);
      {create the horizontal scroll bar and save its handle}
      myData^^.hScrollBar := GetNewControl(rHScroll, window);
      good := (vScrollBar <> NIL) AND (hScrollBar <> NIL);
   END;
   IF good THEN
   BEGIN    {adjust size, location, settings, and visibility of scroll bars}
      MyAdjustScrollBars(window, FALSE);
      {perform other initialization here}
      IF NOT newDocument THEN
         ShowWindow(window);
   END;
   {clean up here}
END; {DoNew}
```

The `DoNew` routine uses Window Manager routines to create a window; its window resource specifies that the window is invisible. The window resource specifies an initial size and location for the window, but because the window is invisible, this window is not drawn.

Then `DoNew` creates a document record and stores a handle to it in the `myData` variable. The SurfWriter sample application uses this document record to store the data that the user creates in this window—as well as handles to the scroll bars that it creates. The SurfWriter sample application later uses these control handles to handle scrolling through the document and to move and resize the scroll bars when the user resizes the window. (See the chapter "Window Manager" in this book for more information about creating such a document record.)

To create scroll bars, `DoNew` uses `GetNewControl` twice—once for the vertical scroll bar and once for the horizontal scroll bar. The `GetNewControl` function returns a control handle; `DoNew` stores these handles in the `vScrollBar` and `hScrollBar` fields of its document record for later reference.

Because the window and the scroll bars are invisible, nothing is drawn onscreen yet for the user. Before drawing the window and its scroll bars, `DoNew` calls another application-defined procedure, `MyAdjustScrollBars`. In turn, `MyAdjustScrollBars` calls other application-defined routines that move and resize the scroll bars to fit the window and then calculate the maximum settings of these controls. (Listing 5-14 on page 5-39 shows the `MyAdjustScrollBars` procedure.)

After creating the window and its scroll bars, and then sizing and positioning them appropriately, `DoNew` uses the Window Manager procedure `ShowWindow` to display the window with its scroll bars.

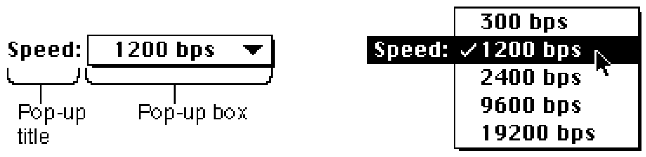
Creating a Pop-Up Menu

The values you specify in a control resource for a pop-up menu differ from those you specify for other controls. The control resource for a pop-up menu contains the following information:

- a rectangle, specified by coordinates local to the window, that determines the size and location of the pop-up title and pop-up box
- the alignment of the pop-up title with the pop-up box
- a constant (either `visible` or `invisible`) that specifies whether the control should be drawn on the screen immediately
- the width of the pop-up title
- the resource ID of the `'MENU'` resource describing the pop-up menu items
- the control definition ID
- a reference value, which your application may use for any purpose
- the title of the control

Figure 5-13 on the next page shows a pop-up menu; Listing 5-6 shows the control resource that creates this pop-up menu. (The chapter "Menu Manager" in this book recommends typical uses of pop-up menus and describes the relation between pop-up menus and menus you display in the menu bar.)

Figure 5-13 A pop-up menu

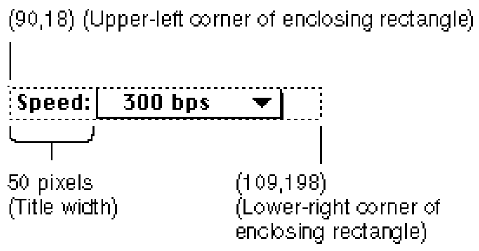


Listing 5-6 Rez input for the control resource of a pop-up menu

```
resource 'CNTL' (kPopUpCNTL, preload, purgeable) {
   {90, 18, 109, 198},  /*rectangle of control*/
   popupTitleLeftJust,  /*title position*/
   visible,             /*make control visible*/
   50,                  /*pixel width of title*/
   kPopUpMenu,          /*'MENU' resource ID*/
   popupMenuCDEFProc,   /*control definition ID*/
   0,                   /*reference value*/
   "Speed:"             /*control title*/
};
```

Listing 5-6 specifies a rectangle with the coordinates (90,18,109,198). Figure 5-14 illustrates the rectangle for this pop-up menu.

Figure 5-14 Dimensions of a sample pop-up menu


(90,18) (Upper-left corner of enclosing rectangle)
Speed:
300 bps
50 pixels (Title width)
(109,198) (Lower-right corner of enclosing rectangle)

Listing 5-6 uses the `popupTitleLeftJust` constant to specify the position of the control title. Specify any combination of the following constants (or their values) to inform the Control Manager where and how to draw the pop-up menu's title:

Setting	Constant	Description
$0000	`popupTitleLeftJust`	Place title left of the pop-up box
$0001	`popupTitleCenterJust`	Center title over the pop-up box
$00FF	`popupTitleRightJust`	Place title right of the pop-up box
$0100	`popupTitleBold`	Use boldface font style
$0200	`popupTitleItalic`	Use italic font style

Setting	Constant	Description
$0400	`popupTitleUnderline`	Use underline font style
$0800	`popupTitleOutline`	Use outline font style
$1000	`popupTitleShadow`	Use shadow font style
$2000	`popupTitleCondense`	Use condensed characters
$4000	`popupTitleExtend`	Use extended characters
$8000	`popupTitleNoStyle`	Use monostyle font

If `GetNewControl` completes successfully, it sets the value of the `contrlValue` field of the control record by assigning to that field the item number of the first menu item. When the user chooses a different menu item, the Control Manager changes the `contrlValue` field to that item number.

When you create pop-up menus, your application should store the handles for them; for example, in a record pointed to by the `refCon` field of a window record or a dialog record. (See the chapters "Window Manager" and "Dialog Manager" in this book for more information about the window record and the dialog record.) Storing these handles, as shown in the following code fragment, allows your application to respond later to users' choices in pop-up menus:

```
myData: MyDocRecHnd;
window: WindowPtr;

myData^^.popUpControlHandle := GetNewControl(kPopUpCNTL, window);
```

Listing 5-6 specifies 50 pixels (in place of a maximum setting) as the width of the control title. After it creates the control, the Control Manager sets the maximum value in the pop-up menu's control record to the number of items in the pop-up menu. Figure 5-14 illustrates this title width for the pop-up menu.

Listing 5-6 uses a `kPopUpMenu` constant to specify the resource ID of a `'MENU'` resource (in place of a minimum setting for the control). (See the chapter "Menu Manager" in this book for a description of the `'MENU'` resource type.) After it creates the control, the Control Manager assigns 1 as the minimum setting in the pop-up menu's control record.

IMPORTANT

When using the ResEdit application, version 2.1.1, you must use the same resource ID when specifying the menu resource and the control resource that together define a pop-up menu. s

You can also specify a different control definition ID by adding any or all of the following constants (or the variation codes they represent) to the `popupMenuProc` constant:

```
CONST popupFixedWidth      = $0001; {use fixed-width control}
      popupUseAddResMenu   = $0004; {use resource for menu items}
      popupUseWFont        = $0008; {use window font}
```

Constant	Description
`popUpFixedWidth`	Uses a constant control width. If your application specifies this value, the pop-up control definition function does not resize the control horizontally to fit long menu items. The width of the pop-up box is set to the width of the control, minus the width of the pop-up title your application specifies when it creates the control. If a menu item in a pop-up box does not fit in the space provided, the text is truncated to fit, and three ellipsis points (...) are appended at the end. If you do not specify this variation code, the pop-up control definition function may resize the control horizontally.
`popupUseAddResMenu`	Gets menu items from a resource other than the `'MENU'` resource. If your application specifies this value when creating a pop-up menu, the control definition function interprets the value in the `contrlRfCon` field of the control record as a value of type `ResType`. The control definition function uses the Menu Manager procedure `AppendResMenu` to add resources of that type to the menu.
`popupUseWFont`	Uses the font of the specified window. If your application specifies this value, the pop-up control definition function draws the pop-up menu title using the font and size of the window containing the control instead of using the system font.

The reference value that you specify in the control resource (and stored by the Control Manager in the `contrlRfCon` field of the control record) is available for your application's use. However, if you specify `popupUseAddResMenu` as a variation code, the Control Manager coerces the value in the `contrlRfCon` field of the control record to the type `ResType` and then uses `AppendResMenu` to add items of that type to the pop-up menu. For example, if you specify a reference value of `LongInt('FONT')` as the reference value, the control definition function appends a list of the fonts installed in the system to the menu associated with the pop-up menu. After the control has been created, your application can use the control record's `contrlRfCon` field for whatever use it requires. You can determine which menu item is currently chosen by calling `GetControlValue`.

Whenever the pop-up menu is redrawn, its control definition function calls the Menu Manager procedure `CalcMenuSize`. This procedure recalculates the size of the menu associated with the control (to allow for the addition or deletion of items in the menu). The pop-up control definition function may also update the width of the pop-up menu to the sum of the width of the pop-up title, the width of the longest item in the menu, the width of the downward-pointing arrow, and a small amount of white space. As previously described, your application can override this behavior by adding the variation code `popupFixedWidth` to the pop-up control definition ID.

You should not use the Menu Manager function `GetMenuHandle` to obtain a handle to a menu associated with a pop-up control. If necessary, you can obtain the menu handle (and the menu ID) of a pop-up menu by dereferencing the `contrlData` field of the pop-up menu's control record. The `contrlData` field of a control record is a handle to a block of private information. For pop-up menu controls, this field is a handle to a pop-up private data record, which is described on page 5-77.

Updating a Control

Your program should use the `UpdateControls` procedure upon receiving an update event for a window that contains controls such as scroll bars. (Window Manager routines such as `SelectWindow`, `ShowWindow`, and `BringToFront` do not automatically call `UpdateControls` to display the window's controls. Instead, they merely add the appropriate regions to the window's update region. This in turn generates an update event.)

Note
The Dialog Manager automatically updates the controls you use in alert boxes and dialog boxes. u

When your application receives an update event for a window that contains controls, use the `UpdateControls` procedure in your window-updating code to redraw all the controls in the update region of the window. Call `UpdateControls` after using the Window Manager procedure `BeginUpdate` and before using the Window Manager procedure `EndUpdate`.

When you call `UpdateControls`, you pass it parameters specifying the window to be updated and the window area that needs updating. Use the visible region of the window's graphics port, as referenced in the port's `visRgn` field, to specify the window's update region.

Listing 5-7 shows an application-defined routine, `DoUpdate`, that responds to an update event. The `DoUpdate` routine calls the Window Manager procedure `BeginUpdate`. To redraw this portion of the window, `DoUpdate` then calls another of its own procedures, `MyDrawWindow`.

Listing 5-7 Responding to an update event for a window

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:
      BEGIN
         BeginUpdate(window);
         MyDrawWindow(window);
         EndUpdate(window);
      END;  {of updating document windows}
   {handle other window types——modeless dialogs, etc.——here}
   END;  {of windowType CASE}
END;  {of DoUpdate}
```

Listing 5-8 illustrates how the SurfWriter sample application updates window controls and other window contents by using its own application-defined routine, `MyDrawWindow`. To draw only those controls in the window's update region, `MyDrawWindow` calls `UpdateControls`. To draw the size box in the lower-right corner of the window, `MyDrawWindow` calls the Window Manager procedure `DrawGrowIcon`. Finally, `MyDrawWindow` redraws the appropriate information contained in the user's document. Because the SurfWriter application uses TextEdit for all text editing in the window contents, Listing 5-8 calls the TextEdit procedure `TEUpdate`. (TextEdit is described in detail in *Inside Macintosh: Text*.)

Listing 5-8 Redrawing the controls in the update region

```
PROCEDURE MyDrawWindow (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN                {draw the contents of the window}
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
   BEGIN
      EraseRect(portRect);
      UpdateControls(window, visRgn);
      DrawGrowIcon(window);
      TEUpdate(portRect, myData^^.editRec);  {redraw text}
   END;
   HUnLock(Handle(myData));
END;  {MyDrawWindow}
```

For more information about updating window contents, see the chapter "Window Manager" in this book.

Responding to Mouse Events in a Control

The Control Manager provides several routines to help you detect and respond to mouse events involving controls. For mouse events in controls, you generally perform the following tasks:

1. In your event-handling code, use the Window Manager function `FindWindow` to determine the window in which the mouse-down event occurred.
2. If the mouse-down event occurred in the content region of your application's active window, use the `FindControl` function to determine whether the mouse-down event occurred in an active control and, if so, which control.
3. Call `TrackControl` to handle user interaction for the control for as long as the user holds the mouse button down. For scroll arrows and the gray areas of scroll bars, you

must define an action procedure for `TrackControl` to use. This action procedure should cause the document to scroll as long as the user holds down the mouse button. For pop-up menus, you pass `Pointer(-1)` in a parameter to `TrackControl` to use the action procedure defined in the pop-up control definition function. For the scroll box in scroll bars and for the other standard controls, you pass `NIL` in a parameter to `TrackControl` to get the Control Manager's standard response to mouse-down events.

4. When `TrackControl` reports that the user has released the mouse button with the cursor in a control, respond appropriately. This may require you to use other Control Manager routines, such as `GetControlValue` and `SetControlValue`, to determine and change control settings.

These and other routines for responding to events involving controls are described in the next several sections.

Note

The Dialog Manager procedure `ModalDialog` automatically calls `FindWindow`, `FindControl`, and `TrackControl` for mouse-down events in the controls of alert and modal dialog boxes. You can use the Dialog Manager function `DialogSelect`, which automatically calls `FindWindow`, `FindControl`, and `TrackControl`, to help you handle mouse events in your movable modal and modeless dialog boxes. u

Determining a Mouse-Down Event in a Control

When your application receives a mouse-down event, use the Window Manager function `FindWindow` to determine the window in which the event occurred. If the cursor was in the content region of your application's active window when the user pressed the mouse button, use the `FindControl` function to determine whether the mouse-down event occurred in an active control and, if so, which control.

When the mouse-down event occurs in a visible, active control, `FindControl` returns a handle to that control as well as a part code identifying the control's part. (Note that when the mouse-down event occurs in an invisible or inactive control, or when the cursor is not in a control, `FindControl` sets the control handle to `NIL` and returns 0 as its part code.)

A simple control such as a button or checkbox might have just one "part"; a more complex control can have as many parts as are needed to define how the control operates. A scroll bar has five parts: two scroll arrows, the scroll box, and the two gray areas on either side of the scroll box. Figure 5-4 on page 5-7 shows the five parts of a scroll bar.

A **part code** is an integer from 1 through 253 that identifies a part of a control. To allow different parts of a multipart control to respond to mouse events in different ways, many of the Control Manager routines accept a part code as a parameter or return one as a result. Part codes are assigned to a control by its control definition function. The standard control definition functions define the following part codes. Also listed are the constants you can use to represent them.

Constant	Part code	Control part
`inButton`	10	Button
`inCheckBox`	11	Entire checkbox or radio button
`inUpButton`	20	Up scroll arrow for a vertical scroll bar, left scroll arrow for a horizontal scroll bar
`inDownButton`	21	Down scroll arrow for a vertical scroll bar, right scroll arrow for a horizontal scroll bar
`inPageUp`	22	Gray area above scroll box for a vertical scroll bar, gray area to left of scroll box for a horizontal scroll bar
`inPageDown`	23	Gray area below scroll box for a vertical scroll bar, gray area to right of scroll box for a horizontal scroll bar
`inThumb`	129	Scroll box

The pop-up control definition function does not define part codes for pop-up menus. Instead (as explained in “Creating a Pop-Up Menu” beginning on page 5-25), your application should store the handles for your pop-up menus when you create them. Your application should then test the handles you store against the handles returned by `FindControl` before responding to users’ choices in pop-up menus; this is described in more detail later in the next section.

Listing 5-9 illustrates an application-defined procedure, `DoMouseDown`, that an application might call in response to a mouse-down event. The `DoMouseDown` routine first calls the Window Manager function `FindWindow`, which returns two values: a pointer to the window in which the mouse-down event occurred and a constant that provides additional information about the location of that event. If `FindWindow` returns the `inContent` constant, then the mouse-down event occurred in the content area of one of the application’s windows.

Listing 5-9 Detecting mouse-down events in a window

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:       Integer;
   thisWindow: WindowPtr;
BEGIN       {handle mouse-down event}
   part := FindWindow(event.where, thisWindow);
   CASE part OF
      inMenuBar:
         ;  {mouse-down in menu bar, respond appropriately here}
      inContent:
         IF thisWindow <> FrontWindow THEN
            {mouse-down in an inactive window; use SelectWindow }
            { to make it active here}
         ELSE        {mouse-down in the active window}
            DoContentClick(thisWindow, event);
      {handle other cases here}
   END; {of CASE statement}
END;  {DoMouseDown}
```

In Listing 5-9, when `FindWindow` reports a mouse-down event in the content region of a window containing controls, `DoMouseDown` calls another application-defined procedure, `DoContentClick`, and passes it the window pointer returned by the `FindWindow` function as well as the event record.

Listing 5-10 shows an application-defined procedure, `DoContentClick`, that uses this information to determine whether the mouse-down event occurred in a control.

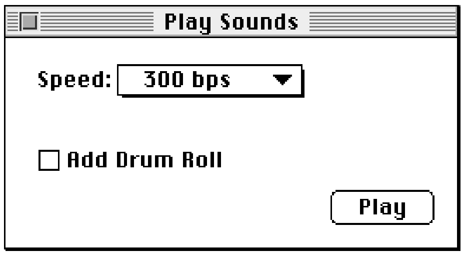
Listing 5-10 Detecting mouse-down events in a pop-up menu and a button

```
PROCEDURE DoContentClick (window: WindowPtr; event: EventRecord);
VAR
   mouse:      Point;
   control:    ControlHandle;
   part:       Integer;
   windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);    {get window type}

   CASE windowType OF

   kPlaySoundsModelessDialogBox:
      BEGIN
         SetPort(window);
         mouse := event.where;   {get the mouse location}
         GlobalToLocal(mouse);   {convert to local coordinates}
         part := FindControl(mouse, window, control);
         IF control = gSpeedPopUpControlHandle THEN
            {mouse-down in Modem Speed pop-up menu}
            DoPopUpMenu(mouse, control);
         CASE part OF
            inButton:   {mouse-down in Play button}
               DoPlayButton(mouse, control);
            inCheckBox: {mouse-down in checkbox}
               DoDrumRollCheckBox(mouse, control);
            OTHERWISE
            ;
         END;  {of CASE for control part codes}
      END;  {of kPlaySoundsModelessDialogBox case}
   {handle other window types, such as document windows, here}
   END; {of CASE for window types}
END; {of DoContentClick}
```

Figure 5-15 shows the Play Sounds window; `DoContentClick` uses the `FindControl` function to determine whether the mouse-down event occurred in the pop-up menu, the Play button, or the Add Drum Roll checkbox.

First, however, `DoContentClick` uses the event record to determine the cursor location, which is specified in global coordinates. Because the `FindControl` function expects the cursor location in coordinates local to the window, `DoContentClick` uses the QuickDraw procedure `GlobalToLocal` to convert the point stored in the `where` field of the event record to coordinates local to the current window. The `GlobalToLocal` procedure takes one parameter, a point in global coordinates—where the upper-left corner of the entire bit image is coordinate (0,0). See *Inside Macintosh: Imaging* for more information about the `GlobalToLocal` procedure.

Figure 5-15 Three controls in a window



When it calls `FindControl`, `DoContentClick` passes the cursor location in the window's local coordinates as well as the pointer returned earlier by the `FindWindow` function (shown in Listing 5-9 on page 5-32).

If the cursor is in a control, `FindControl` returns a handle to the control and a part code indicating the control part. Because the pop-up control definition function does not define control parts, `DoContentClick` tests the control handle returned by `FindControl` against a pop-up menu's control handle that the application stores in its own global variable. If these are handles to the same control, `DoContentClick` calls another application-defined routine, `DoPopUpMenu`.

After checking whether `FindControl` returns a control handle to a pop-up menu, `DoContentClick` uses the part code that `FindControl` returns to determine whether the cursor is in one of the other two controls. If `FindControl` returns the `inButton` constant, `DoContentClick` calls another application-defined routine, `DoPlayButton`. If `FindControl` returns the `inCheckBox` constant, `DoContentClick` calls another application-defined routine, `DoDrumRollCheckBox`.

As described in the next section, all three of these application-defined routines—`DoPopUpMenu`, `DoPlayButton`, and `DoDrumRollCheckBox`—in turn use the `TrackControl` function to follow and respond to the user's mouse movements in the control reported by `FindControl`.

Tracking the Cursor in a Control

After using the `FindControl` function to determine that the user pressed the mouse button when the cursor was in a control, use the `TrackControl` function first to follow and respond to the user's mouse movements, and then to determine which control part contains the cursor when the user releases the mouse button.

Generally, you use `TrackControl` after using the `FindControl` function to determine that the mouse-down event occurred in a control. You pass to `TrackControl` the control handle returned by the `FindControl` function, and you also pass to `TrackControl` the same point you passed to `FindControl` (that is, a point in coordinates local to the window).

The `TrackControl` function follows the movements of the cursor in a control and provides visual feedback until the user releases the mouse button. The visual feedback given by `TrackControl` depends on the control part in which the mouse-down event occurred. When highlighting the control is appropriate—in a button, for example—`TrackControl` highlights the control part (and removes the highlighting when the user releases the mouse button). When the user presses the mouse button while the cursor is in an indicator (such as the scroll box of a scroll bar) and then moves the mouse, `TrackControl` responds by dragging a dotted outline of the indicator. Figure 5-8 on page 5-12 illustrates how `TrackControl` provides visual feedback.

You can also use an action procedure to undertake additional actions as long as the user holds down the mouse button. For example, if the user is working in a text document and holds down the mouse button while the cursor is in a scroll arrow, your action procedure should continuously scroll through the document one line (or some equivalent measure) at a time until the user releases the button or reaches the end of the document. You pass a pointer to this procedure to `TrackControl`. ("Scrolling in Response to Events in Scroll Arrows and Gray Areas" beginning on page 5-57 describes how to do this.)

The `TrackControl` function returns the control's part code if the user releases the mouse button while the cursor is inside the control part, or 0 if the user releases the mouse button while the cursor is outside the control part. Unless `TrackControl` returns 0 as its function result, your application should then respond as appropriate to a mouse-up event in that control part. When `TrackControl` returns 0 as its function result, your application should do nothing.

Listing 5-11 on the next page shows an application-defined procedure, `DoPlayButton`, that uses `TrackControl` to track mouse-down events in the Play button shown in Figure 5-15. The `DoPlayButton` routine passes, to `TrackControl`, the control handle returned by `FindControl`. The `DoPlayButton` routine also passes to `TrackControl` the same cursor location it passed to `FindControl` (that is, a point in local coordinates). Because buttons don't need an action procedure, `NIL` is passed as the final parameter to `TrackControl`.

Listing 5-11 Using the `TrackControl` function with a button

```
PROCEDURE DoPlayButton (mouse: Point; control: ControlHandle);
BEGIN
   IF TrackControl(control, mouse, NIL) <> 0 THEN  {user clicks Play}
   BEGIN
      IF gPlayDrumRoll = TRUE THEN  {user clicked Play Drum Roll checkbox }
         DoPlayDrumRoll;            { so play a drum roll first}
      SysBeep(30);   {always play system alert sound when user clicks Play}
   END;
END;
```

When the user presses the mouse button when the cursor is in the Play button, `TrackControl` inverts the Play button. If the user releases the mouse button after moving the cursor outside the control part, `TrackControl` stops inverting the button and returns the value 0, in which case `DoPlayButton` does nothing.

If, however, the user releases the mouse button with the cursor in the Play button, `TrackControl` stops inverting the Play button and returns the value for the `inButton` constant. Then `DoPlayButton` calls the Sound Manager procedure `SysBeep` to play the system alert sound (which is described in the chapter "Dialog Manager" in this book). Before releasing the mouse button, the user can move the cursor away from the control part and then return to it, and `TrackControl` will still return the part code when the user releases the mouse button.

For buttons, checkboxes, radio buttons, and the scroll box in a scroll bar, your application typically passes `NIL` to `TrackControl` to use no action procedure. However, `TrackControl` still responds visually to mouse events in active controls. That is, when the user presses the mouse button with the cursor over a control whose action procedure is set to `NIL`, `TrackControl` changes the control's display appropriately until the user releases the mouse button.

For scroll arrows and for the gray areas of a scroll box, you need to define your own action procedures. You pass a pointer to the action procedure as one of the parameters to `TrackControl`, as described in "Scrolling in Response to Events in Scroll Arrows and Gray Areas" beginning on page 5-57.

For a pop-up menu, you must pass `Pointer(-1)` to `TrackControl` for its action procedure; this causes `TrackControl` to use the action procedure defined in the pop-up control definition function.

Listing 5-10 on page 5-33 calls an application-defined routine, `DoPopUpMenu`, when `FindControl` reports a mouse-down event in a pop-up menu. Listing 5-12 shows how `DoPopUpMenu` uses `TrackControl` to handle user interaction in the pop-up menu. By passing `Pointer(-1)` to `TrackControl`, `DoPopUpMenu` uses the action procedure defined in the pop-up control definition function.

Listing 5-12 Using `TrackControl` with a pop-up menu

```
PROCEDURE DoPopUpMenu (mouse: Point; control: ControlHandle);
VAR
   menuItem:   Integer;
   part:       Integer;
BEGIN
   part := TrackControl(control, mouse, Pointer(-1));
   menuItem := GetControlValue(control);
   IF menuItem <> gCurrentItem THEN
   BEGIN
      gCurrentItem := menuItem;
      SetMyCommunicationSpeed; {use speed stored in gCurrentItem}
   END;
END; {of DoPopUpMenu}
```

The action procedure for pop-up menus highlights the pop-up menu title, displays the pop-up menu, and handles all user interaction while the user drags up and down the menu. When the user releases the mouse button, the action procedure closes the pop-up box, draws the user's choice in the pop-up box (or restores the previous item if the user doesn't make a new choice), and removes the highlighting of the pop-up title. The pop-up control definition function then changes the value of the `contrlValue` field of the control record to the number of the menu item chosen by the user.

Because buttons do not retain settings, responding to them is very straightforward: when the user clicks a button, your application should immediately undertake the action described by the button's title. For pop-up menus and other types of controls, you must determine their current settings before responding to the user's action. For example, before responding, you need to know which item the user has chosen in a pop-up menu, whether a checkbox is checked, or how far the user has moved the scroll box. The action you take may, in turn, involve changing other control settings. Determining and changing control settings are described in the next section.

After learning how to determine and change control settings, see "Scrolling Through a Document" beginning on page 5-43 for a detailed discussion of how to respond to mouse events in scroll bars.

Determining and Changing Control Settings

Using either the control resource or the parameters to the `NewControl` function, your application specifies a control's various default values—such as its current setting and minimum and maximum settings—when it creates the control.

When the user clicks a control, however, your application often needs to determine the current setting and other possible values of that control. When the user clicks a checkbox, for example, your application must determine whether the box is checked before your application can decide whether to clear or draw a checkmark inside the checkbox. When the user moves the scroll box, your application needs to determine what part of the document to display.

Applications must adjust some controls in response to events other than mouse events in the controls themselves. For example, when the user resizes a window, your application must use the Control Manager procedures `MoveControl` and `SizeControl` to move and resize the scroll bars appropriately.

Your application can use the `GetControlValue` function to determine the current setting of a control, and it can use the `GetControlMaximum` function to determine a control's maximum setting.

You can use the `SetControlValue` procedure to change the control's setting and redraw the control accordingly. You can use the `SetControlMaximum` procedure to change a control's maximum setting and to redraw the indicator or scroll box to reflect the new setting.

In response to user action involving a control, your application often needs to change the setting and possibly redraw the control. When the user clicks a checkbox, for example, your application must determine whether the checkbox is currently selected or not, and then switch its setting. When you use `SetControlValue` to switch a checkbox setting, the Control Manager either draws or removes the X inside the checkbox, as appropriate. When the user clicks a radio button, your application must determine whether the radio button is already on and, if not, turn the previously selected radio button off and turn the newly selected radio button on.

Figure 5-15 on page 5-34 shows a checkbox in the Play Sounds window. When the user clicks the checkbox to turn it on, the application adds a drum roll to the sound it plays whenever the user clicks the Play button.

Listing 5-13 shows the application-defined routine `DoDrumRollCheckBox`, which responds to a click in a checkbox. This routine uses the `GetControlValue` function to determine the last value of the checkbox and then uses the `SetControlValue` procedure to change it. The `GetControlValue` function returns a control's current setting, which is stored in the `contrlValue` field of the control record. The `SetControlValue` procedure sets the `contrlValue` field to the specified value and redraws the control to reflect the new setting. (For checkboxes and radio buttons, the value 1 fills the control with the appropriate mark, and the value 0 removes the mark. For scroll bars, `SetControlValue` redraws the scroll box at the appropriate position along the scroll bar. For a pop-up menu, `SetControlValue` displays in its pop-up box the name of the menu item corresponding to the specified value.)

Listing 5-13 Responding to a click in a checkbox

```
PROCEDURE DoDrumRollCheckBox (mouse: Point; control: ControlHandle);
VAR
   checkbox:Integer;
BEGIN
   IF TrackControl(control, mouse, NIL) <> 0 THEN  {user clicks checkbox}
   BEGIN
      checkbox := GetControlValue(control);  {get last value of checkbox}
      checkbox := 1 - checkbox;              {toggle value of checkbox}
      SetControlValue(control, checkbox);    {set checkbox to new value}
      IF checkbox = 1 THEN                   {the checkbox is checked}
         gPlayDrumRoll := TRUE {play a drum roll next time user clicks Play}
      ELSE
         gPlayDrumRoll := FALSE;
   END;
END;
```

The `DoDrumRollCheckBox` routine uses `TrackControl` to determine which control the user selects. When `TrackControl` reports that the user clicks the checkbox, `DoDrumRollCheckBox` uses `GetControlValue` to determine whether the user last selected the checkbox (that is, whether the control has a current setting of 1) or deselected it (in which case, the control has a current setting of 0). By subtracting the control's current setting from 1, `DoDrumRollCheckBox` toggles to a new setting and then uses `SetControlValue` to assign this new setting to the checkbox. The `SetControlValue` procedure changes the current setting of the checkbox and redraws it appropriately, by either drawing an X in the box if the new setting of the control is 1 or removing the X if the new setting of the control is 0.

Listing 5-4 on page 5-23 shows the control resources that specify a window's scroll bars, and Listing 5-5 on page 5-24 shows an application's `DoNew` routine for creating a document window with these scroll bars. This routine uses the `GetNewControl` function to create the scroll bars and then calls an application-defined routine, `MyAdjustScrollBars`. Listing 5-14 shows `MyAdjustScrollBars`, which in turn calls other application-defined routines that determine the proper sizes, locations, and maximum settings of the scroll bars.

Listing 5-14 Adjusting scroll bar settings and locations

```
PROCEDURE MyAdjustScrollBars (window: WindowPtr;
                              resizeScrollBars: Boolean);
VAR
   myData: MyDocRecHnd;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH myData^^ DO
   BEGIN
      HideControl(vScrollBar);   {hide the vertical scroll bar}
      HideControl(hScrollBar);   {hide the horizontal scroll bar}
      IF resizeScrollBars THEN   {move and size if needed}
         MyAdjustScrollSizes(window);
      MyAdjustScrollValues(window, NOT resizeScrollBars);
      ShowControl(vScrollBar);   {show the vertical scroll bar}
      ShowControl(hScrollBar);   {show the horizontal scroll bar}
   END;
   HUnLock(Handle(myData));
END; {of MyAdjustScrollbars}
```

When calling the DoOpen routine to open an existing document in a window, SurfWriter also uses this MyAdjustScrollBars procedure to size and adjust the scroll bars. When the user changes the window's size, the SurfWriter application uses MyAdjustScrollBars again.

The MyAdjustScrollBars routine begins by getting a handle to the window's document record, which stores handles to the scroll bars as well as other relevant data about the document. (See the chapter "Window Manager" in this book for information about creating your application's own document record for a window.)

Before making any adjustments to the scroll bars, MyAdjustScrollBars passes the handles to these controls to the Control Manager procedure HideControl, which makes the controls invisible. The MyAdjustScrollBars routine then calls another application-defined procedure, MyAdjustScrollSizes (shown in Listing 5-24 on page 5-67), to move and resize the scroll bars appropriately. After calling yet another application-defined procedure, MyAdjustScrollValues, to set appropriate current and maximum settings for the scroll bars, MyAdjustScrollBars uses the Control Manager procedure ShowControl to display the scroll bars in their new locations.

Listing 5-15 shows how the MyAdjustScrollValues procedure calls another application-defined routine, MyAdjustHV, which uses Control Manager routines to assign appropriate settings to the scroll bars.

Listing 5-15 Assigning settings to scroll bars

```
PROCEDURE MyAdjustScrollValues (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH myData^^ DO
   BEGIN
      MyAdjustHV(TRUE, vScrollBar, editRec);
      MyAdjustHV(FALSE, hScrollBar, editRec);
   END;
   HUnLock(Handle(myData));
END; {of MyAdjustScrollValues}
```

To prevent the user from scrolling past the edge of the document and seeing a blank window, you should limit the scroll bars' maximum settings, as illustrated in Figure 5-6 on page 5-10. If the window is larger than the document (which can easily happen with small documents on large monitors), your application should make the maximum scroll bar settings identical to their minimum settings. In this case, the Control Manager then makes the scroll bars inactive, which is appropriate when all the information fits in the window.

Listing 5-16 shows the application-defined `MyAdjustHV` procedure, used for adjusting the current and maximum settings for a scroll bar. When passed `TRUE` in the `isVert` parameter, `MyAdjustHV` calculates and adjusts the maximum and current settings for the vertical scroll bar; when passed `FALSE`, it calculates and adjusts those settings for the horizontal scroll bar.

In this example, the document consists of monostyled text stored in a TextEdit edit record. The `viewRect` field of a TextEdit edit record specifies the rectangle where the text is visible; because `viewRect` already excludes the scroll bar regions, `MyAdjustHV` does not need to subtract the scroll bar regions from the window height or width when calculating the maximum settings for these scroll bars. (For more information about TextEdit in general and the edit record in particular, see *Inside Macintosh: Text*.)

Listing 5-16 Adjusting the maximum and current settings for a scroll bar

```
PROCEDURE MyAdjustHV (isVert: Boolean; control: ControlHandle;
                      editRec: TEHandle);
VAR
   oldValue, oldMax, width:   Integer;
   max, lines, value:         Integer;
BEGIN
   {calculate new maximum and current settings for the vertical or }
   { horizontal scroll bar}
   oldMax := GetControlMaximum(control);
   oldValue := GetControlValue(control);
   MyGetDocWidth(width);
   IF isVert THEN    {adjust max setting for the vertical scroll bar}
   BEGIN
      lines := editRec^^.nLines;
      {since nLines isn't right if the last character is a carriage }
      { return, check for that case}
      IF Ptr(ORD(editRec^^.hText^) + editRec^^.teLength - 1)^ = kCRChar THEN
         lines := lines + 1;
      max := lines - ((editRec^^.viewRect.bottom - editRec^^.viewRect.top)
                      DIV editRec^^.lineHeight);
   END
   ELSE           {adjust max setting for the horizontal scroll bar}
      max := width - (editRec^^.viewRect.right - editRec^^.viewRect.left);
   IF max < 0 THEN
      max := 0;   {check for negative settings}
   SetControlMaximum(control, max); {set the max value of the control}
   IF isVert THEN {adjust current setting for vertical scroll bar}
      value := (editRec^^.viewRect.top - editRec^^.destRect.top)
               DIV editRec^^.lineHeight
   ELSE           {adjust current setting for the horizontal scroll bar}
      value := editRec^^.viewRect.left - editRec^^.destRect.left;
   IF value < 0 THEN
      value := 0
   ELSE IF value > max THEN
      value := max;  {don't allow current setting to be greater than the }
                     { maximum setting}
   SetControlValue(control, value);
END; {of MyAdjustHV}
```

The `MyAdjustHV` routine first uses the `GetControlMaximum` and `GetControlValue` functions to determine the maximum and current settings for the scroll bar being adjusted.

Then `MyAdjustHV` calculates a new maximum setting for the case of a vertical scroll bar. Because the window displays a text-only document, `MyAdjustHV` uses the `nLines` field of the edit record to determine the total number of lines in—and hence, the length of—the document. Then `MyAdjustHV` subtracts the calculated height of the window from the length of the document, and makes this value the maximum setting for the vertical scroll bar.

To calculate the total height in pixels of the window, `MyAdjustHV` begins by subtracting the top coordinate of the view rectangle from its bottom coordinate. (The upper-left corner of a window is normally at point [0,0]; therefore the vertical coordinate of a point at the bottom of a rectangle has a larger value than a point at the top of the rectangle.) Then `MyAdjustHV` divides the pixel height of the window by the value of the edit record's `lineHeight` field, which for monostyled text specifies the document's line height in pixels. By dividing the window height by the line height of the text, `MyAdjustHV` determines the window's height in terms of lines of text.

The `MyAdjustHV` routine uses another application-defined routine, `MyGetDocWidth`, to determine the width of the document. To calculate the width of the window, `MyAdjustHV` subtracts the left coordinate of the view rectangle from its right coordinate. By subtracting the window width from the document width, `MyAdjustHV` derives the maximum setting for the horizontal scroll bar.

For both vertical and horizontal scroll bars, `MyAdjustHV` assigns a maximum setting of 0 whenever the window is larger than the document—for instance, when a window is created for a new document that contains no data yet. In this case, `MyAdjustHV` assigns the same value, 0, to both the maximum and current settings for the scroll bar. The standard control definition function for scroll bars automatically makes a scroll bar inactive when its minimum and maximum settings are identical. This is entirely appropriate, because whenever the user has nowhere to scroll, the scroll bar should be inactive. When you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.

The `MyAdjustHV` routine then uses the Control Manager procedure `SetControlMaximum` to assign the newly calculated maximum settings to either scroll bar. The `SetControlMaximum` procedure revises the control to reflect the new maximum setting; for example, if the user deletes a large portion of the document, thereby reducing the maximum setting, `SetControlMaximum` moves the scroll box to indicate the new position relative to the smaller document.

When the user adds information to or removes information from a document or adjusts its window size, your application may need to adjust the current setting of the scroll bar as well. The `MyAdjustHV` routine calculates a new current setting for the control and then uses `SetControlValue` to assign that setting to the control as well as to reposition the scroll box accordingly.

The destination rectangle, specified in the `destRect` field of the edit record, is the rectangle in which the text is drawn, whereas the view rectangle is the rectangle in which the text is actually visible. By subtracting the top coordinate of the destination rectangle from the top coordinate of the view rectangle, and dividing the result by the line height, `MyAdjustHV` derives the number of the line currently displayed at the top of the window. This is the line number `MyAdjustHV` uses for the current setting of the vertical scroll bar.

To derive the current setting of the horizontal scroll bar in terms of pixels, `MyAdjustHV` subtracts the left coordinate of the destination rectangle from the left coordinate of the view rectangle.

Scrolling Through a Document

Earlier sections of this chapter explain how to create scroll bars, determine when a mouse-down event occurs in a scroll bar, track user actions in a scroll bar, and determine and change scroll bar settings. This section discusses how your application actually scrolls through documents in response to users' mouse activity in the scroll bars. For example, your application scrolls toward the bottom of the document under the following conditions:

- When the user drags the scroll box to the bottom of the vertical scroll bar, your application should display the end of the user's document.
- When the user clicks the gray area below the scroll box, your application should move the document up to display the next window of information toward the bottom of the document, and it should use `SetControlValue` to move the scroll box.
- When the user clicks the down scroll arrow, your application should move the document up by one line (or by some similar measure) and bring more of the bottom of the document into view, and it should use `SetControlValue` to move the scroll box.

As a first step, your application must determine the distance by which to scroll. When the user drags a scroll box to a new location on the scroll bar, you scroll a corresponding distance to a new location in the document.

When the user clicks a scroll arrow, your application determines an appropriate amount to scroll. In general, a word processor scrolls vertically by one line of text and horizontally by the average character width, and a database or spreadsheet scrolls by one field. Graphics applications should scroll to display an entire object when possible. (Typically, applications convert these distances to pixels when using Control Manager, QuickDraw, and TextEdit routines.)

When the user clicks a gray area of a scroll bar, your application should scroll by a distance of just less than the height or width of the window. To determine this height and width, you can use the `contrlOwner` field of the scroll bar's control record. This field contains a pointer to the window record. When you scroll by a distance of one window, it is best to retain part of the previous window. This retained portion helps the user place the material in context. For example, if the user scrolls down by a distance of one window in a text document, the line at the top of the window should be the one that previously appeared at the bottom of the window.

The scrolling direction is determined by whether the scrolling distance is expressed as a positive or negative number. When the user scrolls down or to the right, the scrolling distance is a negative number; when the user scrolls up or to the left, the scrolling distance is a positive number. For example, when the user scrolls from the beginning of a document to a line located 200 pixels down, the scrolling distance is –200 pixels on the vertical scroll bar. When the user scrolls from there back to the start of the document, the scrolling distance is 200 pixels.

Determining the scrolling distance is only the first step. In brief, your application should take the following steps to scroll through a document in response to the user's manipulation of a scroll bar.

1. Use the `FindControl`, `GetControlValue`, and `TrackControl` functions to help calculate the scrolling distance.
2. If you are scrolling for any reason other than the user dragging the scroll box, use the `SetControlValue` procedure to move the scroll box a corresponding amount.
3. Use a routine—such as the QuickDraw procedure `ScrollRect` or the TextEdit procedure `TEPinScroll`—to move the bits displayed in the window by the calculated scrolling distance. Then either use a call that generates an update event or else directly call your application's `DoUpdate` routine, which should perform the rest of these steps.
4. Use the `UpdateControls` procedure to update the scroll bars and then call the Window Manager procedure `DrawGrowIcon` to redraw the size box.
5. Use the QuickDraw procedure `SetOrigin` to change the window origin by an amount equal to the scroll bar settings so that the upper-left corner of the document lies at (0,0) in the window's local coordinate system. (You perform this step so that your application's document-drawing routines can draw in the correct area of the window.)
6. Call your application's routines for redrawing the document inside the window.
7. Use the `SetOrigin` procedure to reset the window origin to (0,0) so that future Window Manager and Control Manager routines draw in the correct area of the window.
8. Return to your event loop.

These steps are explained in greater detail in the rest of this section.

Note
It is not necessary to use `SetOrigin` as described in the rest of this chapter. This procedure merely helps you to offset the window origin by the scroll bars' current settings when you update the window, so that you can locate objects in a document using a coordinate system where the upper-left corner of the document is always at (0,0). As an alternative to this approach, your application can leave the upper-left corner of the window (called the **window origin**) located at (0,0) and instead offset the items in your document by an amount equal to the scroll bars' settings. The QuickDraw procedures `OffsetRect`, `OffsetRgn`, `SubPt`, and `AddPt`, which are described in *Inside Macintosh: Imaging*, are useful if you pursue this alternate approach. u

When the user saves a document, your application should store the data in your own application-defined data structures. (For example, the sample code in this chapter stores a handle to a TextEdit edit record in a *document record*. The edit record contains information about the text, such as it length and its own local coordinate system, and a handle to the text itself.) You typically store information about the objects your application displays onscreen by using coordinates local to the document, where the upper-left corner of the document is located at (0,0).

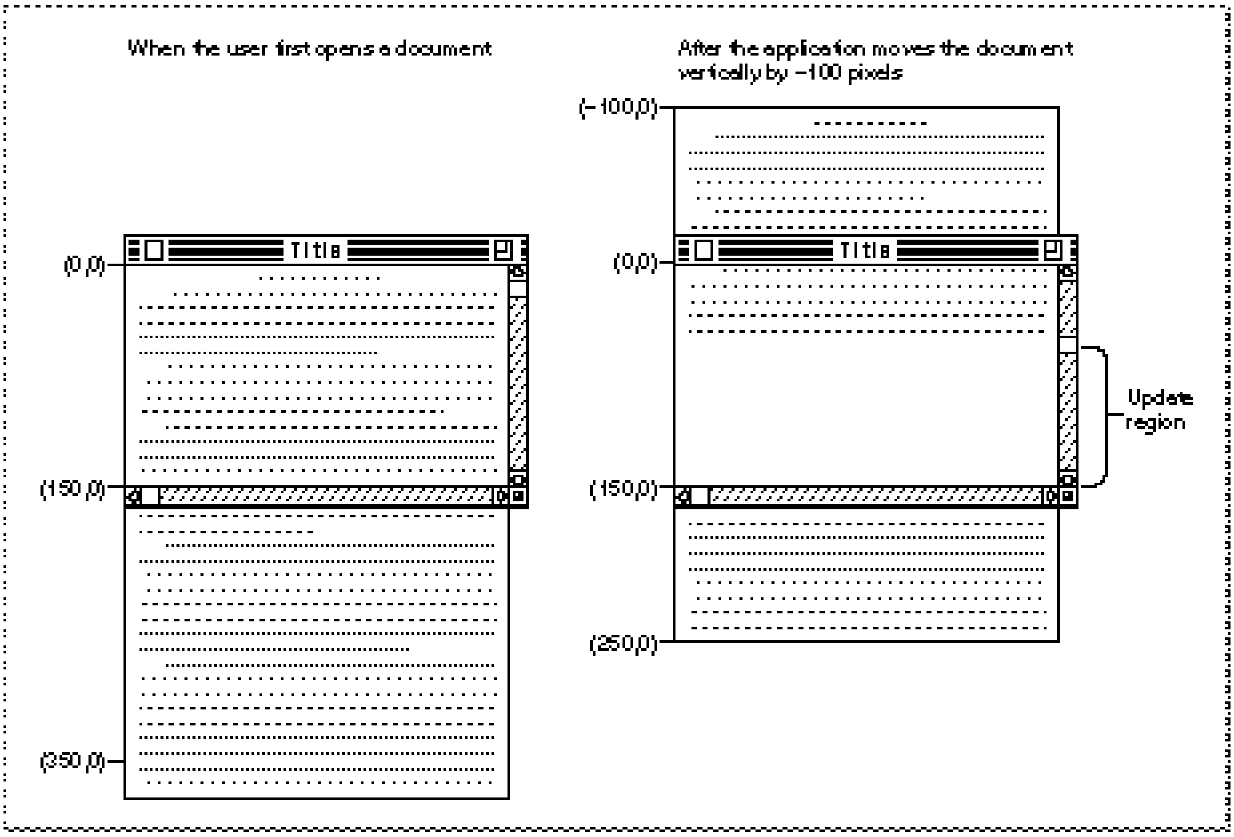
The left side of Figure 5-16 on the next page illustrates a case in which the user has just opened an existing document, and the SurfWriter sample application displays the top of the document. In this example, the document consists of 35 lines of monostyled text, and the line height throughout is 10 pixels. Therefore, the document is 350 pixels long. When the user first opens the document, the window origin is identical to the upper-left point of the document's space: both are at (0,0).

In this example, the window displays 15 lines of text, which amount to 150 pixels. Hence, the maximum setting for the scroll bar is 200 because the vertical scroll bar's maximum setting is the length of the document minus the height of its window.

Imagine that the user drags the scroll box halfway down the vertical scroll bar. Because the user wishes to scroll down, the SurfWriter application must move the text of the document up so that more of the bottom of the document shows. Moving a document *up* in response to a user request to scroll *down* requires a scrolling distance with a *negative* value. (Likewise, moving a document *down* in response to a user request to scroll *up* requires a scrolling distance with a *positive* value.)

Using `FindControl`, `TrackControl`, and `GetControlValue`, the SurfWriter application determines that it must move the document up by 100 pixels—that is, by a scrolling distance of –100 pixels. (Using `FindControl`, `TrackControl`, and `GetControlValue` to determine the scrolling distance is explained in detail in "Scrolling in Response to Events in the Scroll Box" beginning on page 5-53.)

Figure 5-16 Moving a document relative to its window


When the user first opens a document
After the application moves the document vertically by –100 pixels
(0,0)
(150,0)
(350,0)
(–100,0)
(0,0)
(150,0)
(250,0)
Title
Title
Update region

The SurfWriter application then uses the QuickDraw procedure `ScrollRect` to shift the bits displayed in the window by a distance of –100 pixels. The `ScrollRect` procedure moves the document upward by 100 pixels (that is, by 10 lines); 5 lines from the bottom of the previous window display now appear at the top of the window, and the SurfWriter application adds the rest of the window to an update region for later updating.

The `ScrollRect` procedure doesn't change the coordinate system of the window; instead it moves the bits in the window to new coordinates that are still in the window's local coordinate system. For purposes of updating the window, you can think of this as changing the coordinates of the entire document, as illustrated in the right side of Figure 5-16.

The `ScrollRect` procedure takes four parameters: a rectangle to scroll, a horizontal distance to scroll, a vertical distance to scroll, and a region handle. Typically, when specifying the rectangle to scroll, your application passes a value representing the content region minus the scroll bar regions, as shown in Listing 5-17.

Listing 5-17 Using `ScrollRect` to scroll the bits displayed in the window

```
PROCEDURE DoGraphicsScroll (window: WindowPtr;
                            hDistance, vDistance: Integer);
VAR
   myScrollRect: Rect;
   updateRegion: RgnHandle;
BEGIN
   {initially, use the window's portRect as the rectangle to scroll}
   myScrollRect := window^.portRect;
   {subtract vertical and horizontal scroll bars from rectangle}
   myScrollRect.right := myScrollRect.right - 15;
   myScrollRect.bottom := myScrollRect.bottom - 15;
   updateRegion := NewRgn;    {always initialize the update region}
   ScrollRect(myScrollRect, hDistance, vDistance, updateRegion);
   InvalRgn(updateRegion);
   DisposeRgn(updateRegion);
END; {of DoGraphicsScroll}
```

IMPORTANT

You must first pass a horizontal distance as a parameter to `ScrollRect` and then pass a vertical distance. Notice that when you specify a point in the QuickDraw coordinate system, the opposite is true: you name the vertical coordinate first and the horizontal coordinate second. s

Although each scroll bar is 16 pixels along its shorter dimension, the `DoGraphicsScroll` procedure shown in Listing 5-17 subtracts only 15 pixels because the edge of the scroll bar overlaps the edge of the window frame, leaving only 15 pixels of the scroll bar in the content region of the window.

The bits that `ScrollRect` shifts outside of the rectangle specified by `myScrollRect` are not drawn on the screen, and they are not saved—it is your application's responsibility to keep track of this data.

The `ScrollRect` procedure shifts the bits a distance of `hDistance` pixels horizontally and `vDistance` pixels vertically; when `DoGraphicsScroll` passes positive values in these parameters, `ScrollRect` shifts the bits in the `myScrollRect` parameter to the right and down, respectively. This is appropriate when the user intends to scroll left or up, because when the SurfWriter application finishes updating the window, the user sees more of the left and top of the document, respectively. (Remember: to scroll up or left, move the document down or right, both of which are in the positive direction.)

When `DoGraphicsScroll` passes negative values in these parameters, `ScrollRect` shifts the bits in the `myScrollRect` parameter to the left or up. This is appropriate when the user intends to scroll right or down, because when the SurfWriter application finishes updating the window, the user sees more of the right and the bottom of the document. (Remember: to scroll down or right, move the document up or left, both of which are in the negative direction.)

In Figure 5-16, the SurfWriter application determines a vertical scrolling distance of –100, which it passes in the `vDistance` parameter as shown here:

```
ScrollRect(myScrollRect, 0, –100, updateRegion);
```

If, however, the user were to move the scroll box back to the beginning of the document at this point, the SurfWriter application would determine that it has a distance of 100 pixels to scroll up, and it would therefore pass a positive value of 100 in the `vDistance` parameter.

After using `ScrollRect` to move the bits that already exist in the window, the SurfWriter application should draw the bits in the update region of the window by using its standard window-updating code.

As previously explained, `ScrollRect` in effect changes the coordinates of the document relative to the local coordinates of the window. In terms of the window's local coordinate system, the upper-left corner of the document is now at (–100, 0), as shown on the right side of Figure 5-16. To facilitate updating the window, the SurfWriter application uses the QuickDraw procedure `SetOrigin` to change the local coordinate system of the window so that the SurfWriter application can treat the upper-left corner of the document as again lying at (0,0).

The `SetOrigin` procedure takes two parameters: the first is a new horizontal coordinate for the window origin, and the second is a new vertical coordinate for the window origin.

IMPORTANT

Like `ScrollRect`, `SetOrigin` requires you to pass a horizontal coordinate and then a vertical coordinate. Notice that when you specify a point in the QuickDraw coordinate system, the opposite is true: you name the vertical coordinate first and the horizontal coordinate second. s

Any time you are ready to update a window (such as after scrolling it), you can use `GetControlValue` to determine the current setting of the horizontal scroll bar and pass this value as the new horizontal coordinate for the window origin. Then use `GetControlValue` to determine the current setting of the vertical scroll bar and pass this value as the new vertical coordinate for the window origin. Using `SetOrigin` in this fashion shifts the window's local coordinate system so that the upper-left corner of the document is always at (0,0) when you redraw the document within its window.

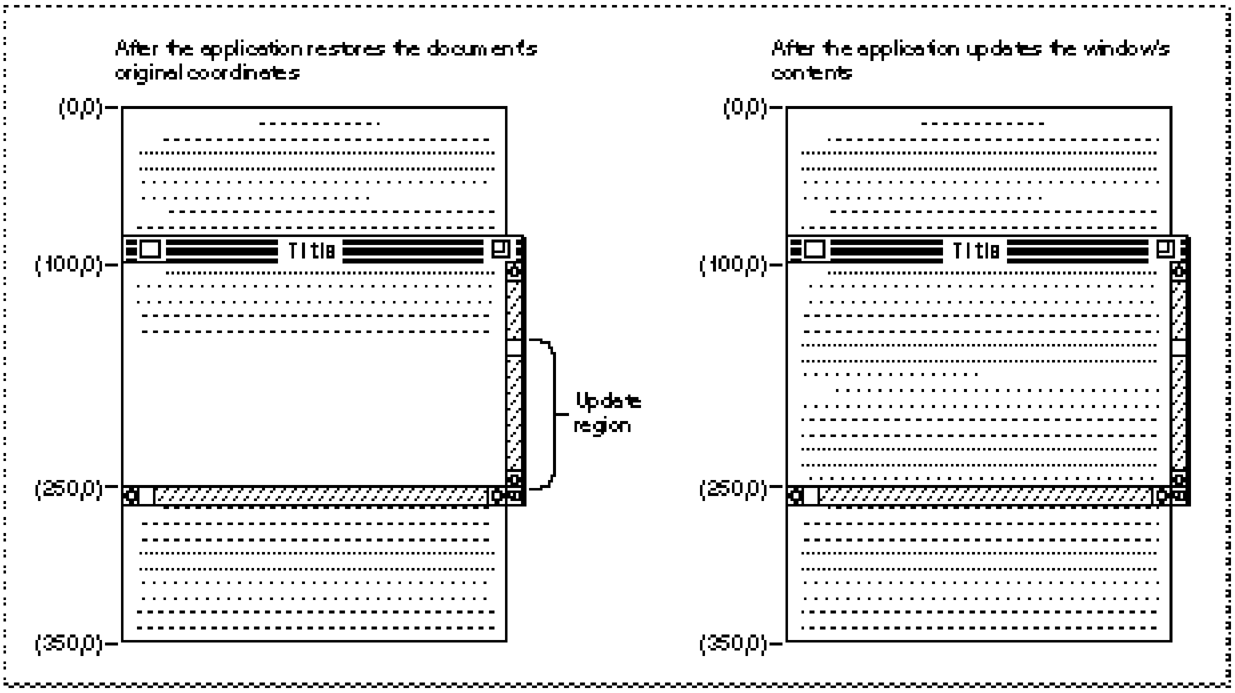
For example, after the user manipulates the vertical scroll bar to move (either up or down) to a location 100 pixels from the top of the document, the SurfWriter application makes the following call:

```
SetOrigin(0, 100);
```

Although the scrolling distance was –100, which is relative, the current setting for the scroll bar is now at 100. (Because you specify a point in the QuickDraw coordinate system by its vertical coordinate first and then its horizontal coordinate, the order of parameters to `SetOrigin` may be initially confusing.)

The left side of Figure 5-17 shows how the SurfWriter application uses the `SetOrigin` procedure to move the window origin to the point (100,0) so that the upper-left corner of the document is now at (0,0) in the window's local coordinate system. This restores the document's original coordinate space and makes it easier for the application to draw in the update region of the window.

Figure 5-17 Updating the contents of a scrolled window


After the application restores the document's original coordinates
After the application updates the window's contents
(0,0)
(100,0)
(250,0)
(350,0)
Title
Update region

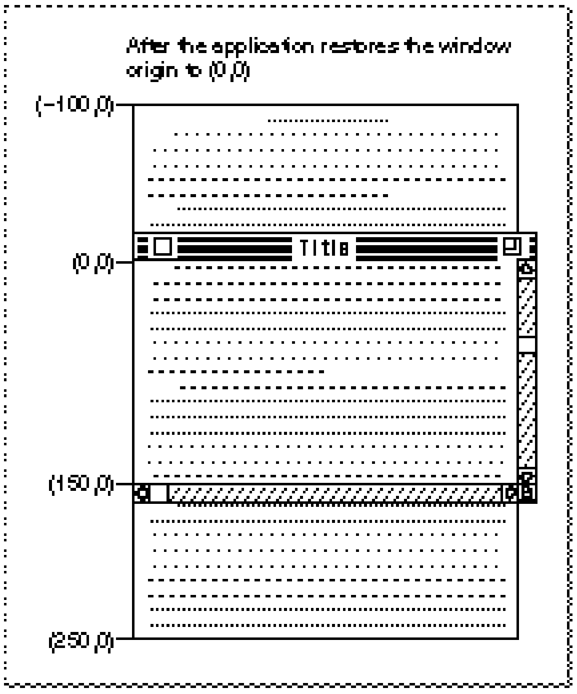
After restoring the document's original coordinates, the SurfWriter application updates the window, as shown on right side of Figure 5-17. The application draws lines 16 through 24, which it stores in its document record as beginning at (160,0) and ending at (250,0).

To review what has happened up to this point: the user has dragged the scroll box one-half of the distance down the vertical scroll bar; the SurfWriter application determines that this distance amounts to a scroll distance of –100 pixels; the SurfWriter application passes this distance to `ScrollRect`, which shifts the bits in the window 100 pixels upward and creates an update region for the rest of the window; the SurfWriter application passes the vertical scroll bar's current setting (100 pixels) in a parameter to `SetOrigin` so that the document's local coordinates are used when the update region of the window is redrawn; and, finally, the SurfWriter application draws the text in the update region of the window.

However, the window origin cannot be left at (100,0); instead, the SurfWriter application must use `SetOrigin` to reset it to (0,0) after performing its own drawing, because the

Window and Control Managers always assume the window's upper-left point is at (0,0) when they draw in a window. Figure 5-18 shows how the application uses `SetOrigin` to set the window origin back to (0,0) at the conclusion of its window-updating routine. After the update, the application begins processing events in its event loop again.

Figure 5-18 Restoring the window origin to (0,0)



The left side of Figure 5-19 illustrates what happens when the user scrolls all the way to the end of the document—a distance of another 10 lines, or 100 pixels. After the SurfWriter application calls `ScrollRect`, the bottom 5 lines from the previous window display appear at the top of the new window and the bottom of the window becomes a new update region. Because the user has scrolled a total distance of 200 pixels, the application uses `SetOrigin` to change the window origin to (200,0), as shown on the right side of Figure 5-19.

The left side of Figure 5-20 shows the SurfWriter application drawing in the update region of the window; the right side of the figure shows the SurfWriter application restoring the window origin to (0,0).

Figure 5-19 Scrolling to the end of a document

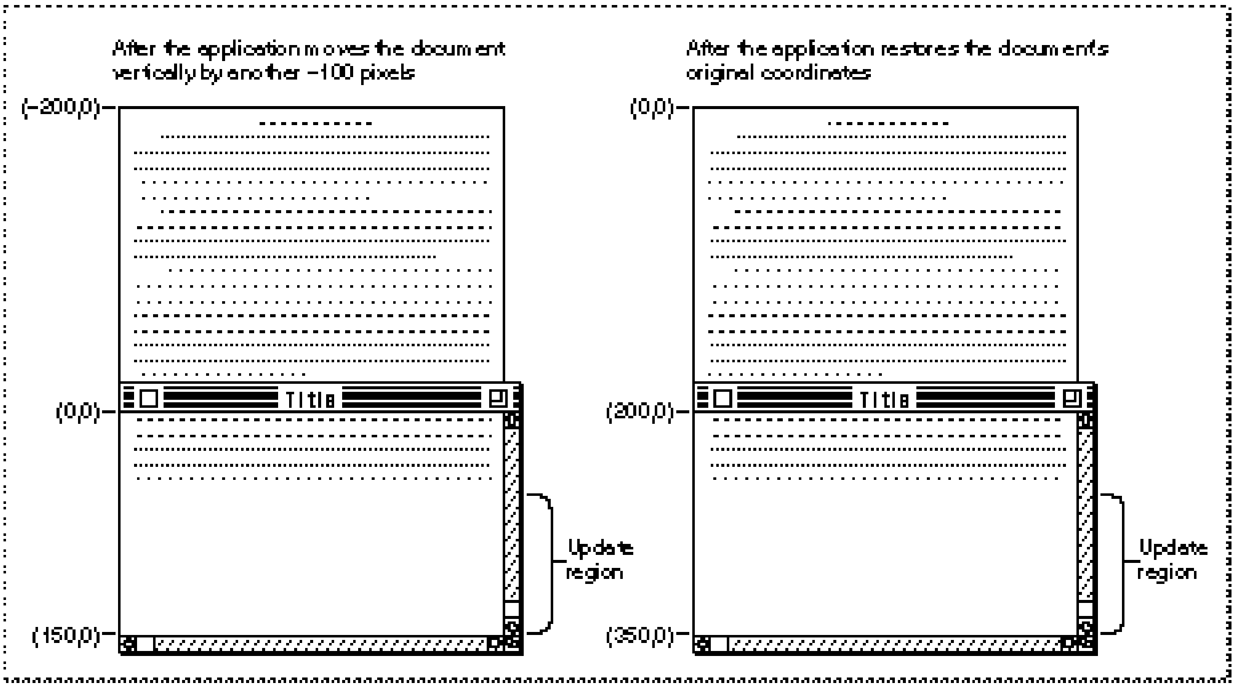
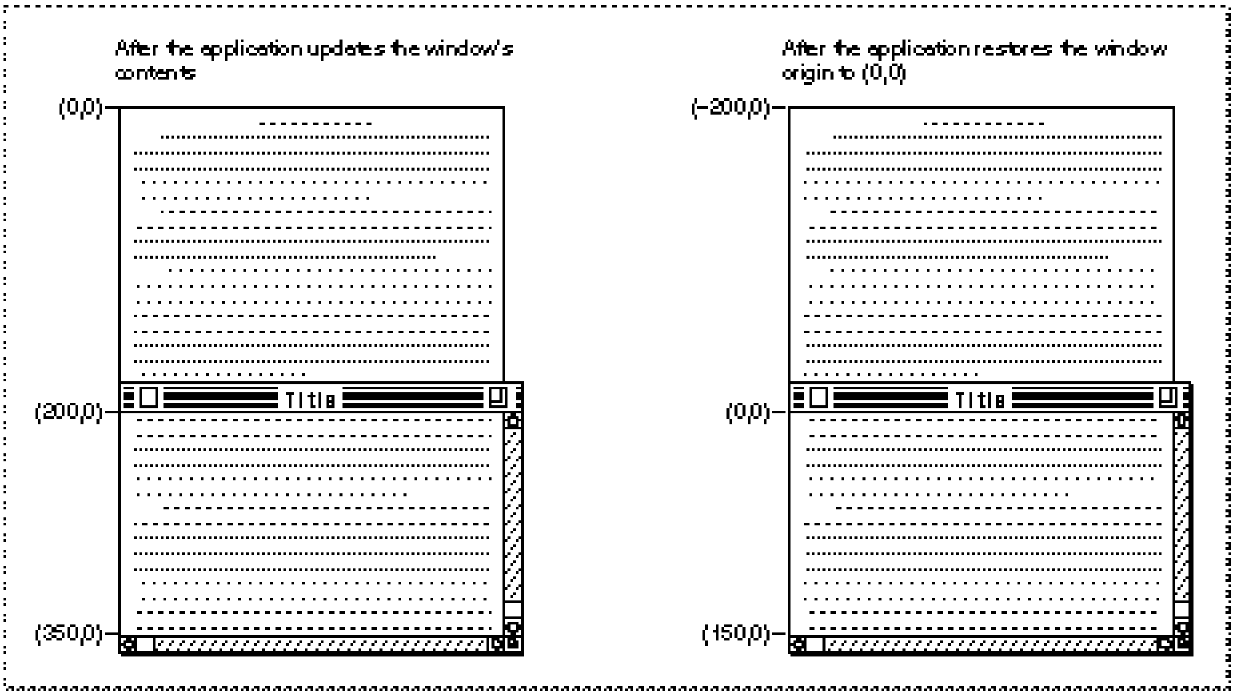

After the application moves the document vertically by another –100 pixels
(–200,0)
(0,0)
(150,0)
Title
Update region
After the application restores the document's original coordinates
(0,0)
(200,0)
(350,0)
Title
Update region

Figure 5-20 Updating a window's contents and returning the window origin to (0,0)


After the application updates the window's contents
(0,0)
(200,0)
(350,0)
Title
After the application restores the window origin to (0,0)
(–200,0)
(0,0)
(150,0)
Title

How your application determines a scrolling distance and how it then moves the bits in the window by this distance are explained in greater detail in the next two sections, "Scrolling in Response to Events in the Scroll Box" and "Scrolling in Response to Events in Scroll Arrows and Gray Areas." "Drawing a Scrolled Document Inside a Window," which follows these two sections, describes what your application should do in its window-updating code to draw in a window that has been scrolled. You can find more detailed information about the `SetOrigin` and `ScrollRect` procedures in *Inside Macintosh: Imaging.*

So far, this discussion has assumed that you are scrolling in response to the user's manipulation of a scroll bar. Most of the time, the user decides when and where to scroll. However, in addition to user manipulation of scroll bars, there are four cases in which your application must scroll through the document. Your application design must take these cases into account.

- When your application performs an operation whose side effect is to make a new selection or move the insertion point, you should scroll to show the new selection. For example, when the user invokes a search operation, your application locates the desired text. If this text appears in a part of the document that isn't currently visible, you should scroll to show the selection. Such scrolling might also be necessary after the user invokes a paste operation. If the insertion point appears after the end of whatever was pasted, scroll until the selection and the new insertion point are visible.
- When the user enters information from the keyboard at the edge of a window, you should scroll to incorporate and display the new information. The user's focus will be on the new information, so it doesn't make sense to maintain the document's position and record the new information out of the user's view. In general, a word processor scrolls one line of text, and a database or spreadsheet scrolls one field. Graphics applications should scroll to display an entire object when possible. Otherwise, determine how quickly your application can redraw the window contents during scrolling and adjust the scrolling to minimize blinking and redrawing. Try to ensure that the scrolling is sufficiently fast so as not to annoy users but not so fast as to confuse them.
- When the user moves the cursor past the edge of the window while holding down the mouse button to make an extended selection, you should scroll the window in the direction of cursor movement. The rate of scrolling can be the same as if the user were holding down the mouse button on the corresponding scroll arrow. In some cases it makes sense to vary the scrolling speed so that it is faster as the user moves the cursor farther away from the edge of the window.
- Sometimes the user selects something, scrolls to a new location, and then tries to perform an operation on the selection. In this case, you should scroll so that the selection is showing before your application performs the operation. Showing the selection makes it clear to the user what is being changed.

When designing the document-scrolling routines for your application, also try to keep the following user interface guidelines in mind:

- Whenever your application scrolls automatically, avoid unnecessary scrolling. Users want to control the position of documents, so your application should move a document only as much as necessary. Thus, if part of a selection is already showing in a window, don't scroll at all. One exception to this rule is when the hidden part of the selection is more important than the visible part; then scroll to show the important part. For example, suppose a user selects a large block of text and only the bottom is currently visible. If the user then types a character, your application must scroll to the location of the newly typed characters so that they are visible.

- If your application can scroll in one orientation to reveal the selection, don't scroll in both orientations. That is, if you can scroll vertically to show the selection, don't also scroll horizontally.
- When you can show context on either side of a selection, it's useful to do so. It's also better to position a selection somewhere near the middle of a window than against a corner. When the selection is too large to fit in the window, it's helpful to display unselected information at either the beginning or the end of the selection to provide context.

Scrolling in Response to Events in the Scroll Box

"Responding to Mouse Events in a Control" beginning on page 5-30 describes in general how to use `FindControl` and `TrackControl` in your event-handling code. Listing 5-18 shows how to use these routines to respond in particular to mouse events in a scroll bar.

Listing 5-18 Responding to mouse events in a scroll bar

```
PROCEDURE DoContentClick (window: WindowPtr; event: EventRecord);
VAR
   mouse:              Point;
   control:            ControlHandle;
   part:               Integer;
   myData:             MyDocRecHnd;
   oldSetting:         Integer;
   scrollDistance:     Integer;
   windowType:         Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window));
            HLock(Handle(myData));
            mouse := event.where;
            GlobalToLocal(mouse);   {convert to local coordinates}
            part := FindControl(mouse, window, control);
            CASE part OF
               {handle all other parts first; handle scroll bar parts last}
            inThumb:    {mouse-down in scroll box}
               BEGIN    {get scroll bar setting}
                  oldSetting := GetControlValue(control);
                  {let user drag scroll box around}
                  part := TrackControl(control, mouse, NIL);
                  {until user releases mouse button}
                  IF part = inThumb THEN
                  BEGIN               {get new distance to scroll}
                     scrollDistance := oldSetting - GetControlValue(control);
                     IF scrollDistance <> 0 THEN
                        IF control = myData^^.vScrollBar THEN
                           TEPinScroll(0, scrollDistance *
                                       myData^^.editRec^^.lineHeight,
                                       myData^^.editRec);
                        ELSE
                           TEPinScroll(scrollDistance, 0, myData^^.editRec);
                  END; {of handling mouse-up in scroll box}
               END; {of handling mouse-down in scroll box}
            inUpButton, inDownButton, inPageUp, in PageDown:
            {mouse-down in scroll arrows or gray areas}
               IF control = myData^^.vScrollBar THEN
                        {handle vertical scroll}
                  part := TrackControl(control, mouse, @MyVerticalActionProc)
               ELSE     {handle horizontal scroll}
                  part := TrackControl(control, mouse, @MyHorzntlActionProc);
            OTHERWISE   ;
            END; {of CASE part}
            HUnLock(Handle(myData));
         END; {of kMyDocWindowType}
         {handle other window types here}
   END; {of CASE windowType}
END;
```

When the user presses the mouse button while the cursor is in a visible, active scroll box, `FindControl` returns as its result the part code for a scroll box. That part code and the constant you can use to represent it are listed here:

Constant	Part code	Control part
`inThumb`	129	Scroll box

As shown in Listing 5-18, when `FindControl` returns the value for `inThumb`, your application should immediately call `GetControlValue` to determine the current setting of the scroll bar. If the user drags the scroll box, you subtract from this setting the new current setting that becomes available when the user releases the mouse button, and you use this result for your scrolling distance.

After using `GetControlValue` to determine the current setting of the scroll bar, use `TrackControl` to follow the movements of the cursor inside the scroll box and to drag a dotted outline of the scroll box in response to the user's movements.

When the user releases the mouse button, `TrackControl` returns `inThumb` if the cursor is still in the scroll box or 0 if the cursor is outside the scroll box. When `TrackControl` returns 0, your application does nothing. Otherwise, your application again uses `GetControlValue` to calculate the distance to scroll.

Calculate the distance to scroll by calling `GetControlValue` and subtracting the new current setting of the scroll bar from its previous setting, which you determine by calling `GetControlValue` before the user releases the mouse button. If this distance is not 0, you should move the bits in the window by this distance and update the contents of the rest of the window.

Before scrolling, you must determine if the scroll bar is a vertical scroll bar or a horizontal scroll bar. As previously explained in this chapter, you should store handles to your scroll bars in a document record, one of which you create for every document. By comparing the field containing the vertical scroll bar handle, you can determine whether the control handle returned by `FindControl` is the handle to the vertical scroll bar. If so, the user has moved the scroll box of the vertical scroll box. If not, the user has moved the scroll box of the horizontal scroll bar.

After determining which scroll bar contains the scroll box that the user has dragged, you move the document contents of the window by the appropriate scrolling distance. That is, for a positive scrolling distance in the vertical scroll bar, move the bits in the window down by that distance. When you update the window, this displays more lines from the top of the document—which is appropriate when the user moves the scroll box *up.* For a positive scrolling distance in the horizontal scroll bar, move the bits in the window to the right by that distance. When you update the window, this displays more lines from the left side of the document—which is appropriate when the user moves the scroll box *to the left.* (Remember: to scroll up or left, move the document down or right, both of which are in the positive direction.)

For a negative scrolling distance in the vertical scroll bar (such as that shown in Figure 5-16 on page 5-46), move the bits in the window up by that distance. When you update the window, this displays more lines from the bottom of the document—which is appropriate when the user moves the scroll box *down.* For a negative scrolling distance in the horizontal scroll bar, move the bits in the window to the left by that distance. When you update the window, this displays more lines from the right side of the document—which is appropriate when the user moves the scroll box *to the right.* (Remember: to scroll down or right, move the document up or left, both of which are in the negative direction.)

The previous examples in this chapter have shown an application that uses a TextEdit edit record to store monostyled text created by the user. For simple text-handling needs, TextEdit provides many routines that simplify your work; for example, the `TEPinScroll` procedure scrolls through the text in the view rectangle of an edit record by the number of pixels specified by your application; `TEPinScroll` stops scrolling when the last line scrolls into the view rectangle.

The TEPinScroll procedure takes three parameters: the number of pixels to move the text horizontally, the number of pixels to move the text vertically, and a handle to an edit record. Positive values in the first two parameters move the text right and down, respectively, and negative values move the text left and up.

The DoContentClick procedure, illustrated in Listing 5-18 on page 5-53, passes the scrolling distance in the second parameter of TEPinScroll for a vertical scroll bar, and it passes the scrolling distance in the first parameter for a horizontal scroll bar.

Listing 5-16 on page 5-41 shows an application-defined routine, MyAdjustHV, called by the SurfWriter sample application whenever it creates, opens, or resizes a window. This routine defines the current and maximum settings for a vertical scroll bar in terms of lines of text.

The DoContentClick procedure on page 5-53 uses GetControlValue to determine the control's current setting—which for the vertical scroll bar DoContentClick calculates as some number of lines. When determining the vertical scroll bar's scrolling distance, DoContentClick again calculates a value representing some number of lines.

However, TEPinScroll expects pixels, not lines, to be passed in its parameters. Therefore, DoContentClick multiplies the scrolling distance (which it calculates as some number of lines of text) by the line height (which is maintained in the edit record for monostyled text as some number of pixels). In this way, DoContentClick passes a scrolling distance—in terms of pixels—to TEPinScroll, as shown in this code fragment.

```
IF control = myData^^.vScrollBar THEN
   TEPinScroll(0, scrollDistance * myData^^.editRec^^.lineHeight,
               myData^^.editRec);
```

Figure 5-16 on page 5-46 illustrates a scrolling distance of –10 lines. If the line height is 10 pixels, the SurfWriter application passes –100 as the second parameter to TEPinScroll.

The TEPinScroll procedure adds the scrolled-away area to the update region and generates an update event so that the text in the edit record's view rectangle can be updated. In its code that handles update events for windows, the SurfWriter sample application then uses the TEUpdate procedure—as described in "Drawing a Scrolled Document Inside a Window" beginning on page 5-62— for its windows that include TextEdit edit records.

To learn more about TEPinScroll, the TextEdit edit record, and other facilities offered by TextEdit, see *Inside Macintosh: Text.*

The QuickDraw procedure ScrollRect is a more general-purpose routine for moving bits in a window when scrolling. If you use ScrollRect to scroll the bits displayed in the window, you should define a routine like DoGraphicsScroll, shown in Listing 5-17 on page 5-47, and use it instead of TEPinScroll, which is used in Listing 5-18 on page 5-53.

The ScrollRect procedure returns in the updateRegion parameter the area that needs to be updated. The DoGraphicsScroll procedure shown in Listing 5-17 on page 5-47 then uses the QuickDraw procedure InvalRgn to add this area to the update region, forcing an update event. In your code for handling update events, you draw in the area of the window from which `ScrollRect` has moved the bits, as described in “Drawing a Scrolled Document Inside a Window” beginning on page 5-62.

When a mouse-down event occurs in the scroll arrows or gray areas of the vertical scroll bar, the `DoContentClick` routine in Listing 5-18 on page 5-53 calls `TrackControl` and passes it a pointer to an application-defined action procedure called `MyVerticalActionProc`. For the horizontal scroll bar, `DoContentClick` calls `TrackControl` and passes it a pointer to an action procedure called `MyHorzntlActionProc`. These action procedures are described in the next section.

Scrolling in Response to Events in Scroll Arrows and Gray Areas

With each click in a scroll arrow, your application should scroll by a distance of one unit (that is, by a single line, character, cell, or whatever your application deems appropriate) in the chosen direction. When the user holds the mouse button down while the cursor is in a scroll arrow, your application should scroll continuously by single units until the user releases the mouse button or until your application has scrolled as far as possible in the document.

With each click in a gray area, your application should scroll in the appropriate direction by a distance of just less than the height or width of one window to show part of the previous window (thus placing the newly displayed material in context). When the user holds the mouse button down while the cursor is in a gray area, your application should scroll continuously in units of this distance until the user releases the mouse button or until your application has scrolled as far as possible in the document.

When your application finishes scrolling, it should use `SetControlValue` to move the scroll box accordingly.

As previously described in this chapter, you use `FindControl` to determine when a mouse-down event has occurred in a control in one of your windows, and you use `TrackControl` to follow the movements of the cursor inside the control, to give the user visual feedback, and then to inform your application when the user releases the mouse button.

When a mouse-down event occurs in the scroll arrows or the gray areas of an active scroll bar, `FindControl` returns as its result the appropriate part code. The part codes for the scroll arrows and gray areas, and the constants you can use to represent them, are listed here:

Constant	Part code	Control part
`inUpButton`	20	Up scroll arrow for a vertical scroll bar, left scroll arrow for a horizontal scroll bar
`inDownButton`	21	Down scroll arrow for a vertical scroll bar, right scroll arrow for a horizontal scroll bar
`inPageUp`	22	Gray area above scroll box for a vertical scroll bar, gray area to left of scroll box for a horizontal scroll bar
`inPageDown`	23	Gray area below scroll box for a vertical scroll bar, gray area to right of scroll box for a horizontal scroll bar

When FindControl returns one of these part codes, your application should immediately call TrackControl. As long as the user holds down the mouse button while the cursor is in a scroll arrow, TrackControl highlights the scroll arrow, as shown in Figure 5-8 on page 5-12. When the user releases the mouse button, TrackControl removes the highlighting.

For all of the other standard controls, as well as for the scroll box in a scroll bar, your application doesn't respond until TrackControl reports a mouse-up event in the same control part where the mouse-down event initially occurred. However, for scroll arrows and gray areas, your application must respond by scrolling the document *before* TrackControl reports that the user has released the mouse button. When you call TrackControl for scroll arrows and gray areas, you must define an action procedure that scrolls appropriately until TrackControl reports that the user has released the mouse button.

When the user releases the mouse button or moves the cursor away from the scroll arrow or gray area, TrackControl returns as its result one of the previously listed values that represent the control part. As shown in Listing 5-18 on page 5-53, the DoContentClick procedure tests for the part codes inUpButton, inDownButton, inPageUp, and inPageDown to determine when a mouse-down event occurs in a scroll arrow or a gray area.

When the user presses or holds down the mouse button while the cursor is in either the scroll arrow or the gray area of the vertical scroll bar, DoContentClick calls TrackControl and passes it a pointer to an application-defined action procedure called MyVerticalActionProc. For the horizontal scroll bar, DoContentClick calls TrackControl and passes it a pointer to an action procedure called MyVerticalActionProc. In turn, TrackControl calls these action procedures to scroll continuously until the user releases the mouse button.

Note

As an alternative to passing a pointer to your action procedure in a parameter to TrackControl, you can use the SetControlAction procedure to store a pointer to the action procedure in the contrlAction field in the control record. When you pass Pointer(-1) instead of a procedure pointer to TrackControl, TrackControl uses the action procedure pointed to in the control record. u

Listing 5-19 shows two sample action procedures: MyVerticalActionProc—which responds to mouse events in the scroll arrows and gray areas of a vertical scroll bar—and MyHorzntlActionProc—which responds to those same events in a horizontal scroll bar. When TrackControl calls these action procedures, it passes a control handle and an integer representing the part of the control in which the mouse event occurred. Both MyVerticalActionProc and MyHorzntlActionProc use the constants inUpButton, inDownButton, inPageUp, and inPageDown to test for the control part passed by TrackControl.

Listing 5-19 Action procedures for scrolling through a text document

```
PROCEDURE MyVerticalActionProc (control: ControlHandle; part: Integer);
VAR
   scrollDistance:   Integer;
   window:           WindowPtr;
   myData:           MyDocRecHnd;
BEGIN
   IF part <> 0 THEN
   BEGIN
      window := control^^.contrlOwner; {get the control's window}
      myData := MyDocRecHnd(GetWRefCon(window));
      HLock(Handle(myData));
      CASE part OF
         inUpButton, inDownButton:  {get one line to scroll}
            scrollDistance := 1;
         inPageUp, inPageDown:   {get the window's height}
         BEGIN
            scrollDistance := (myData^^.editRec^^.viewRect.bottom -
                              myData^^.editRec^^.viewRect.top)
                              DIV myData^^.editRec^^.lineHeight;
               {subtract 1 line so user sees part of previous window}
            scrollDistance := scrollDistance - 1;
         END;
      END;  {of part CASE}
      IF (part = inDownButton) OR (part = inPageDown) THEN
         scrollDistance := -scrollDistance;
      MyMoveScrollBox(control, scrollDistance);
      IF scrollDistance <> 0 THEN   {scroll by line or by window}
         TEPinScroll(0, scrollDistance * myData^^.editRec^^.lineHeight,
                     myData^^.editRec);
      HUnLock(Handle(myData));
   END;
END; {of MyVerticalActionProc}

PROCEDURE MyHorzntlActionProc (control: ControlHandle; part: Integer);
VAR
   scrollDistance:   Integer;
   window:           WindowPtr;
   myData:           MyDocRecHnd;
BEGIN
   IF part <> 0 THEN
   BEGIN
      window := control^^.contrlOwner; {get the control's window}
      myData := MyDocRecHnd(GetWRefCon(window));
      HLock(Handle(myData));
      CASE part OF
         inUpButton, inDownButton:  {get a few pixels}
            scrollDistance := kButtonScroll;
         inPageUp, inPageDown:   {get a window's width}
            scrollDistance := myData^^.editRec^^.viewRect.right -
                                 myData^^.editRec^^.viewRect.left;
      END;  {of part CASE}
      IF (part = inDownButton) OR (part = inPageDown) THEN
         scrollDistance := -scrollDistance;
      MyMoveScrollBox(control, scrollDistance);
      IF scrollDistance <> 0 THEN
         TEPinScroll(scrollDistance, 0, myData^^.editRec);
      HUnLock(Handle(myData));
   END;
END; {of MyHorzntlActionProc}
```

Each action procedure begins by determining an appropriate scrolling distance. For the scroll arrows in a vertical scroll bar, `MyVerticalActionProc` defines the scrolling distance as one line. For the gray areas in a vertical scroll bar, `MyVerticalActionProc` determines the scrolling distance in lines by dividing the window height by the line height; the window height is determined by subtracting the bottom coordinate of the view rectangle (defined in the edit record) from its top coordinate. Then `MyVerticalActionProc` subtracts 1 from this distance so that when the user presses the mouse button while the cursor is in a gray area, `MyVerticalActionProc` scrolls one line less than the total number of lines in the window.

The `MyVerticalActionProc` procedure later multiplies these line distances by the line height to derive pixel distances to pass in parameters to `TEPinScroll`. Also, `MyVerticalActionProc` turns these distances into negative values when the mouse-down event occurs in the lower scroll arrow or in the gray area below the scroll box.

For the scrolling distance of the scroll arrows in horizontal scroll bars, `MyHorzntlActionProc` uses a predetermined pixel distance—roughly the document's average character width. For the scrolling distance of the gray areas `MyHorzntlActionProc` uses the window width (which is derived by subtracting the left coordinate of the view rectangle from its right coordinate). The `MyHorzntlActionProc` routine turns these distances into negative values when the mouse-down event occurs in the right scroll arrow or in the gray area to the right of the scroll box.

After calling `MyMoveScrollBox`, an application-defined routine that moves the scroll box, both action procedures use `TEPinScroll` to move the text displayed in the window by the scrolling distance. (In this example, the SurfWriter application is scrolling a simple monostyled text document stored as a TextEdit edit record. For a discussion of using the more general-purpose QuickDraw scrolling routine `ScrollRect`, see the previous section, “Scrolling in Response to Events in the Scroll Box” beginning on page 5-53.)

The `TEPinScroll` procedure automatically creates an update region and invokes an update event. In its window-updating code, the SurfWriter application uses the `TEUpdate` procedure to draw the text in the update region, as shown in Listing 5-23 on page 5-65.

The action procedures continue moving the text by the specified distances over and over until the user releases the mouse button and `TrackControl` completes. If there is no more area to scroll through, `TEPinScroll` automatically stops scrolling, as your application should if you implement your own scrolling routine.

Listing 5-20 shows how the application-defined procedure `MyMoveScrollBox` uses `GetControlValue`, `GetControlMaximum`, and `SetControlValue` to move the scroll box an appropriate distance while the action procedures scroll through the document. The `MyMoveScrollBox` procedure uses `GetControlMaximum` to determine the maximum scrolling distance, `GetControlValue` to determine the current setting for the scroll box, and `SetControlValue` to assign the new setting and move the scroll box. Use of the `SetControlMaximum` and `SetControlValue` routines is described in “Determining and Changing Control Settings” beginning on page 5-37; `GetControlMaximum` is described in detail on page 5-104.

Listing 5-20 Moving the scroll box from the action procedures

```
PROCEDURE MyMoveScrollBox (control: ControlHandle;
                           scrollDistance: Integer);
VAR
   oldSetting, setting, max:  Integer;
BEGIN
   oldSetting := GetControlValue(control);   {get last setting}
   max := GetControlMaximum(control);        {get maximum setting}
{subtract action procs' scroll amount from last setting to get new setting}
   setting := oldSetting - scrollDistance;
   IF setting < 0 THEN
      setting := 0
   ELSE IF setting > max THEN
      setting := max;
   SetControlValue(control, setting); {assign new current setting}
END; {of MyMoveScrollBox}
```

The previous two sections have described how to move the bits displayed in the window; the next section describes how to draw into the update region.

Drawing a Scrolled Document Inside a Window

The previous two sections have described how to use the QuickDraw procedure `ScrollRect` and the TextEdit procedure `TEPinScroll` in response to the user manipulating any of the five parts of a scroll bar. After using these or your own routines for moving the bits in your window, your application must draw into the update region. Typically, you use your own window-updating code for this purpose.

Both `InvalRect` and `TEPinScroll`, which are used in the examples shown earlier in this chapter, create update regions that cause update events. As described in the chapters "Window Manager" and "Event Manager" in this book, your application should draw in the update regions of your windows when it receives update events. If you create your own scrolling routine to use instead of `ScrollRect` or `TEPinScroll`, you should guarantee that it generates an update event or that it explicitly calls your own window-updating routine.

Listing 5-21 shows an application-defined routine, `DoUpdate`, that the SurfWriter application calls whenever it receives an update event. In this procedure, the application tests for two different types of windows: windows containing graphics objects and windows containing text created with TextEdit routines.

Listing 5-21 An application-defined update routine

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyGraphicsWindow:   {window containing graphics objects}
      BEGIN
         BeginUpdate(window);
         MyDrawGraphicsWindow(window);
         EndUpdate(window);
      END;  {of updating graphics windows}
   kMyDocWindow:        {window containing TextEdit text}
      BEGIN
         BeginUpdate(window);
         MyDrawWindow(window);
         EndUpdate(window);
      END;  {of updating TextEdit document windows}
   {handle other window types——modeless dialogs, etc.——here}
   END;  {of windowType CASE}
END;  {of DoUpdate}
```

In this example, when the window requiring updating is of type `kMyGraphicsWindow`, `DoUpdate` uses another application-defined routine called `MyDrawGraphicsWindow`. When the window requiring updating is of type `kMyDocWindow`, `DoUpdate` uses another application-defined routine—namely, `MyDrawWindow`. Listing 5-22 shows the `MyDrawGraphicsWindow` routine and Listing 5-23 on page 5-65 shows the `MyDrawWindow` routine.

Before drawing into the scrolled-away portion of the window, both of these routines use the QuickDraw, Window Manager, and Control Manager routines necessary for updating windows. (“Updating a Control” beginning on page 5-29 describes the `UpdateControls` procedure; see the chapter “Window Manager” in this book for a detailed description of how to use the rest of these routines to update a window.)

Listing 5-22 Redrawing a window containing graphics objects

```
PROCEDURE MyDrawGraphicsWindow (window: WindowPtr);
VAR
   myData:  MyDocRecHnd;
   i:       Integer;
BEGIN
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
      BEGIN
         EraseRect(portRect);
         UpdateControls(window, visRgn);
         DrawGrowIcon(window);
         SetOrigin(GetControlValue(myData^^.hScrollBar),
                   GetControlValue(myData^^.vScrollBar));
         i := 1;
         WHILE i <= myData^^.numObjects DO
            DrawMyObjects(portRect, myData^^.numObjects[i]);
            i := i + 1;
         END; {of WHILE}
         SetOrigin(0, 0);
      END;
   HUnLock(Handle(myData));
END;  {of MyDrawGraphicsWindow}
```

The `MyDrawGraphicsWindow` routine uses the QuickDraw procedure `SetOrigin` to change the window origin by an amount equal to the scroll bar settings, so that the upper-left corner of the document lies at (0,0) in the window’s local coordinate system. The SurfWriter sample application performs this step so that its own drawing routines can draw into the correct area of the window.

Notice that MyDrawGraphicsWindow calls SetOrigin only after calling the necessary Window Manager and Control Manager routines, because the Window Manager and Control Manager always expect the window origin to be at (0,0).

By using SetOrigin to change the window origin, MyDrawGraphicsWindow can treat the objects in its document as being located in a coordinate system where the upper-left corner of the document is always at (0,0). Then MyDrawGraphicsWindow calls another of its own routines, DrawMyObjects, to draw the objects it has stored in its document record for the window.

After performing all its own drawing in the window, MyDrawGraphicsWindow again uses SetOrigin—this time to reset the window origin to (0,0) so that future Window Manager and Control Manager routines will draw into the correct area of the window.

Figure 5-16 through Figure 5-20 earlier in this chapter help to illustrate how to use SetOrigin to offset the window's coordinate system so that you can treat the objects in your document as fixed in the document's own coordinate space. However, it is not necessary for your application to use SetOrigin. Your application can leave the window's coordinate system fixed and instead offset the items in your document by the amount equal to the scroll bar settings. The QuickDraw procedures OffsetRect, OffsetRgn, SubPt, and AddPt, which are described in *Inside Macintosh: Imaging,* are useful if you pursue this approach.

Note

The SetOrigin procedure does not move the window's clipping region. If you use clipping regions in your windows, use the QuickDraw procedure GetClip to store your clipping region immediately after your first call to SetOrigin. Before calling your own window-drawing routine, use the QuickDraw procedure ClipRect to define a new clipping region—to avoid drawing over your scroll bars, for example. After calling your own window-drawing routine, use the QuickDraw procedure ClipRect to restore the original clipping region. You can then call SetOrigin again to restore the window origin to (0,0) with your original clipping region intact. See *Inside Macintosh: Imaging* for detailed descriptions of clipping regions and of these QuickDraw routines. u

The previous examples in this chapter have shown an application that uses a TextEdit edit record to store the information created by the user. For simple text-handling needs, TextEdit provides many routines that simplify your work; for example, the TEPinScroll procedure (used in Listing 5-18 on page 5-53 and Listing 5-19 on page 5-59) resets the view rectangle of text stored in an edit record by the amount of pixels specified by the application. The TEPinScroll procedure then generates an update event for the window. The TextEdit procedure TEUpdate should then be called in an application's update routine to draw the update region of the scrolled window.

Listing 5-23 shows an application-defined procedure, MyDrawWindow, that uses TEUpdate to update the text in windows of type kMyDocWindow. The TEUpdate procedure manages all necessary shifting of coordinates during window updating, so MyDrawWindow does not have to call SetOrigin as it does when it uses ScrollRect.

Listing 5-23 Redrawing a window after scrolling a TextEdit edit record

```
PROCEDURE MyDrawWindow (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
      BEGIN
         EraseRect(portRect);
         UpdateControls(window, visRgn);
         DrawGrowIcon(window);
         TEUpdate(portRect, myData^^.editRec);
      END;
   HUnLock(Handle(myData));
END;  {of MyDrawWindow}
```

Moving and Resizing Scroll Bars

As described earlier in “Creating Scroll Bars” beginning on page 5-21, your application initially defines the location of a scroll bar within a window—and the size of the scroll bar—by specifying a rectangle in a control resource or in a parameter to `NewControl`. However, your application must be able to size and move the scroll bar dynamically in response to the user’s resizing of your windows.

The chapter “Window Manager” in this book describes how to size windows when your application opens them and how to resize them—for example, in response to the user dragging the size box or clicking the zoom box. This section describes how to move and resize your scroll bars so that they fit properly on the right and bottom edges of your windows.

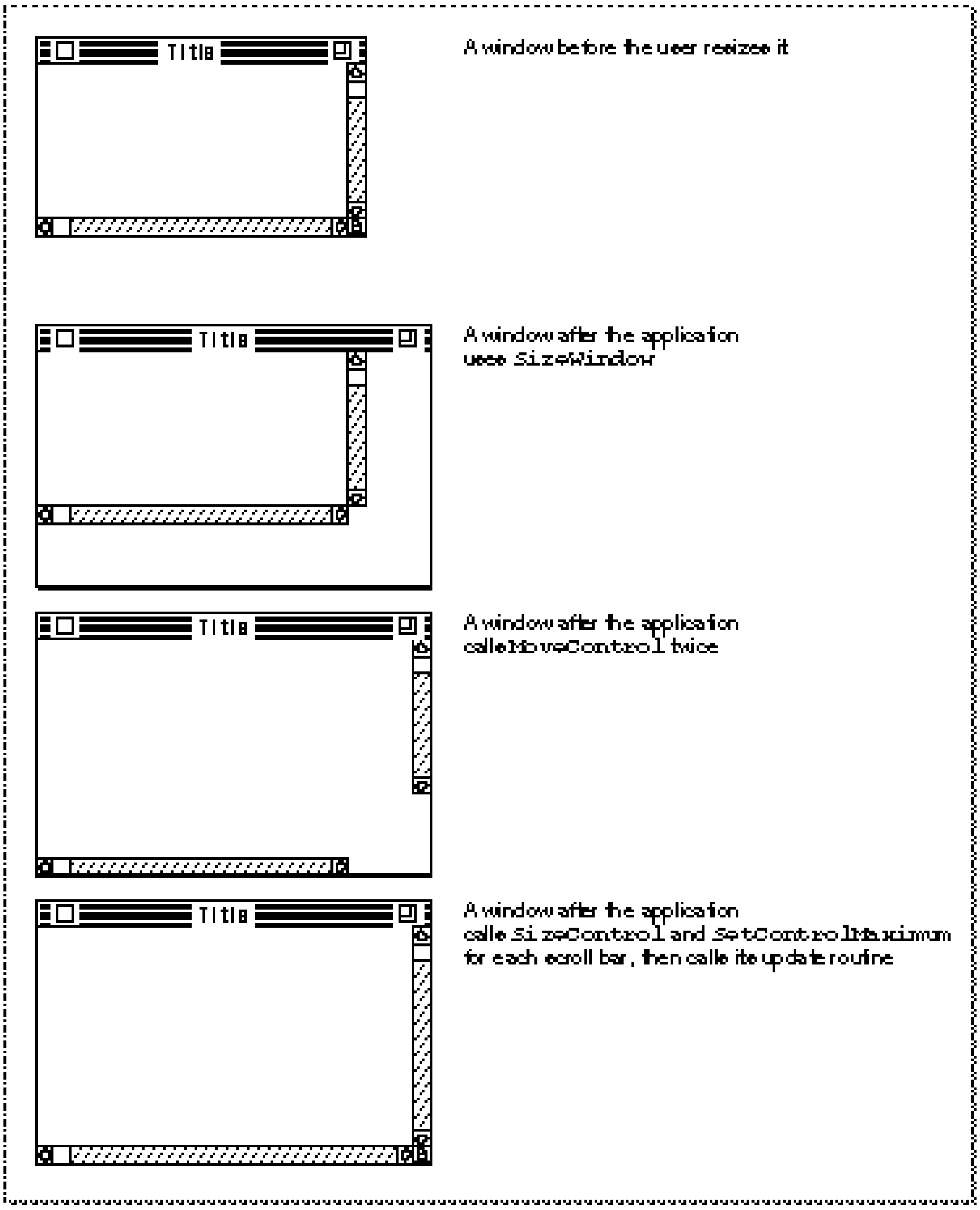
When resizing your windows, your application should perform the following steps to adjust each scroll bar.

1. Resize the window.
2. Use the `HideControl` procedure to make each scroll bar invisible.
3. Use the `MoveControl` procedure to move the vertical scroll bar to the right edge of the window, and use the `MoveControl` procedure to move the horizontal scroll bar to the bottom edge of the window.
4. Use the `SizeControl` procedure to lengthen or shorten each scroll bar, so that each extends to the size box in the lower-right corner of the window.
5. Recalculate the maximum settings for the scroll bars and use `SetControlMaximum` to update the settings and to redraw the scroll boxes appropriately. (Remember, you derive a scroll bar’s maximum setting by subtracting the length or width of its window from the length or width of the document.)

6. Use the `ShowControl` procedure to make each scroll bar visible in its new location.

Figure 5-21 illustrates how to move and resize scroll bars in a resized window; if your application neglected to use the `HideControl` procedure, the user would see each of these steps as it took place.

Figure 5-21 Moving and resizing scroll bars


Title
A window before the user resizes it
Title
A window after the application uses SizeWindow
Title
A window after the application calls MoveControl twice
Title
A window after the application calls SizeControl and SetControlMaximum for each scroll bar, then calls its update routine

Listing 5-14 on page 5-39 shows an application-defined routine, `MyAdjustScrollBars`, that is called when the user opens a new window, opens an existing document in a window, or resizes a window.

When it creates a window, `MyAdjustScrollBars` stores handles to each scroll bar in a document record. By dereferencing the proper fields of the document record, `MyAdjustScrollBars` passes handles for the vertical and horizontal scroll bars to the `HideControl` procedure, which makes the scroll bars invisible. By making the scroll bars invisible until it has finished manipulating them, `MyAdjustScrollBars` ensures that the user won't see the scroll bars blinking in different locations onscreen.

When `MyAdjustScrollBars` needs to adjust the size or location of either of the scroll bars, it calls another application-defined routine, `MyAdjustScrollSizes`, which is shown in Listing 5-24.

Listing 5-24 Changing the size and location of a window's scroll bars

```
CONST
   kScrollbarWidth = 16;                    {conventional width}
   kScrollbarAdjust = kScrollbarWidth - 1;  {to align with window frame}
   kScrollTweek = 2;                  {to align scroll bars with size box}

PROCEDURE MyAdjustScrollSizes (window: WindowPtr);
VAR
   teRect:                          Rect;
   myData:                          MyDocRecHnd;
   teTop, teRight, teBottom,teLeft: Integer;
BEGIN
   MyGetTERect(window, teRect);  {calculate the teRect based on the }
                                 { portRect, adjusted for the scroll bars}
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^.portRect DO
   BEGIN
      teTop := top;
      teRight := right;
      teBottom := bottom;
      teLeft := left;
   END;
   WITH myData^^ DO
   BEGIN
      editRec^^.viewRect := teRect;    {set the viewRect}
      MyAdjustViewRect(editRec);       {snap to nearest line}
      {move the controls to match the new window size}
      MoveControl(vScrollBar, teRight - kScrollbarAdjust, -1);
      SizeControl(vScrollBar, kScrollbarWidth, (teBottom - teTop) -
                  (kScrollbarAdjust - kScrollTweek));
      MoveControl(hScrollBar, -1, teBottom - kScrollbarAdjust);
      SizeControl(hScrollBar, (teRight - teLeft) -
            (kScrollbarAdjust - kScrollTweek), kScrollbarWidth);
   END;
   HUnLock(Handle(myData));
END; {of MyAdjustScrollSizes}
```

The `MyAdjustScrollSizes` routine uses the boundary rectangle of the window's content region—which is stored in the `portRect` field of the window record—to determine the size of the window. To move the scroll bars to the edges of the window, `MyAdjustScrollSizes` uses the `MoveControl` procedure.

The `MoveControl` procedure takes three parameters: a handle to the control being moved, the horizontal coordinate (local to the control's window) for the new location of the upper-left corner of the control's rectangle, and the vertical coordinate for that new location. The `MoveControl` procedure moves the control to this new location and changes the rectangle specified in the `controlRect` field of the control's control record.

In Listing 5-24, `MyAdjustScrollSizes` passes to `MoveControl` the handles to the scroll bars. (The SurfWriter sample application stores the handle in its document record for the window.)

Figure 5-22 illustrates the location of a vertical scroll bar before it is moved to a new location within its resized window.

To determine a new horizontal (that is, left) coordinate of the upper-left corner of the vertical scroll bar, `MyAdjustScrollSizes` subtracts 15 from the right coordinate of the window. As shown in Figure 5-23, this puts the right edge of the 16-pixel-wide scroll bar directly over the 1-pixel-wide window frame on the right side of the window.

In Listing 5-24 on page 5-67, `MyAdjustScrollSizes` specifies –1 as the vertical (that is, top) coordinate of the upper-left corner of the vertical scroll bar. As shown in Figure 5-23, this places the top edge of the scroll bar directly over the 1-pixel-wide line at the bottom of the title bar. (The bottom line of the title bar has a vertical value of –1 in the window's local coordinate system.)

The `MyAdjustScrollSizes` routine specifies –1 as the horizontal coordinate of the upper-left corner of the horizontal scroll bar; this puts the left edge of the horizontal scroll bar directly over the 1-pixel-wide window frame. (The left edge of the window frame has a horizontal value of –1 in the window's local coordinate system.)

To fit your scroll bars inside the window frame properly, you should set the top coordinate of a vertical scroll bar at –1 and the left coordinate of a horizontal scroll bar at –1, unless your application uses part of the window's scroll regions opposite the size box for displaying information or additional controls. For example, you may choose to display the current page number of the document in the lower-left corner of a window. In this case, specify a left coordinate so that the horizontal scroll bar doesn't obscure this area.

Figure 5-22 A vertical scroll bar before the application moves it within a resized window

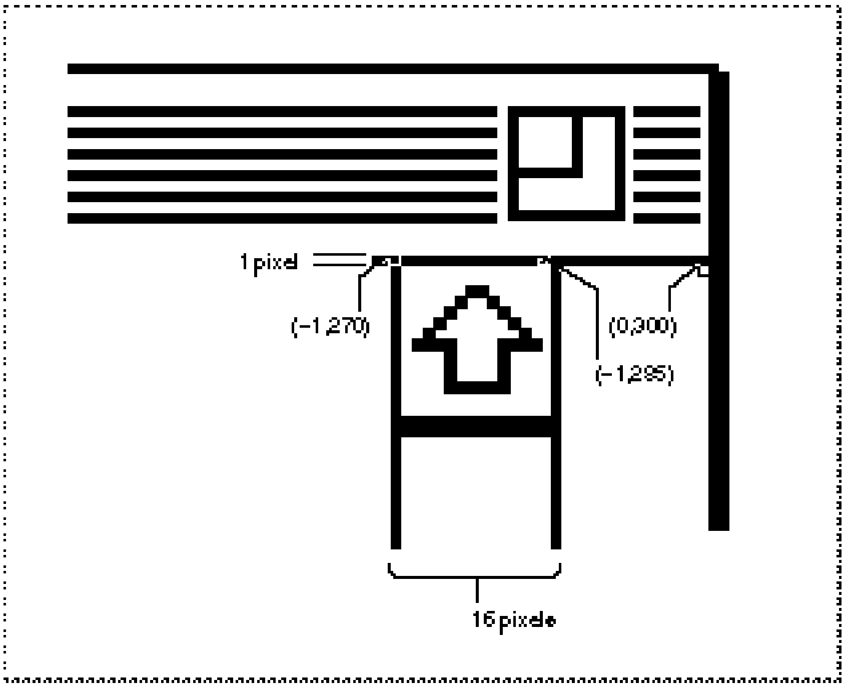
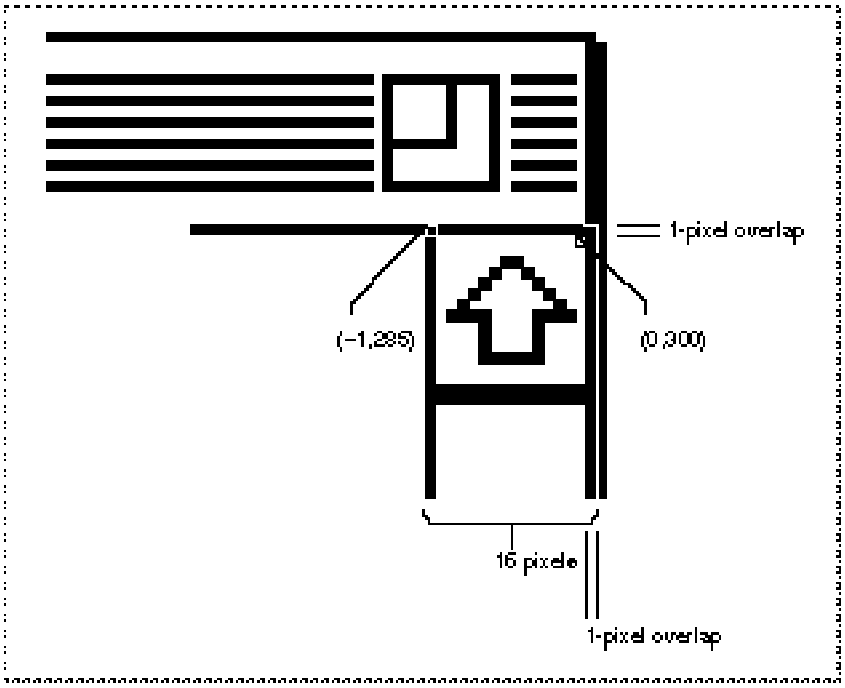


Figure 5-23 A vertical scroll bar after the application moves its upper-left point



See *Macintosh Human Interface Guidelines* for a discussion of appropriate uses of a window's scroll areas for items other than scroll bars.

To determine a new vertical coordinate for the upper-left corner of the horizontal scroll bar, `MyAdjustScrollSizes` subtracts 15 from the bottom coordinate of the window; this puts the bottom edge of the scroll bar directly over the window frame at the bottom of the window.

The `MoveControl` procedure moves the upper-left corner of a scroll bar so that it's in the proper location within its window frame. To make the vertical scroll bar fit the height of the window, and to make the horizontal scroll bar fit the width of the window, `MyAdjustScrollSizes` then uses the `SizeControl` procedure.

The `SizeControl` procedure takes three parameters: a handle to the control being sized, a width in pixels for the control, and a height in pixels for the control. When resizing a vertical scroll bar, you adjust its height; when resizing a horizontal scroll bar, you adjust its width.

When using `SizeControl` to adjust the vertical scroll bar, `MyAdjustScrollSizes` passes a constant representing 16 pixels for the vertical scroll bar's width, which is the conventional size.

To determine the proper height for this scroll bar, `MyAdjustScrollSizes` first derives the height of the window by subtracting the top coordinate of the window's rectangle from its bottom coordinate. Then `MyAdjustScrollSizes` subtracts 13 pixels from this window height and passes the result to `SizeControl` as the height of the vertical scroll bar. The `MyAdjustScrollSizes` routine subtracts 13 pixels from the window height to leave room for the 16-pixel-high size box (at the bottom of the window) minus three 1-pixel overlaps: one at the top of the window frame, one at the top of the size box, and one at the bottom of the size box.

When using `SizeControl` to adjust the horizontal scroll bar, `MyAdjustScrollSizes` passes a constant representing 16 pixels—the conventional height of the horizontal scroll bar. To determine the proper width of this scroll bar, `MyAdjustScrollSizes` first derives the width of the window by subtracting the left coordinate of the window's rectangle from its right coordinate. From this window width, `MyAdjustScrollSizes` then subtracts 13 pixels to allow for the size box (just as it does when determining the height of the vertical scroll bar).

When `MyAdjustScrollSizes` completes, it returns to `MyAdjustScrollBars`, which then uses another of its own routines, `MyAdjustScrollValues`. In turn, `MyAdjustScrollValues` calls `MyAdjustHV` (shown in Listing 5-16 on page 5-41), which recalculates the maximum settings for the scroll bars and uses `SetControlMaximum` to update the maximum settings and redraw the scroll boxes appropriately.

When `MyAdjustHV` completes, it eventually returns to the SurfWriter application's `MyAdjustScrollBars` procedure, which then uses the `ShowControl` procedure to make the newly adjusted scroll bars visible again.

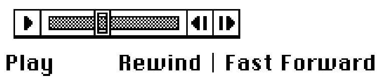
Defining Your Own Control Definition Function

The Control Manager allows you to implement controls other than the standard ones (buttons, checkboxes, radio buttons, pop-up menus, and scroll bars). To implement nonstandard controls, you must define your own control definition functions. Typically, the only types of controls you might need to implement are sliders or dials, which are similar to scroll bars in that they graphically represent a range of values the user can set. As scroll bars have scroll boxes, your sliders and dials should have indicators for setting values and indicating current settings.

Dials and sliders display the value, magnitude, or position of something, typically in some pseudo-analog form—for instance, the position of a sliding switch, the reading on a scale, or the angle of a needle on a gauge; the setting may be displayed digitally as well. The user should be able to change the control's setting by dragging its indicator.

Figure 5-24 illustrates a control supported by an application-defined control definition function. This control might be used to play back a sound or a QuickTime movie. The application might wish to define the control so that it plays the sound or movie at normal speed when the user clicks the control part on the left. The application might use the indicator along the slider to show what portion of the entire sound or movie sequence is currently playing. The application also allows the user to move quickly forward and backward through the sequence by dragging the indicator. Finally, the application might wish to define the two control parts on the far right so that they play backward (that is, "rewind") and play forward quickly (that is, "fast forward"), respectively, when the user clicks them.

Figure 5-24 A custom control



Note
When you design a dial or slider, be sure to include meaningful labels that indicate to users the range and the direction of the indicator. u

Rather than create such a control yourself, you might be tempted to use a scroll bar for this purpose. Do not do so. Using a scroll bar for any purpose other than scrolling through a window compromises the consistency of the Macintosh interface.

To define your own nonstandard control, you must write a control definition function, compile it as a resource of type `'CDEF'`, and include it in your resource file. (For more information about creating resources, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.)

When you use Control Manager routines, they in turn call your control definition function as necessary. For example, for the control in Figure 5-24 to work properly, its control definition function must be able to

- draw the control—including repositioning its indicator, making it inactive or active, and highlighting its control parts appropriately when mouse events occur in them
- determine when a mouse-down event occurs in a control part
- calculate the region of the control and its indicator
- move the indicator and update the control record with a new setting

You can also use your control definition function to modify or expand certain Control Manager behaviors; for example, you can implement your own manner of dragging an indicator, and you can perform your own type of control initialization.

For details about writing a control definition function, see "Defining Your Own Control Definition Function" beginning on page 5-109.

Control Manager Reference

This section describes the data structures, routines, and resources that are specific to the Control Manager.

The "Data Structures" section shows the data structures for the control record, the auxiliary control record, the pop-up menu private data record, and the control color table record. The "Control Manager Routines" section describes Control Manager routines for creating controls, drawing controls, handling mouse events in controls, changing control settings and display, determining control settings, and removing controls. The "Application-Defined Routines" section describes the control definition function, which you need to provide when defining your own controls. The "Application-Defined Routines" section also describes the action procedure, which defines an action to be performed repeatedly as long as the user holds down the mouse button while the cursor is in a control. The "Resources" section describes the control resource and the control color table resource.

Data Structures

This section describes the control record, the auxiliary control record, the pop-up menu private data record, and the control color table record.

Your application doesn't specifically create the control record, the auxiliary control record, or the pop-up menu private data record; rather, your application simply creates any necessary resources and uses the appropriate Control Manager routines. The Control Manager creates these records as necessary.

You can use Control Manager routines to change values in the control record, or you can access and change its fields yourself; normally, you don't change the values in the auxiliary control record. However, both the control record and the auxiliary control record have fields in which your application can store information as you deem appropriate.

You can obtain the menu handle and the menu ID of the menu associated with a pop-up menu by dereferencing the `contrlData` field of the control record, which, for pop-up menu controls, contains a handle to a pop-up private data record. This record contains the menu handle and the menu ID for the associated menu.

You use a control color table record only when you want to use nonstandard colors for a control that you create while your application is running. Your application probably shouldn't ever create a control color table record because you should use the system's default colors to ensure consistency of the interface across applications.

The Control Record

When you create a control, the Control Manager incorporates the information you specify (either in the control resource or in the parameters of the `NewControl` function) into a control record, which is a data structure of type `ControlRecord`. The Control Manager functions you use for creating a control, `GetNewControl` and `NewControl`, return a handle to a newly allocated control record. Thereafter, your application normally refers to the control by this handle, because most other Control Manager routines expect a control handle as their first parameter.

You can use Control Manager routines to determine and change several of the values in the control record, or you can access and change its fields yourself.

```
TYPE ControlRecord =
PACKED RECORD
   nextControl:   ControlHandle; {next control}
   contrlOwner:   WindowPtr;     {control's window}
   contrlRect:    Rect;          {rectangle}
   contrlVis:     Byte;          {255 if visible}
   contrlHilite:  Byte;          {highlight state}
   contrlValue:   Integer;       {control's current setting}
   contrlMin:     Integer;       {control's minimum setting}
   contrlMax:     Integer;       {control's maximum setting}
   contrlDefProc: Handle;        {control definition function}
   contrlData:    Handle;        {data used by contrlDefProc}
   contrlAction:  ProcPtr;       {action procedure}
   contrlRfCon:   LongInt;       {control's reference value}
   contrlTitle:   Str255;        {control's title}
END;
```

Field descriptions

`nextControl`	A handle to the next control associated with this control's window. All the controls belonging to a given window are kept in a linked list, beginning in the `controlList` field of the window record and chained together through the `nextControl` fields of the individual control records. The end of the list is marked by a `NIL` value; as new controls are created, they're added to the beginning of the list.
`contrlOwner`	A pointer to the window to which this control belongs.
`contrlRect`	The rectangle that completely encloses the control, in the local coordinates of the control's window. You can use the `MoveControl` and `SizeControl` procedures to change the rectangle stored in this field.
`contrlVis`	The invisible / visible state for the control. When the value of this field is 0, the Control Manager does not draw the control (its state is invisible); when the value of this field is 255, the Control Manager draws the control (its state is visible). Note that even when a control is visible, it might still be obscured from sight by an overlapping window or some other object. You can use the `HideControl` procedure to change this field from visible to invisible, and you can use the `ShowControl` procedure to change this field from invisible to visible.
`contrlHilite`	Specifies whether and how the control is to be displayed, indicating whether it's active or inactive and, if active, whether it's selected. The value of 0 signifies an active control that is not selected. A value from 1 through 253 signifies a part code designating the part of the (active) control to highlight, indicating that the user is pressing the mouse button while the cursor is in that part. The value 255 signifies that the control is to be made inactive and drawn accordingly. The `HiliteControl` procedure lets you change the value of this field.
`contrlValue`	The control's current setting. For buttons, checkboxes, and radio buttons, 0 means the control is off and 1 means it's on. For scroll bars and other sliders, `contrlValue` may take any value within the range specified in the `contrlMin` and `contrlMax` fields. For pop-up menus, this value is the item number of the menu item chosen by the user; if the user hasn't chosen a menu item, it is the item number of the first menu item. For other controls, you can use this field as you wish. You can use the `GetControlValue` function to determine the value of this field, and you can use the `SetControlValue` procedure to change the value of this field.
`contrlMin`	The control's minimum possible setting. For on-and-off controls—like checkboxes and radio buttons—this value should be 0 (meaning that the control is off). For scroll bars and other sliders, this can be any appropriate minimum value. For controls—like buttons—that don't retain a setting, this value should be 0. For pop-up menus, the Control Manager sets this field to 1. For other controls, you can use this field as you wish. You can use the `GetControlMinimum` function to determine the value of this field, and you can use the `SetControlMinimum` procedure to change the value of this field.

`contrlMax`	The control's maximum possible setting. For on-and-off controls like checkboxes and radio buttons, this value should be 1 (meaning that the control is on). For scroll bars and other sliders, this can be any appropriate maximum value. When you make the maximum setting of a scroll bar equal to its minimum setting, the control definition function automatically makes the scroll bar inactive. When you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again. For controls—like buttons—that don't retain a setting, this value should be 1. For pop-up menus, the Control Manager sets this value to the number of items in the menu. For other controls, you can use this field as you wish. You can use the `GetControlMaximum` function to determine the value of this field, and you can use the `SetControlMaximum` procedure to change the value of this field.
`contrlDefProc`	A handle to the control definition function for this type of control. When you create a control, you identify its type with a control definition ID, which is converted into a handle to the control definition function and stored in this field. Thereafter, the Control Manager uses this handle to access the definition function; you should never need to refer to this field directly. **Note** In systems running in 24-bit mode, the high-order byte of the `contrlDefProc` field contains the variant, which the Control Manager gets from the control definition ID. u
`contrlData`	Reserved for use by the control definition function, typically to hold additional information specific to a particular control type. For example, the control definition function for scroll bars uses this field for a handle to the region containing the scroll box. (If no more than 4 bytes of additional information are needed, the definition function may store the information directly in the `contrlData` field rather than using a handle.) The control definition function for pop-up menus uses this field to store a pop-up private data record, which is described on page 5-77.
`contrlAction`	A pointer to the control's action procedure, if any. The `TrackControl` function may call this procedure to respond to the user's dragging of the control, and this procedure responds by repeatedly performing some action as long as the user holds down the mouse button. See the description of `TrackControl` on page 5-90 for more information about the action procedure. You can use the `GetControlAction` function to determine the current value of this field and the `SetControlAction` procedure to change it.
`contrlRfCon`	The control's reference value, which your application may use for any purpose. You can use the `GetControlReference` function to determine the current value of this field and the `SetControlReference` procedure to change it.
`contrlTitle`	The control title, if any. You can use the `GetControlTitle` procedure to determine the current value of this field and the `SetControlTitle` procedure to change it.

The Auxiliary Control Record

For drawing all controls on systems running in 32-bit mode (which users can select using the Memory control panel), and for drawing controls that use colors other than the system default, the Control Manager creates and maintains a linked list of auxiliary control records, beginning in the global variable `AuxCtlHead`. (There is only one global list for all controls in all windows, not a separate one for each window. Each window record, by contrast, has a handle to the list of its own controls.)

An auxiliary control record is a data structure of type `AuxCtlRec`. Your application doesn't create and generally shouldn't manipulate an auxiliary control record for a control; rather, you let the Control Manager create and manipulate the auxiliary control record. To create controls using colors other than the system default colors, use the `SetControlColor` procedure (described on page 5-101) or create a control color table resource (described on page 5-121) and let the Control Manager create the necessary auxiliary control records. There is, however, a field in the auxiliary control record that you can use to store information as you see fit; to get a handle to the auxiliary control record for a control, you can use the `GetAuxiliaryControlRecord` function (described on page 5-107).

Each auxiliary control record is relocatable and resides in your application heap. Here is how an auxiliary control record is defined:

```
TYPE AuxCtlRec =
RECORD
   acNext:        AuxCtlHandle;  {handle to next AuxCtlRec}
   acOwner:       ControlHandle; {handle to this record's control}
   acCTable:      CCTabHandle;   {handle to control color table }
                                 { record}
   acFlags:       Integer;       {reserved}
   acReserved:    LongInt;       {reserved for future use}
   acRefCon:      LongInt;       {for use by application}
END;
```

Field descriptions

`acNext`	A handle to the next record in the auxiliary control list.
`acOwner`	The handle of the control to which this auxiliary record belongs; used as an ID field.
`acCTable`	The handle to a control color table record. (The control color table record is described on page 5-77.)
`acFlags`	Reserved for use by the Control Manager.
`acReserved`	Reserved for future expansion.
`acRefCon`	A reference value, which your application may use for any purpose.

On systems using 32-bit mode, every control has its own auxiliary record, and the `acCTable` field contains a handle to the default control color table unless your application uses the `SetControlColor` procedure or creates a control color table resource.

When drawing a control, the standard control definition functions search the linked list of auxiliary control records for the auxiliary control record whose acOwner field points to the control being drawn. If the standard control definition functions find an auxiliary control record for the control, they use the control color table specified in the acCTable field. If the standard control definition functions do not find an auxiliary control record for the control, they use the default system colors.

The Pop-Up Menu Private Data Record

You can obtain the menu handle and the menu ID of the menu associated with a pop-up menu by dereferencing the contrlData field of the pop-up menu's control record. The contrlData field of a control record is a handle to a block of private information. For pop-up menu controls, this field is a handle to a pop-up private data record, which is a data structure of type popupPrivateData.

```
TYPE  popupPrivateData =
      RECORD
         mHandle:    MenuHandle;       {handle to menu record}
         mID:        Integer;          {menu ID}
         mPrivate:   ARRAY[0..0] OF SignedByte; {reserved}
      END;
```

Field descriptions

mHandle	Contains a handle to the menu.
mID	The menu ID of the menu.
mPrivate	Reserved.

You can use the standard pop-up control definition function to manage pop-up menus. For information on creating pop-up menus, see "Creating a Pop-Up Menu" beginning on page 5-25. See the chapter "Menu Manager" in this book for additional information.

The Control Color Table Record

By creating a control color table record and using the SetControlColor procedure (described on page 5-101), your application can draw a control that uses colors other than the system default. (Alternatively, you can use nonstandard colors for a control you define in a control resource by creating a control color table resource—described on page 5-121—with the same resource ID as the control resource.) Be aware that controls in nonstandard colors may initially confuse your users.

A control color table record is a data structure of type CtlCTab; it is defined as follows:

```
TYPE CtlCTab =
RECORD
   ccSeed:     LongInt;    {reserved; set to 0}
   ccRider:    Integer;    {reserved; set to 0}
   ctSize:     Integer;    {number of ColorSpec records in next }
                           { field; 3 for standard controls}
   ctTable:    ARRAY[0..3] OF ColorSpec;
END;
```

Field descriptions

`ccSeed`	Reserved in control color tables; set to 0.
`ccRider`	Reserved in control color tables; set to 0.
`ctSize`	The number of `ColorSpec` records in the next field. For controls drawn with the standard definition procedure, this field is always 3, because a standard control has three parts: frame, control body, and scroll box for scroll bars, and frame, control body, and text for other controls. If you want to supply `ColorSpec` records for additional parts, you must define your own controls, as described in "Defining Your Own Control Definition Function" beginning on page 5-109.
`ctTable`	An array of `ColorSpec` records. Each `ColorSpec` record describes the color of a different control part. Here is how a `ColorSpec` record is defined:

```
TYPE ColorSpec =
RECORD
   partIdentifier:   Integer;    {control part}
   partRGB:          RGBColor;   {color of part}
END;
```

The `partIdentifier` field of the `ColorSpec` record holds an integer that associates an `RGBColor` record with a particular part of the control.

Three `ColorSpec` records are used to describe the parts of buttons, checkboxes, and radio buttons. Here are the constants that are used in the `partIdentifier` fields of the three `ColorSpec` records used to describe these controls:

```
{for buttons, checkboxes, and radio buttons}
CONST cFrameColor = 0;  {frame color}
      cBodyColor = 1;   {fill color for body of }
                        { control}
      cTextColor = 2;   {text color}
```

When highlighted, buttons exchange their body and text colors; checkboxes and radio buttons change their appearance without changing colors. All three types indicate deactivation by dimming their text with no change in colors.

A number of ColorSpec records are used to describe the parts of scroll bars. Here are the constants that are used in the partIdentifier fields of the ColorSpec records used to describe the colors in scroll bars:

```
CONST
cFrameColor       = 0;  {Used to produce foreground color for scroll arrows }
                        { & gray area}
cBodyColor        = 1;  {Used to produce colors in the scroll box}
cArrowsColorLight = 5;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cArrowsColorDark  = 6;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cThumbLight       = 7;  {Used to produce colors in scroll box}
cThumbDark        = 8;  {Used to produce colors in scroll box}
cHiliteLight      = 9;  {Use same value as wHiliteColorLight in 'wctb'}
cHiliteDark       = 10; {Use same value as wHiliteColorDark in 'wctb'}
cTitleBarLight    = 11; {Use same value as wTitleBarLight in 'wctb'}
cTitleBarDark     = 12; {Use same value as wTitleBarDark in 'wctb'}
cTingeLight       = 13; {Use same value as wTingeLight in 'wctb'}
cTingeDark        = 14; {Use same value as wTingeDark in 'wctb'}
```

When highlighted, scroll arrows are filled with the foreground color. A deactivated scroll bar shows no scroll box and displays its gray areas in a solid background color with no pattern.

The ColorSpec records for a control can appear in any order. If you include a part identifier that is not found, the Control Manager uses the first ColorSpec record with an identifiable part. If you do not specify a part identifier, the Control Manager uses the default color for that part.

The partRGB field of the ColorSpec record specifies an RGBColor record, which in turn specifies the red, green, and blue values for the part's color. Use three 16-bit unsigned integers to give the intensity values for the three additive primary colors. Here is how the RGBColor record is defined:

```
TYPE RGBColor =
RECORD
   red:     Integer; {red value for control part}
   green:   Integer; {green value for control part}
   blue:    Integer; {blue value for control part}
END;
```

When you create a control color table record, your application should not deallocate it if another control is still using it.

When drawing a control, the standard control definition functions search the linked list of auxiliary control records for the record whose acOwner field points to that control. If a standard control definition function finds such a record, it uses the color table designated by that record; otherwise, it uses the default system colors. Each control using colors other than the system default has its own auxiliary control record, even if that control uses the same control color table record as another control; two or more auxiliary records can share the same control color table record. (Auxiliary control records are described on page 5-76.)

If you create a control definition function (as explained in "Defining Your Own Control Definition Function" beginning page 5-109), you can use color tables of any desired size and define their contents in any way you wish, except that part indices 1 through 127 are reserved for system definition. Any such nonstandard control definition function should bypass the defaulting mechanism by allocating an explicit auxiliary record for every control it creates.

Control Manager Routines

This section describes the Control Manager routines for creating controls, drawing controls, tracking mouse events within controls, changing control display, determining control values, and removing controls.

Some Control Manager routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, `SetControlValue` is also available as `SetCtlValue`. Table 5-1 provides a mapping between the previous name of a routine and its new equivalent name.

Table 5-1 Mapping between new and previous names of Control Manager routines

New name	Previous name
`GetAuxiliaryControlRecord`	`GetAuxCtl`
`GetControlAction`	`GetCtlAction`
`GetControlMaximum`	`GetCtlMax`
`GetControlMinimum`	`GetCtlMin`
`GetControlReference`	`GetCRefCon`
`GetControlTitle`	`GetCTitle`
`GetControlValue`	`GetCtlValue`
`GetControlVariant`	`GetCVariant`
`SetControlAction`	`SetCtlAction`
`SetControlColor`	`SetCtlColor`
`SetControlMaximum`	`SetCtlMax`
`SetControlMinimum`	`SetCtlMin`
`SetControlReference`	`SetCRefCon`
`SetControlTitle`	`SetCTitle`
`UpdateControls`	`UpdtControl`

Creating Controls

To create a control, you should generally use the `GetNewControl` function, which takes information about the control from a control resource. Like menu resources, control resources isolate descriptive information from your application code, making your application easier to modify or translate. However, you can also use the `NewControl` function—for which you pass descriptive information in parameters—to create controls.

Both `GetNewControl` and `NewControl` return a handle to the control record of the newly created control. Thereafter, your application normally refers to the control by this handle, because most other Control Manager routines expect a control handle as their first parameter. When you create scroll bars and pop-up menus, you should store their handles in one of your application's own data structures for later reference.

When you use the Dialog Manager to implement buttons, radio buttons, checkboxes, and pop-up menus in alert boxes and dialog boxes, the Dialog Manager automatically uses the Control Manager to create these controls for you. If you implement other controls in alert or dialog boxes, and whenever you implement controls—such as scroll bars—in your application's windows, you must use either `GetNewControl` or `NewControl` to create these controls.

GetNewControl

To create a control from a description in a control resource (`'CNTL'`), use the `GetNewControl` function.

```
FUNCTION GetNewControl (controlID: Integer; owner: WindowPtr)
                        : ControlHandle;
```

`controlID`	The resource ID of a control resource.
`owner`	A pointer to the window in which you want to attach the control.

DESCRIPTION

The `GetNewControl` function creates a control record from the information in the specified control resource, adds the control record to the control list for the specified window, and returns as its function result a handle to the control. You use this handle when referring to the control in most other Control Manager routines. After making a copy of the control resource, `GetNewControl` releases the memory occupied by the original control resource before returning.

If you provide a control color table resource with the same resource ID as the control resource, `GetNewControl` creates an auxiliary control record that uses the colors you specify in your control color table resource. If you don't provide a control color table, `GetNewControl` creates an auxiliary control record that uses the default control color table if the computer is running in 32-bit mode.

The control resource specifies the rectangle for the control, its initial setting, its visibility state, its maximum and minimum settings, its control definition ID, a reference value, and its title (if any). After you use `GetNewControl` to create the control, you can change the current setting, the maximum setting, the minimum setting, the reference value, and the title by using, respectively, the `SetControlValue`, `SetControlMaximum`, `SetControlMinimum`, `SetControlReference`, and `SetControlTitle` procedures. You can use the `MoveControl` and `SizeControl` procedures to change the control's rectangle. You can use the `GetControlValue`, `GetControlMaximum`, `GetControlMinimum`, `GetControlReference`, and `GetControlTitle` functions to determine the control values.

If the control resource specifies that the control should be visible, the Control Manager draws the control. If the control resource specifies that the control should initially be invisible, you can use the `ShowControl` procedure to make the control visible.

If `GetNewControl` can't read the control resource from the resource file, `GetNewControl` returns `NIL`.

SEE ALSO

See Listing 5-1 on page 5-17 and Listing 5-5 on page 5-24 for examples of how to use `GetNewControl` to create, respectively, a button and a scroll bar. For information about windows' control lists, see the chapter "Window Manager" in this book.

NewControl

To create a control, you can use the `NewControl` function, which accepts in its parameters the information that describes the control. Generally, you should instead use the `GetNewControl` function to create a control. The `GetNewControl` function takes information about the control from a control resource, and as a result your application is easier to modify or translate into other languages.

```
FUNCTION NewControl (theWindow: WindowPtr; boundsRect: Rect;
                     title: Str255; visible: Boolean;
                     value: Integer; min: Integer; max: Integer;
                     procID: Integer; refCon: LongInt)
                     : ControlHandle;
```

`theWindow`	A pointer to the window in which you want to attach the control. All coordinates pertaining to the control are interpreted in this window's local coordinate system.
`boundsRect`	The rectangle, specified in the given window's local coordinates, that encloses the control and thus determines its size and location.

`title`	For controls that need a title—such as buttons, checkboxes, radio buttons, and pop-up menus—the string for that title. For controls that don't use titles, pass an empty string.
`visible`	The visible/invisible state for the control. If you pass `TRUE` in this parameter, `NewControl` draws the control immediately, without using your window's standard updating mechanism. If you pass `FALSE`, you must later use the `ShowControl` procedure to display the control.
`value`	The initial setting for the control. For controls—such as buttons—that don't retain a setting, pass 0 in this parameter. For controls—such as checkboxes and radio buttons—that retain an on-or-off setting, pass 0 in this parameter for a control that is off, and pass 1 for a control that is on. For controls—such as scroll bars and sliders—that can take a range of settings, specify whatever value is appropriate within that range.
`min`	The minimum setting for the control. For controls—such as buttons—that don't retain a setting, pass 0 in this parameter. For controls—such as checkboxes and radio buttons—that retain an on-or-off setting, use 0 (meaning "off") for the minimum value. For controls—such as scroll bars and sliders—that can take a range of settings, specify whatever minimum value is appropriate.
`max`	The maximum setting for the control. For controls—such as buttons—that don't retain a setting, pass 1 in this parameter. For controls—such as checkboxes and radio buttons—that retain an on-or-off setting, use 1 (meaning "on") for the maximum value. For controls—such as scroll bars and sliders—that can take a range of settings, specify whatever maximum value is appropriate. When you make the maximum setting of a scroll bar equal to its minimum setting, the control definition function automatically makes the scroll bar inactive; when you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.
`procID`	The control definition ID, which leads to the control definition function for this type of control. The control definition function is read into memory if it isn't already in memory. The control definition IDs and their constants for the standard controls are listed here. (You can also define your own control definition function and specify it the `procID` parameter.)

```
CONST
   pushButProc        = 0;     {button}
   checkBoxProc       = 1;     {checkbox}
   radioButProc       = 2;     {radio button}
   useWFont           = 8;     {add to above to display }
                               { title in window font}
   scrollBarProc      = 16;    {scroll bar}
   popupMenuProc      = 1008;  {pop-up menu}
   popupFixedWidth    = $0001; {add to popupMenuProc to }
                               { use a fixed-width ctrl}
```

```
popupUseAddResMenu = $0004; {add to popupMenuProc to }
                            { specify a value of }
                            { type ResType in the }
                            { contrlRfCon field of }
                            { the control record; }
                            { Menu Manager adds }
                            { resources of this type }
                            { to the menu}
popupUseWFont      = $0008; {add to popupMenuProc to }
                            { display in window font}
```

`refCon`	The control's reference value, which is set and used only by your application.

DESCRIPTION

The `NewControl` function creates a control record from the information you specify in its parameters, adds the control record to the control list for the specified window, and returns as its function result a handle to the control. You use this handle when referring to the control in most other Control Manager routines.

The `NewControl` function creates an auxiliary control record that uses the default control color table if the computer is running in 32-bit mode.

If you need to use colors other than the default colors for the control, create a control color table record and use the `SetControlColor` procedure.

When specifying the rectangle in the `boundsRect` parameter, keep the following guidelines in mind:

- Buttons are drawn to fit the rectangle exactly. To accommodate the tallest characters in the system font, allow at least a 20-point difference between the top and bottom coordinates of the rectangle.
- For checkboxes and radio buttons, there should be at least a 16-point difference between the top and bottom coordinates.
- By convention, scroll bars are 16 pixels wide, so there should be a 16-point difference between the left and right (or top and bottom) coordinates. (If there isn't, the scroll bar is scaled to fit the rectangle.) A standard scroll bar should be at least 48 pixels long, to allow room for the scroll arrows and scroll box.

The Control Manager displays control titles in the system font. When specifying a title for the control in the `title` parameter, make sure the title fits in the control's rectangle; otherwise, `NewControl` truncates the title. For example, `NewControl` truncates the titles of checkboxes and radio buttons on the right in Roman scripts, and it centers and truncates both ends of the button titles.

The Control Manager allows multiple lines of text in the titles of buttons, checkboxes, and radio buttons. When specifying a multiple-line title, separate the lines with the ASCII character code $0D (carriage return). If the control is a button, each line is horizontally centered, and the font leading is inserted between lines. (The height of each line is equal to the distance from the ascent line to the descent line plus the leading of the font used. Be sure to make the total height of the rectangle greater than the number of lines times this height.) If the control is a checkbox or a radio button, the text is justified as appropriate for the user's current script system, and the checkbox or button is vertically centered within its rectangle.

After you use `NewControl` to create the control, you can change the current setting, the maximum setting, the minimum setting, the reference value, and the title by using, respectively, the `SetControlValue`, `SetControlMaximum`, `SetControlMinimum`, `SetControlReference`, and `SetControlTitle` procedures. You can use the `MoveControl` and `SizeControl` procedures to change the control's rectangle. You can use the `GetControlValue`, `GetControlMaximum`, `GetControlMinimum`, `GetControlReference`, and `GetControlTitle` functions to determine the control values.

SPECIAL CONSIDERATIONS

The title of a button, checkbox, radio button, or pop-up menu normally appears in the system font, which in Roman script systems is 12-point Chicago. Do not use a smaller font; some script systems, such as KanjiTalk, require 12-point fonts. You should generally use the system font in your controls; doing so will simplify localization effort. However, if you absolutely need to display a control title in the font currently associated with the window's graphics port, you can add the `popupUseWFont` constant to the pop-up menu control definition ID or add the `useWFont` constant to the other standard control definition IDs.

SEE ALSO

For information about windows' control lists, see the chapter "Window Manager" in this book. Control definition IDs for other controls are discussed in "Defining Your Own Control Definition Function" beginning on page 5-109.

Drawing Controls

If you specify that a control is initially visible (either in the control resource or in a parameter to `NewControl`), the Control Manager draws the control inside its window when you call either the `GetNewControl` or the `NewControl` function. In either case, the Control Manager draws the control immediately, without using your window's standard updating mechanism. If you specify that a control is invisible, you can use the `ShowControl` procedure when you want to draw the control.

Note that even a visible control might be completely or partially obscured by overlapping windows or other objects.

When your application receives an update event for a window that contains controls, use `UpdateControls` to redraw the necessary controls in the updated window. Note that the Dialog Manager automatically draws and updates controls in alert boxes and dialog boxes.

ShowControl

To draw a control that is currently invisible, you can use the `ShowControl` procedure.

```
PROCEDURE ShowControl (theControl: ControlHandle);
```

`theControl`	A handle to the control you want to make visible.

DESCRIPTION

If the specified control is invisible, the `ShowControl` procedure makes it visible and immediately draws the control within its window without using your window's standard updating mechanism. If the control is already visible, `ShowControl` has no effect.

You can make a control invisible in several ways:

- You can specify that it's invisible in its control resource.
- You can specify that it's invisible in a parameter to the `NewControl` function.
- You can use the `HideControl` procedure to change a visible control into an invisible one.
- You can directly change the `contrlVis` field of the control's control record.

SPECIAL CONSIDERATIONS

The `ShowControl` procedure draws the control in its window, but the control can still be completely or partially obscured by overlapping windows or other objects.

SEE ASO

Listing 5-14 on page 5-39 illustrates the use of `ShowControl` to redisplay scroll bars after moving and resizing them.

UpdateControls

To update controls in a window, you can use the `UpdateControls` procedure. The `UpdateControls` procedure is also available as the `UpdtControl` procedure.

```
PROCEDURE UpdateControls (theWindow: WindowPtr;
                          updateRgn: RgnHandle);
```

`theWindow`	A pointer to the window containing the controls to update.
`updateRgn`	The update region within the specified window.

DESCRIPTION

The `UpdateControls` procedure draws those controls that are in the specified update region. This procedure is faster than the `DrawControls` procedure, which draws *all* of the controls in a window. By contrast, `UpdateControls` draws only those controls in the update region.

Your application should call `UpdateControls` upon receiving an update event for a window that contains controls. Window Manager routines such as `SelectWindow`, `ShowWindow`, and `BringToFront` do not automatically call `DrawControls` to display the window's controls. They just add the appropriate regions to the window's update region, generating an update event.

In response to an update event, you normally call `UpdateControls` after using the Window Manager procedure `BeginUpdate` and before using the Window Manager procedure `EndUpdate`. You should set the `updateRgn` parameter to the visible region of the window's port, as specified in the port's `visRgn` field.

SPECIAL CONSIDERATIONS

If your application draws parts of a control outside of its rectangle, `UpdateControls` might not redraw it.

The Dialog Manager handles update events for controls in alert boxes and dialog boxes.

SEE ALSO

Listing 5-8 on page 5-30 illustrates the use of `UpdateControls`. The `BeginUpdate` and `EndUpdate` procedures are described in the chapter "Window Manager" in this book. See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes.

DrawControls

Although you should generally use the `UpdateControls` procedure to update controls in a window, you can instead use the `DrawControls` procedure.

```
PROCEDURE DrawControls (theWindow: WindowPtr);
```

`theWindow`	A pointer to a window whose controls you want to display.

DESCRIPTION

The `DrawControls` procedure draws *all* controls currently visible in the specified window. The controls are drawn in reverse order of creation; thus, in case of overlapping controls, the control created first appears frontmost in the window.

Because the `UpdateControls` procedure redraws only those controls that need updating, your application should generally use it instead of `DrawControls` upon receiving an update event for a window that contains controls.

You should call either `DrawControls` or `UpdateControls` after calling the Window Manager procedure `BeginUpdate` and before calling `EndUpdate`.

SPECIAL CONSIDERATIONS

The Dialog Manager automatically draws and updates controls in alert boxes and dialog boxes.

Window Manager routines such as `SelectWindow`, `ShowWindow`, and `BringToFront` do not automatically update the window's controls. They just add the appropriate regions to the window's update region, generating an update event.

SEE ALSO

See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes. See the chapter "Window Manager" in this book for more information about Window Manager routines.

Draw1Control

Although you should generally use the `UpdateControls` procedure to update controls, you can use the `Draw1Control` procedure to update a single control.

```
PROCEDURE Draw1Control (theControl: ControlHandle);
```

`theControl` A handle to the control you want to draw.

DESCRIPTION

The `Draw1Control` procedure draws the specified control if it's visible within its window. The `UpdateControls` procedure automatically calls `Draw1Control`.

Handling Mouse Events in Controls

When the user presses the mouse button, your application receives a mouse-down event. Use the Window Manager function `FindWindow` to determine which window contains the cursor. If the mouse-down event occurred in the content region of your application's active window, use the `FindControl` function to determine whether the cursor was in an active control and, if so, which control. To follow and respond to the cursor movements in that control, and then to determine in which part of the control the mouse-up event occurs, use the `TrackControl` function.

FindControl

To determine whether a mouse-down event has occurred in a control and, if so, in which part of that control, use the `FindControl` function.

```
FUNCTION FindControl (thePoint: Point; theWindow: WindowPtr;
                      VAR theControl: ControlHandle): Integer;
```

`thePoint`	A point, specified in coordinates local to the window, where the mouse-down event occurred.
`theWindow`	A pointer to the window in which the mouse-down event occurred.
`theControl`	A handle to the control in which the mouse-down event occurred.

DESCRIPTION

When the user presses the mouse button while the cursor is in a visible, active control, `FindControl` returns as its function result a part code identifying the control's part; the function also returns a handle to the control in the parameter `theControl`. The part codes that `FindControl` returns, and the constants you can use to represent them, are listed here:

```
CONST inButton       = 10;    {button}
      inCheckBox     = 11;    {checkbox or radio button}
      inUpButton     = 20;    {up arrow for a vertical scroll }
                              { bar, left arrow for a horizontal }
                              { scroll bar}
      inDownButton   = 21;    {down arrow for a vertical scroll }
                              { bar, right arrow for a }
                              { horizontal scroll bar}
      inPageUp       = 22;    {gray area above scroll box for a }
                              { vertical scroll bar, gray area }
                              { to left of scroll box for a }
                              { horizontal scroll bar}
      inPageDown     = 23;    {gray area below scroll box for a }
                              { vertical scroll bar, gray area }
                              { to right of scroll box for a }
                              { horizontal scroll bar}
      inThumb        = 129;   {scroll box}
```

The pop-up control definition function does not define part codes for pop-up menus. Instead, your application should store the handles for your pop-up menus when you create them. Your application should then test the handles you store against the handles returned by `FindControl` before responding to users' choices in pop-up menus.

If the mouse-down event occurs in an invisible or inactive control, or if it occurs outside a control, `FindControl` sets `theControl` to `NIL` and returns 0 as its function result.

When a mouse-down event occurs, your application should call `FindControl` after using the Window Manager function `FindWindow` to ascertain that a mouse-down event has occurred in the content region of a window containing controls.

Before calling `FindControl`, use the `GlobalToLocal` procedure to convert the point stored in the `where` field (which describes the location of the mouse-down event) of the event record to coordinates local to the window. Then, when using `FindControl`, pass this point in the parameter `thePoint`.

In the parameter `theWindow`, pass the window pointer returned by the `FindWindow` function.

After using `FindControl` to determine that a mouse-down event has occurred in a control, you generally use the `TrackControl` function, which automatically follows the movements of the cursor and responds as appropriate until the user releases the mouse button.

SPECIAL CONSIDERATIONS

The Dialog Manager automatically calls `FindControl` and `TrackControl` for mouse-down events inside controls of alert boxes and dialog boxes.

The `FindControl` function also returns `NIL` in the parameter `theControl` and 0 as its function result if the window is invisible or if it doesn't contain the given point. (However, `FindWindow` won't return a window pointer to an invisible window or to one that doesn't contain the point where the mouse-down event occurred. As long as you call `FindWindow` before `FindControl`, this situation won't arise.)

SEE ALSO

Listing 5-10 on page 5-33 illustrates the use of `FindControl` for detecting mouse-down events in a pop-up menu and a button; Listing 5-18 on page 5-53 illustrates its use for detecting mouse-down events in scroll bars.

The `FindWindow` function is described in the chapter "Window Manager" in this book. The `GlobalToLocal` procedure is described in *Inside Macintosh: Imaging*.

The event record is described in the chapter "Event Manager" in this book. See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes.

TrackControl

To follow and respond to cursor movements in a control and then to determine the control part in which the mouse-up event occurs, use the `TrackControl` function.

```
FUNCTION TrackControl (theControl: ControlHandle;
                       thePoint: Point; actionProc: ProcPtr)
                       : Integer;
```

`theControl`	A handle to the control in which a mouse-down event occurred.
`thePoint`	A point, specified in coordinates local to the window, where the mouse-down event occurred.
`actionProc`	The action procedure. Typically, you should set this parameter to `NIL` for buttons, checkboxes, radio buttons, and the scroll box of a scroll bar; set this parameter to `Pointer(-1)` for pop-up menus; and set this parameter to the pointer to an action procedure for scroll arrows and gray areas of scroll bars, as well as for any other controls that require you to define additional actions to take while the user holds down the mouse button.

DESCRIPTION

The `TrackControl` function follows the user's cursor movements in a control and provides visual feedback until the user releases the mouse button. The visual feedback given by `TrackControl` depends on the control part in which the mouse-down event occurs. When highlighting is appropriate, for example, `TrackControl` highlights the control part (and removes the highlighting when the user releases the mouse button). When the user holds down the mouse button while the cursor is in an indicator (such as the scroll box of a scroll bar) and moves the mouse, `TrackControl` responds by dragging a dotted outline of the indicator.

The `TrackControl` function returns as its function result the control's part code if the user releases the mouse button while the cursor is inside the control part, or 0 if the user releases the mouse button while the cursor is outside the control part. For control parts, the `TrackControl` function returns the same values (represented by the constants `inButton`, `inCheckBox`, `inUpButton`, `inDownButton`, `inPageUp`, `inPageDown`, and `inThumb`) returned by the `FindControl` function, as described on page 5-89.

When `TrackControl` returns a value other than 0 as its function result, your application should respond as appropriate to a mouse-up event in that control part. When `TrackControl` returns 0 as its function result, your application should do nothing.

If the user releases the mouse button when the cursor is in an indicator such as a scroll box, `TrackControl` calls the control's control definition function to reposition the indicator. The control definition function for scroll bars, for example, responds to the user dragging a scroll box by redrawing the scroll box, calculating the control's current setting according to the new relative position of the scroll box, and storing the current setting in the control record. Thus, if the minimum and maximum settings are 0 and 10, and the scroll box is in the middle of the scroll bar, 5 is stored as the current setting. For a scroll bar, your application must then respond by scrolling to the corresponding relative position in the document.

Generally, you use `TrackControl` after using the `FindControl` function. In the parameter `theControl` of `TrackControl`, pass the control handle returned by the `FindControl` function, and in the parameter `thePoint`, supply the same point you passed to `FindControl` (that is, a point in coordinates local to the window).

While the user holds down the mouse button with the cursor in one of the standard controls, `TrackControl` performs the following actions, depending on the value you pass in the parameter `actionProc`. (For other controls, what you pass in this parameter depends on how you define the control.)

- If you pass `NIL` in the `actionProc` parameter, `TrackControl` uses no action procedure and therefore performs no additional actions beyond highlighting the control or dragging the indicator. This is appropriate for buttons, checkboxes, radio buttons, and the scroll box of a scroll bar.
- If you pass a pointer to an action procedure in the `actionProc` parameter, you must provide the procedure, and it must define some action that your application repeats as long as the user holds down the mouse button. This is appropriate for the scroll arrows and gray areas of a scroll bar.
- If you pass `Pointer(-1)` in the `actionProc` parameter, `TrackControl` looks in the `contrlAction` field of the control record for a pointer to the control's action procedure. This is appropriate when you are tracking the cursor in a pop-up menu. (You can use the `GetControlAction` function to determine the value of this field, and you can use the `SetControlAction` procedure to change this value.) If the `contrlAction` field of the control record contains a procedure pointer, `TrackControl` uses the action procedure it points to; if the field of the control record also contains the value `Pointer(-1)`, `TrackControl` calls the control's control definition function to perform the necessary action; you may wish to do this if you define your own control definition function for a custom control. If the field of the control record contains the value `NIL`, `TrackControl` performs no action.

SPECIAL CONSIDERATIONS

When you need to handle events in alert and dialog boxes, Dialog Manager routines automatically call `FindControl` and `TrackControl`.

ASSEMBLY-LANGUAGE INFORMATION

The `TrackControl` function invokes the Window Manager function `DragGrayRgn`, so you can use the global variables `DragHook` and `DragPattern`.

SEE ALSO

See "Defining Your Own Action Procedures" beginning on page 5-115 for information about an action procedure to specify in the `actionProc` parameter. See "Defining Your Own Control Definition Function" beginning on page 5-109 for information about creating a control definition function.

Listing 5-11 on page 5-36, Listing 5-12 on page 5-37, Listing 5-13 on page 5-38, and Listing 5-18 on page 5-53 illustrate the use of `TrackControl` for responding to mouse-down events in, respectively, a button, a pop-up menu, a checkbox, and a scroll bar.

See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes.

TestControl

The `TestControl` function is called by the `FindControl` and `TrackControl` functions—normally you won't need to call it yourself. However, should you ever need to determine the control part in which a mouse-down event occurred, you can use the `TestControl` function.

```
FUNCTION TestControl (theControl: ControlHandle; thePt: Point)
                      : Integer;
```

`theControl`	A handle to the control in which the mouse-down event occurred.
`thePt`	The point, in a window's local coordinates, where the mouse-down event occurred.

DESCRIPTION

When the control specified by the parameter `theControl` is visible and active, `TestControl` tests which part of the control contains the point specified by the parameter `thePt`. For its function result, `TestControl` returns the part code of the control part, or 0 if the point is outside the control.

If the control is invisible or inactive, `TestControl` returns 0.

Changing Control Settings and Display

In response to user actions, you often need to change the settings, highlight states, sizes, and locations of your controls. Whenever your application calls the `TrackControl` function, the Control Manager automatically manipulates control display as appropriate as the user presses and releases the mouse button. For example, `TrackControl` calls the `HiliteControl` procedure to highlight buttons; for scroll bars, `TrackControl` calls the `DragControl` procedure to move an outline of the scroll box in a scroll bar and the `SetControlValue` procedure to change the scroll bar's current setting and redraw the scroll box in its new location. (Note that the Dialog Manager automatically calls `TrackControl` for controls in alert boxes and dialog boxes. See the chapter "Dialog Manager" in this book for more information.)

When the user releases the mouse button while the cursor is in a control, your application often needs to change its setting. When the user clicks a checkbox, for example, your application must change its setting to on or off, and the Control Manager automatically draws or removes an X in the checkbox.

There are other instances when you must change the settings and display of a control. For example, when the user changes the size of a window that contains a scroll bar, you need to resize and move the scroll bar accordingly.

For controls whose values the user can set, you can use the `SetControlValue` procedure to change the control's setting and redraw the control accordingly. When you need to change the maximum setting of a scroll bar or a dial, you can use the

SetControlMaximum procedure; if you need to change the minimum setting, you can use the SetControlMinimum procedure. If you need to change a control title, you can use the SetControlTitle procedure. You can use the HideControl procedure to make a control invisible. When you need to make a control inactive (such as when its window is not frontmost) or in any other way change the highlighting of a control, you can use the HiliteControl procedure.

To move a scroll bar, you use the MoveControl and SizeControl procedures.

Although it's not recommended, you can also change a control's default colors to those of your own choosing by using the SetControlColor procedure.

To invoke a continuous action while the user holds down the mouse button, you can specify an action procedure (described in "Defining Your Own Action Procedures" beginning on page 5-115) in a parameter to TrackControl. Under certain circumstances, you can use the SetControlAction procedure to change the control's action procedure, though you should rarely if ever need to.

SetControlValue

To change the current setting of a control and redraw it accordingly, you can use the SetControlValue procedure. The SetControlValue procedure is also available as the SetCtlValue procedure.

```
PROCEDURE SetControlValue (theControl: ControlHandle;
                           theValue: Integer);
```

theControl	A handle to the control whose current setting you wish to change.
theValue	The new setting for the control.

DESCRIPTION

The SetControlValue procedure changes the contrlValue field of the control record to the specified value and redraws the control to reflect the new setting. For checkboxes and radio buttons, the value 1 fills the control with the appropriate mark, and 0 removes the mark. For scroll bars, SetControlValue redraws the scroll box where appropriate.

If the specified value is less than the minimum setting for the control, SetControlValue sets the control to its minimum setting; if the value is greater than the maximum setting, SetControlValue sets the control to its maximum.

When you create a control, you specify an initial setting either in the control resource or in the value parameter of the NewControl function. To determine a control's current setting before changing it in response to a user's click in that control, use the GetControlValue function.

SEE ALSO

Listing 5-13 on page 5-38 illustrates the use of `SetControlValue` to change the setting of a checkbox. Listing 5-16 on page 5-41 and Listing 5-20 on page 5-61 illustrate the use of `SetControlValue` to change the setting of a scroll bar.

SetControlMinimum

To change the minimum setting of a control and redraw its indicator or scroll box accordingly, you can use the `SetControlMinimum` procedure. The `SetControlMinimum` procedure is also available as the `SetCtlMin` procedure.

```
PROCEDURE SetControlMinimum (theControl: ControlHandle;
                             minValue: Integer);
```

`theControl`	A handle to the control whose minimum setting you wish to change.
`minValue`	The new minimum setting.

DESCRIPTION

The `SetControlMinimum` procedure changes the `contrlMin` field of the control record to the setting you specify in the `minValue` parameter and redraws its indicator or scroll box to reflect its new range.

When you create a control, you specify an initial minimum setting either in the control resource or in the `min` parameter of the `NewControl` function. To determine a control's current minimum setting, use the `GetControlMinimum` function.

SetControlMaximum

To change the maximum setting of a control and redraw its indicator or scroll box accordingly, you can use the `SetControlMaximum` procedure. The `SetControlMaximum` procedure is also available as the `SetCtlMax` procedure.

```
PROCEDURE SetControlMaximum (theControl: ControlHandle;
                             maxValue: Integer);
```

`theControl`	A handle to the control whose maximum setting you wish to change.
`maxValue`	The new maximum setting.

DESCRIPTION

The `SetControlMaximum` procedure changes the `contrlMax` field of the control record to the setting you specify in the `maxValue` parameter and redraws its indicator or scroll box to reflect its new range.

When you create a control, you specify an initial maximum setting either in the control resource or in the `max` parameter of the `NewControl` function. To determine a control's current maximum setting, use the `GetControlMaximum` function.

When you set the maximum setting of a scroll bar equal to its minimum setting, the control definition function makes the scroll bar inactive; when you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.

SEE ALSO

Listing 5-16 on page 5-41 illustrates the use of `SetControlMaximum` to specify the maximum setting for a scroll bar.

SetControlTitle

To change the title of a control and redraw the control accordingly, use the `SetControlTitle` procedure. The `SetControlTitle` procedure is also available as the `SetCTitle` procedure.

```
PROCEDURE SetControlTitle (theControl: ControlHandle;
                           title: Str255);
```

`theControl`	A handle to a control, the title of which you want to change.
`title`	The new title for the control.

DESCRIPTION

The `SetControlTitle` procedure changes the `contrlTitle` field of the control record to the given string and redraws the control, using the system font for the control title.

The Control Manager allows multiple lines of text in the titles of buttons, checkboxes, and radio buttons. When specifying a multiple-line title, separate the lines with the ASCII character code $0D (carriage return). If the control is a button, each line is horizontally centered, and the font leading is inserted between lines. (The height of each line is equal to the distance from the ascent line to the descent line plus the leading of the font used. Be sure to make the total height of the rectangle greater than the number of lines times this height.) If the control is a checkbox or a radio button, the text is justified as appropriate for the user's current script system, and the checkbox or button is vertically centered within its rectangle.

When you create a control, you specify an initial title either in the control resource or in the `title` parameter of the `NewControl` function. To determine a control's current title, use the `GetControlTitle` procedure.

HideControl

To make a control invisible, before adjusting its size and location, for example, use the `HideControl` procedure.

```
PROCEDURE HideControl (theControl: ControlHandle);
```

`theControl`	A handle to the control you want to hide.

DESCRIPTION

The `HideControl` procedure makes the specified control invisible by changing the value of the `contrlVis` field of the control record and removing the control from the screen. To fill the region previously occupied by the control, `HideControl` uses the background pattern of the window's graphics port. It also adds the control's rectangle to the window's update region, so that anything else that was previously obscured by the control will reappear on the screen. If the control is already invisible, `HideControl` has no effect.

To make the control visible again, you can use the `ShowControl` procedure.

SPECIAL CONSIDERATIONS

The `MoveControl` and `SizeControl` procedures both call `HideControl` and `ShowControl` automatically. However, so that the control will not blink on the screen when you make both of these calls, you should use `HideControl` to make the control invisible until you are finished manipulating it, and then use `ShowControl`.

SEE ALSO

Listing 5-14 on page 5-39 illustrates the use of `HideControl` before adjusting scroll bar settings and locations.

MoveControl

To move a control within its window, you can use the `MoveControl` procedure.

```
PROCEDURE MoveControl (theControl: ControlHandle;
                       h: Integer; v: Integer);
```

`theControl`	A handle to the control you wish to move.
`h`	The horizontal coordinate (local to the control's window) of the new location of the upper-left corner of the control's rectangle.
`v`	The vertical coordinate (local to the control's window) of the new location of the upper-left corner of the control's rectangle.

DESCRIPTION

The `MoveControl` procedure moves the control to the new location specified by the `h` and `v` parameters, using them to change the rectangle specified in the `contrlRect` field of the control's control record. When the control is visible, `MoveControl` first hides it and then redraws it at its new location.

For example, if the user resizes a document window that contains a scroll bar, your application can use `MoveControl` to move the scroll bar to its new location.

SEE ALSO

Listing 5-24 on page 5-67 illustrates the use of `MoveControl` to change the location of a scroll bar.

SizeControl

To change the size of a control's rectangle, use the `SizeControl` procedure.

```
PROCEDURE SizeControl (theControl: ControlHandle;
                       h: Integer; v: Integer);
```

`theControl`	A handle to the control you wish to resize.
`w`	The new width, in pixels, of the resized control.
`h`	The new height, in pixels, of the resized control.

DESCRIPTION

The `SizeControl` procedure changes the rectangle specified in the `contrlRect` field of the control's control record. The lower-right corner of the rectangle is adjusted so that it has the width and height specified by the `w` and `h` parameters; the position of the upper-left corner is not changed. If the control is currently visible, it's first hidden and then redrawn in its new size. The `SizeControl` procedure uses `HideControl`, which changes the window's update region.

SEE ALSO

Listing 5-24 on page 5-67 illustrates the use of `SizeControl` to change the size of a scroll bar.

HiliteControl

If you need to change the highlighting of a control, you can use the `HiliteControl` procedure.

```
PROCEDURE HiliteControl (theControl: ControlHandle;
                         hiliteState: Integer);
```

`theControl` A handle to the control.

`hiliteState`
A value from 0 through 255 to signify the highlighting of the control. The value of 0 signifies no highlighting for the active control. A value from 1 through 253 signifies a part code designating the part of the (active) control to highlight. (Part codes are explained in the description of `FindControl` on page 5-89.) The value 255 signifies that the control is to be made inactive and drawn accordingly.

DESCRIPTION

The `HiliteControl` procedure calls the control definition function to redraw the control with the highlighting specified in the `hiliteState` parameter. The `HiliteControl` procedure uses the value in this parameter to change the value of the `contrlHilite` field of the control's control record.

Except for scroll bars (which you should hide using the `HideControl` procedure), you should use `HiliteControl` to make all controls inactive when their windows are not frontmost. The `TrackControl` function automatically uses the `HiliteControl` procedure as appropriate; when you use `TrackControl`, you don't need to call `HiliteControl`.

SPECIAL CONSIDERATIONS

The value 254 should not be passed in the `hiliteState` parameter; this value is reserved for future use.

SEE ALSO

The chapter "Dialog Manager" in this book provides several examples of the use of `HiliteControl`.

DragControl

If you need to draw and move an outline of a control or its indicator (such as the scroll box of a scroll bar) while the user drags it, you can use the `DragControl` procedure.

```
PROCEDURE DragControl (theControl: ControlHandle;
                       startPt: Point;
                       limitRect: Rect; slopRect: Rect;
                       axis: Integer);
```

`theControl` A handle to the control to drag.

`startPt` The location of the cursor, expressed in the local coordinates of the control's window, at the time the user first presses the mouse button.

`limitRect`	A rectangle—which should normally coincide with or be contained in the window's content region—delimiting the area in which the user can drag the control's outline.
`slopRect`	A rectangle that allows some extra space for the user to move the mouse while still constraining the control within the rectangle specified in the `limitRect` parameter.
`axis`	The axis along which the user may drag the control's outline. The following list shows the constants you can use—and the values they represent—for constraining the motion along an axis:

```
CONST
   noConstraint = 0; {no constraint}
   hAxisOnly    = 1; {drag along horizontal axis only}
   vAxisOnly    = 2; {drag along vertical axis only}
```

DESCRIPTION

The `DragControl` procedure moves a dotted outline of the control around the screen, following the movements of the cursor until the user releases the mouse button. When the user releases the mouse button, `DragControl` calls `MoveControl`. In turn, `MoveControl` moves the control to the location to which the user dragged it.

The `TrackControl` function automatically uses the `DragControl` procedure as appropriate; when you use `TrackControl`, you don't need to call `DragControl`.

The `startPt`, `limitRect`, `slopRect`, and `axis` parameters have the same meaning as for the Window Manager function `DragGrayRgn`.

SPECIAL CONSIDERATIONS

Before tracking the cursor, `DragControl` calls the control definition function. If you define your own control definition function, you can specify custom dragging behavior.

ASSEMBLY-LANGUAGE INFORMATION

Like `TrackControl`, `DragControl` invokes the Window Manager function `DragGrayRgn`, so you can use the global variables `DragHook` and `DragPattern`.

SEE ALSO

For information about creating your own control definition functions, see "Defining Your Own Control Definition Function" beginning on page 5-109. See the description of the `DragGrayRgn` function in the chapter "Window Manager" in this book for a more complete discussion of the `startPt`, `limitRect`, `slopRect`, and `axis` parameters, which are used identically in the `DragControl` function.

SetControlColor

To draw a control using colors other than the default colors used by system software, you can use the `SetControlColor` procedure. The `SetControlColor` procedure is also available as the `SetCtlColor` procedure.

```
PROCEDURE SetControlColor (theControl: ControlHandle;
                           newColorTable: CCTabHandle);
```

`theControl`	A handle to the control whose colors you wish to change.
`newColorTable`	A handle to a control color table record.

DESCRIPTION

The `SetControlColor` procedure changes the color table for the specified control. If the control currently has no auxiliary control record, `SetControlColor` creates one that includes the control color table record specified in the parameter `newColorTable` and adds the auxiliary control record to the head of the auxiliary control list. If there is already an auxiliary record for the control, `SetControlColor` replaces its color table with the contents of the control color table record specified in the parameter `newColorTable`.

To use nonstandard colors for a control, you must create a control color table, either by creating a color control table record and calling `SetControlColor` or by creating a control color table resource. Generally, you use `SetControlColor` when you create a control using `NewControl` and want to use nonstandard colors for it or when you change any control's colors after you've created it. When you want to use nonstandard colors for those controls you create in a control (`'CNTL'`) resource, you should create a control color table (`'cctb'`) resource with the same resource ID as the control resource.

A control whose colors you set with `SetControlColor` should initially be invisible. After using `SetControlColor` to set the control's colors, use the `ShowControl` procedure to make the control visible.

SPECIAL CONSIDERATIONS

On color monitors, the Control Manager automatically draws controls so that they match the colors of the controls used by system software. Be aware that nonstandard colors in your controls may initially confuse your users.

When you create a control color table record, your application should not deallocate it if another control is still using it.

SetControlAction

If you set the action procedure to `Pointer(-1)` when you use `TrackControl`, you can use the `SetControlAction` procedure to set or change the action procedure. The `SetControlAction` procedure is also available as the `SetCtlAction` procedure.

```
PROCEDURE SetControlAction (theControl: ControlHandle;
                            actionProc: ProcPtr);
```

`theControl`	A handle to the control whose action procedure you wish to change.
`actionProc`	A pointer to an action procedure defining what action your application takes while the user holds down the mouse button.

DESCRIPTION

The `SetControlAction` procedure changes the `contrlAction` field of the control's control record to point to the action procedure specified in the `actionProc` parameter. If the cursor is in the specified control, `TrackControl` calls this action procedure when user holds down the mouse button. You must provide the action procedure, and it must define some action to perform repeatedly as long as the user holds down the mouse button. (The `TrackControl` function always highlights and drags the control as appropriate.)

SPECIAL CONSIDERATIONS

The value in the `contrlAction` field of the control's control record is used by `TrackControl` only if you set the action procedure to `TrackControl` to `Pointer(–1)`.

An action procedure is usually specified in a parameter to `TrackControl`; you generally don't need to call `SetControlAction` to change it.

SEE ALSO

Action procedures are described in "Defining Your Own Action Procedures" beginning on page 5-115.

Determining Control Values

Your application sets a control's various values—such as current setting, minimum and maximum settings, title, reference value, and action procedure—when it creates the control. When the user clicks a control, however, your application often needs to determine the current setting and other possible values of that control. When the user clicks a checkbox, for example, your application must determine whether the box is checked before deciding whether to draw a checkmark inside the checkbox or remove the checkmark.

You can use the `GetControlValue`, `GetControlTitle`, `GetControlMinimum`, `GetControlMaximum`, `GetControlAction`, and `GetControlReference` routines to determine, respectively, a control's current setting, title, minimum setting, maximum setting, action procedure, and reference value. To get a handle to a control's auxiliary control record, you can use the `GetAuxiliaryControlRecord` function; your application can use the `acRefCon` field of an auxiliary control record for any purpose. To determine the variation code that is specified in the control definition function for a particular control, you can use the `GetControlVariant` function. This section also includes a description of the `SetControlReference` procedure, which allows your application to change its reference value for a control.

GetControlValue

To determine a control's current setting, use the `GetControlValue` function. The `GetControlValue` function is also available as the `GetCtlValue` function.

```
FUNCTION GetControlValue (theControl: ControlHandle): Integer;
```

`theControl` A handle to a control.

DESCRIPTION

The `GetControlValue` function returns as its function result the specified control's current setting, which is stored in the `contrlValue` field of the control record.

When you create a control, you specify an initial setting either in the control resource or in the `value` parameter of the `NewControl` function. You can change the setting by using the `SetControlValue` procedure.

SEE ALSO

Listing 5-12 on page 5-37 and Listing 5-13 on page 5-38 illustrate the use of `GetControlValue` for determining the current setting of, respectively, a pop-up menu and a checkbox. Listing 5-16 on page 5-41, Listing 5-18 on page 5-53, and Listing 5-20 on page 5-61 illustrate the use of this function for determining the current setting of a scroll bar.

GetControlMinimum

To determine a control's minimum setting, use the `GetControlMinimum` function. The `GetControlMinimum` function is also available as the `GetCtlMin` function.

```
FUNCTION GetControlMinimum (theControl: ControlHandle): Integer;
```

`theControl` A handle to the control whose minimum value you wish to determine.

DESCRIPTION

The `GetControlMinimum` function returns as its function result the specified control's minimum setting, which is stored in the `contrlMin` field of the control record.

When you create a control, you specify an initial minimum setting either in the control resource or in the `min` parameter of the `NewControl` function. You can change the minimum setting by using the `SetControlMinimum` procedure.

GetControlMaximum

To determine a control's maximum setting, use the `GetControlMaximum` function. The `GetControlMaximum` function is also available as the `GetCtlMax` function.

```
FUNCTION GetControlMaximum (theControl: ControlHandle): Integer;
```

`theControl`	A handle to the control whose maximum value you wish to determine.

DESCRIPTION

The `GetControlMaximum` function returns as its function result the specified control's maximum setting, which is stored in the `contrlMax` field of the control record.

When you create a control, you specify an initial maximum setting either in the control resource or in the `max` parameter of the `NewControl` function. You can change the maximum setting by using the `SetControlMaximum` procedure.

SEE ALSO

Listing 5-16 on page 5-41 and Listing 5-20 on page 5-61 illustrate the use of `GetControlMaximum` for determining the maximum scrolling distance of a scroll bar.

GetControlTitle

To determine the title of a control, use the `GetControlTitle` procedure. The `GetControlTitle` procedure is also available as the `GetCTitle` procedure.

```
PROCEDURE GetControlTitle (theControl: ControlHandle;
                           VAR title: Str255);
```

`theControl`	A handle to the control whose title you want to determine.
`title`	The title of the control.

DESCRIPTION

The `GetControlTitle` procedure returns the specified control title, which is stored in the `contrlTitle` field of the control record.

When you create a control, you specify an initial title either in the control resource or in the `title` parameter of the `NewControl` function. You can change the title by using the `SetControlTitle` procedure.

GetControlReference

To determine a control's current reference value, use the `GetControlReference` function. The `GetControlReference` function is also available as the `GetCRefCon` function.

```
FUNCTION GetControlReference (theControl: ControlHandle): LongInt;
```

`theControl`	A handle to the control whose current reference value you wish to determine.

DESCRIPTION

The `GetControlReference` function returns as its function result the current reference value for the specified control.

When you create a control, you specify an initial reference value, either in the control resource or in the `refCon` parameter of the `NewControl` function. The reference value is stored in the `contrlRfCon` field of the control record. You can use this field for any purpose, and you can use the `SetControlReference` procedure, described next, to change this value.

SetControlReference

To change a control's current reference value, use the `SetControlReference` procedure. The `SetControlReference` procedure is also available as the `SetCRefCon` procedure.

```
PROCEDURE SetControlReference (theControl: ControlHandle;
                               data: LongInt);
```

`theControl`	A handle to the control whose reference value you wish to change.
`data`	The new reference value for the control.

DESCRIPTION

The `SetControlReference` procedure sets the control's reference value to the value you specify in the `data` parameter.

When you create a control, you specify an initial reference value, either in the control resource or in the `refCon` parameter of the `NewControl` function. The reference value is stored in the `contrlRfCon` field of the control record; you can use the `GetControlReference` function to determine the current value. You can use this value for any purpose.

GetControlAction

To get a pointer to the action procedure stored in the `contrlAction` field of the control's control record, use the `GetControlAction` function. The `GetControlAction` function is also available as the `GetCtlAction` function.

```
FUNCTION GetControlAction (theControl: ControlHandle): ProcPtr;
```

`theControl` A handle to a control.

DESCRIPTION

The `GetControlAction` function returns as its function result whatever value is stored in the `contrlAction` field of the control's control record. This field specifies the action procedure that `TrackControl` uses if you set its `actionProc` parameter to `Pointer(-1)`. The action procedure should define an action to take in response to the user's holding down the mouse button while the cursor is in the control. You can use the `SetControlAction` procedure to change this action procedure.

SEE ALSO

For information about defining an action procedure, see "Defining Your Own Action Procedures" beginning on page 5-115.

GetControlVariant

To determine the variation code specified in the control definition function for a particular control, you can use the `GetControlVariant` function. The `GetControlVariant` function is also available as the `GetCVariant` function.

```
FUNCTION GetControlVariant (theControl: ControlHandle): Integer;
```

`theControl` A handle to the control whose variation code you wish to determine.

DESCRIPTION

The `GetControlVariant` function returns as its function result the variation code for the specified control.

SEE ALSO

Variation codes are described in "The Control Definition Function" on page 5-14.

GetAuxiliaryControlRecord

Use the `GetAuxiliaryControlRecord` function to get a handle to a control's auxiliary control record. The `GetAuxiliaryControlRecord` function is also available as the `GetAuxCtl` function.

```
FUNCTION GetAuxiliaryControlRecord (theControl: ControlHandle;
                                    VAR acHndl: AuxCtlHandle)
                                    : Boolean;
```

`theControl`	A handle to a control.
`acHndl`	A handle to the auxiliary control record for the control.

DESCRIPTION

In its `acHndl` parameter, the `GetAuxiliaryControlRecord` function returns a handle to the auxiliary control record for the specified control. Your application typically doesn't need to access an auxiliary control record unless you need its `acRefCon` field, which your application can use for any purpose.

The value that `GetAuxiliaryControlRecord` returns for a function result depends on the control's color control table, as described here:

- If your application has changed the default control color table for the given control (either by using the `SetControlColor` procedure or by creating its own control color table), the function returns `TRUE`.
- If your application has *not* changed the default control color table, the function returns `FALSE`.
- If you set the parameter `theControl` to `NIL`, the Dialog Manager ensures that the control uses the default color table, and `GetAuxiliaryControlRecord` returns `TRUE`.

Removing Controls

When you use the Window Manager procedures `DisposeWindow` and `CloseWindow` to remove a window, they automatically remove all controls associated with the window and release the memory the controls occupy.

When you no longer need a control in a window that you want to keep, you can use the `DisposeControl` procedure to remove the control from the window's control list and release the memory it occupies. You can use the `KillControls` procedure to dispose of all of a window's controls at once.

DisposeControl

To remove a particular control from a window that you want to keep, use the `DisposeControl` procedure.

```
PROCEDURE DisposeControl (theControl: ControlHandle);
```

`theControl` A handle to the control you wish to remove.

DESCRIPTION

The `DisposeControl` procedure removes the specified control from the screen, deletes it from its window's control list, and releases the memory occupied by the control record and any data structures associated with the control.

SPECIAL CONSIDERATIONS

The Window Manager procedures `CloseWindow` and `DisposeWindow` automatically dispose of all controls associated with the given window.

SEE ALSO

To remove all of the controls in a window, use the `KillControls` procedure, described next. The `CloseWindow` and `DisposeWindow` procedures are described in the chapter "Window Manager" in this book.

KillControls

To remove all of the controls in a particular window that you want to keep, use the `KillControls` procedure.

```
PROCEDURE KillControls (theWindow: WindowPtr);
```

`theWindow` A pointer to the window containing the controls to remove.

DESCRIPTION

The `KillControls` procedure disposes of all controls associated with the specified window by calling the `DisposeControl` procedure for each control.

SPECIAL CONSIDERATIONS

The Window Manager procedures `CloseWindow` and `DisposeWindow` automatically dispose of all controls associated with the given window.

SEE ALSO

The `CloseWindow` and `DisposeWindow` procedures are described in the chapter "Window Manager" in this book.

Application-Defined Routines

This section describes how to create your own control definition function—declared here as `MyControl`—which your application needs to provide when defining new, nonstandard controls. This section also describes action procedures—declared here as `MyAction` and `MyIndicatorAction`—which define additional actions to be performed repeatedly as long as the user holds down the mouse button while the cursor is in a control. For example, you need to define an action procedure for scrolling through a document while the user holds down the mouse button and the cursor is in a scroll arrow.

Defining Your Own Control Definition Function

In addition to the standard controls (buttons, checkboxes, radio buttons, pop-up menus, and scroll bars), the Control Manager allows you to define new, nonstandard controls as appropriate for your application. For example, you can define a three-way selector switch, a memory-space indicator that looks like a thermometer, or a thruster control for a spacecraft simulator. Controls and their indicators may occupy regions of any shape, as permitted by QuickDraw.

To define your own type of control, you write a control definition function, compile it as a resource of type `'CDEF'`, and store it in your resource file. (See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about creating resources.) Whenever you create a control, you specify a control definition ID, which the Control Manager uses to determine the control definition function. The control definition ID is an integer that contains the resource ID of the control definition function in its upper 12 bits and a variation code in its lower 4 bits. Thus, for a given resource ID and variation code

control definition ID = 16 x resource ID + variation code

For example, buttons, checkboxes, and radio buttons all use the standard control definition function with resource ID 0. Because they have variation codes of 0, 1, and 2, respectively, their respective control definition IDs are 0, 1, and 2.

You can define your own variation codes, which various Control Manager routines pass to your control definition function. This allows you to use one 'CDEF' resource to handle several variations of the same general control.

The Control Manager calls the Resource Manager to access your control definition function with the given resource ID. The Resource Manager reads your control definition function into memory and returns a handle to it. The Control Manager stores this handle in the contrlDefProc field of the control record. In 24-bit addressing mode, the variation code is placed in the high-order byte of this field; in 32-bit mode, the variation code is placed in the most significant byte of the acReserved field in the control's AuxCtlRec record. Later, when various Control Manager routines need to perform a type-dependent action on the control, they call your control definition function and pass it the variation code as a parameter.

If you create a control definition function, you can use control color table records of any desired size and define their contents in any way you wish, except that part indices 1 through 127 are reserved for system definition. Note that in this case, you should allocate explicit auxiliary records for every control you create.

MyControl

If you wish to define new, nonstandard controls for your application, you must write a control definition function and store it in a resource file as a resource of type 'CDEF'. Here's how you would declare a procedure named MyControl:

```
FUNCTION MyControl (varCode: Integer; theControl: ControlHandle;
                     message: Integer; param: LongInt): LongInt;
```

varCode	The variation code for this control. To derive the control definition ID for the control, add this value to the result of 16 multiplied by the resource ID of the 'CDEF' resource containing this function. The variation code allows you to specify several control definition IDs within one 'CDEF' resource, thereby defining several variations of the same basic control.
theControl	A handle to the control that the operation will affect.
message	A value (from the following list) that specifies which operation your function must undertake.

```
CONST drawCntl      = 0;  {draw the control or its part}
      testCntl      = 1;  {test where mouse button }
                          { is pressed}
      calcCRgns     = 2;  {calculate region for }
                          { control or indicator in }
                          { 24-bit systems}
      initCntl      = 3;  {peform any additional }
                          { control initialization}
```

```
               dispCntl       = 4;  {perform any additional }
                                    { disposal actions}
               posCntl        = 5;  {move indicator and }
                                    { update its setting}
               thumbCntl      = 6;  {calculate parameters for }
                                    { dragging indicator}
               dragCntl       = 7;  {perform any custom dragging }
                                    { of control or its indicator}
               autoTrack      = 8;  {execute action procedure }
                                    { specified by your function}
               calcCntlRgn    = 10; {calculate region for control}
               calcThumbRgn   = 11; {calculate region for }
                                    { indicator}
```

`param`	A value whose meaning depends on the operation specified in the `message` parameter.

DESCRIPTION

The Control Manager calls your control definition function under various circumstances; the Control Manager uses the `message` parameter to inform your control definition function what action it must perform. The data that the Control Manager passes in the `param` parameter, the action that your control definition function must undertake, and the function result that your control definition function returns all depend on the value that the Control Manager passes in the `message` parameter. The rest of this section describes how to respond to the various values that the Control Manager passes in the `message` parameter.

Drawing the Control or Its Part

When the Control Manager passes the value for the `drawCntl` constant in the `message` parameter, the low word in the `param` parameter has one of the following values:

- the value 0, indicating the entire control
- the value 129, signifying an indicator that must be moved
- any other value, indicating a part code for the control (Don't use part code 128, which is reserved for future use, or part code 129, which the Control Manager uses to signify an indicator that must be moved.)

Note
For the `drawCntl` message, the high-order word of the `param` parameter may contain undefined data; therefore, evaluate only the low-order word of this parameter. u

If the specified control is visible, your control definition function should draw the control (or the part specified in the `param` parameter) within the control's rectangle. If the control is invisible (that is, if its `contrlVis` field is set to 0), your control definition function does nothing.

When drawing the control or its part, take into account the current values of its `contrlHilite` and `contrlValue` fields of the control's control record.

If the part code for your control's indicator is passed in `param`, assume that the indicator hasn't moved; the Control Manager, for example, may be calling your control definition function so that you may simply highlight the indicator. However, when your application calls the `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum` procedures, they in turn may call your control definition function to redraw the indicator. Since these routines have no way of determining what part code you chose for your indicator, they all pass 129 in `param`, meaning that you should move your indicator. Your control definition function must detect this part code as a special case and remove the indicator from its former location before drawing it. If your control has more than one indicator, you should interpret 129 to mean all indicators.

When passed the value for the `drawCntl` constant in the `message` parameter, your control definition function should always return 0 as its function result.

Testing Where the Mouse-Down Event Occurs

To request your control definition function to determine whether a specified point is in a visible control, the `FindControl` function sends the value for the `testCntl` constant in the `message` parameter. In this case, the `param` parameter specifies a point (in coordinates local to the control's window) as follows:

- The point's vertical coordinate is contained in the high-order word of the long integer.
- The point's horizontal coordinate is contained in the low-order word.

When passed the value for the `testCntl` constant in the `message` parameter, your control definition function should return the part code of the part that contains the specified point; it should return 0 if the point is outside the control or if the control is inactive.

Calculating the Control and Indicator Regions

When the Control Manager passes the value for the `calcCRgns` constant in the `message` parameter, your control definition function should calculate the region occupied by either the control or its indicator. The Control Manager passes a QuickDraw region handle in the `param` parameter; it is this region that you calculate. If the high-order bit of `param` is set, the region requested is that of the control's indicator; otherwise, the region requested is that of the entire control. Your control definition function should clear the high bit of the region handle before calculating the region.

When the Control Manager passes the value for the `calcCntlRgn` constant in the `message` parameter, your control definition function should calculate the region passed in the `param` parameter for the specified control. When the Control Manager passes the value for the `calcThumbRgn` constant, calculate the region occupied by the indicator.

When passed the values for the `calcCRgns`, `calcCntlRgn`, and `calcThumbRgn` constants, your control definition function should always return 0, and it should express the region in the local coordinate system of the control's window.

IMPORTANT
The Control Manager passes the `calcCRgns` constant when the 24-bit Memory Manager is in operation. When the 32-bit Memory Manager is in operation, the Control Manager instead passes the `calcCntlRgn` constant or the `calcThumbRgn` constant. Your control definition function should respond to all three constants. s

Performing Any Additional Initialization

After initializing fields of a control record as appropriate when creating a new control, the Control Manager passes `initCntl` in the `message` parameter to give your control definition function the opportunity to perform any type-specific initialization you may require. For example, if you implement the control's action procedure in its control definition function, you'll need to store `Pointer(-1)` in the `contrlAction` field of the control's control record. Then, in a call to `TrackControl` for this control, you would pass `Pointer(-1)` in the `actionProc` parameter of `TrackControl`.

The standard control definition function for scroll bars allocates space for a region to hold the scroll box and stores the region handle in the `contrlData` field of the new control record.

When passed the value for the `initCntl` constant in the `message` parameter, your control definition function should ignore the `param` parameter and return 0 as a function result.

Performing Any Additional Disposal Actions

The `DisposeControl` procedure passes `dispCntl` in the `message` parameter to give your control definition function the opportunity to carry out any additional actions when disposing of a control. For example, the standard definition function for scroll bars releases the memory occupied by the scroll box region, whose handle is kept in the `contrlData` field of the control's control record.

When passed the value for the `dispCntl` constant in the `message` parameter, your control definition function should ignore the `param` parameter and return 0 as a function result.

Moving the Indicator

When a mouse-up event occurs in the indicator of a control, the `TrackControl` function calls your control definition function and passes `posCntl` in the `message` parameter. In this case, the `param` parameter contains a point (in coordinates local to the control's window) that specifies the vertical and horizontal offset, in pixels, by which your control definition function should move the indicator from its current position. Typically, this is the offset between the points where the cursor was when the user pressed and released the mouse button while dragging the indicator. The offset point is specified as follows:

- The point's vertical offset is contained in the high-order word of the `param` parameter.
- The point's horizontal offset is contained in the low-order word.

Your definition function should calculate the control's new setting based on the given offset and then, to reflect the new setting, redraw the control and update the `contrlValue` field in the control's control record. Your control definition function should ignore the `param` parameter and return 0 as a function result.

Note that the `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum` procedures do not call your control definition function with the `posCntl` message; instead, they pass the `drawCntl` message.

Calculating Parameters for Dragging the Indicator

When the Control Manager passes the value for `thumbCntl` in the `message` parameter, your control definition function should respond by calculating values (analogous to the `limitRect`, `slopRect`, and `axis` parameters of `DragControl`) that constrain how the indicator is dragged. The `param` parameter contains a pointer to the following data structure:

```
RECORD
   limitRect,slopRect:  Rect;
   axis:                Integer;
END;
```

On entry, the field `param^.limitRect.topLeft` contains the point where the mouse-down event first occurred. Your definition function should store the appropriate values into the fields of the record pointed to by `param`; they're analogous to the similarly named parameters to the Window Manager function `DragGrayRgn`.

Performing Custom Dragging

The Control Manager passes `dragCntl` in the `message` parameter to give your control definition function the opportunity to specify its own method for dragging a control (or its indicator).

The `param` parameter specifies whether the user is dragging an indicator or the whole control:

- A value of 0 means the user is dragging the entire control.
- Any nonzero value means the user is dragging only the indicator.

If you want to use the Control Manager's default method of dragging (which is to call `DragControl` to drag the control or the Window Manager function `DragGrayRgn` to drag its indicator), return 0 as the function result for your control definition function.

If your control definition function returns any nonzero result, the Control Manager does not drag your control, and instead your control definition function must drag the specified control (or its indicator) to follow the cursor until the user releases the mouse button, as follows:

- If the user drags the entire control, your definition function should use the `MoveControl` procedure to reposition the control to its new location after the user releases the mouse button.

- If the user drags the indicator, your definition function must calculate the control's new setting (based on the pixel offset between the points where the cursor was when the user pressed and released the mouse button while dragging the indicator) and then, to reflect the new setting, redraw the control and update the `contrlValue` field in the control's control record. Note that, in this case, the `TrackControl` function returns 0 whether or not the user changes the indicator's position. Thus, you must determine whether the user has changed the control's setting, for instance, by comparing the control's value before and after the call to `TrackControl`.

Executing an Action Procedure

You can design a control whose action procedure is specified by your control definition function. When you create the control, your control definition function must first respond to the `initCntl` message by storing `Pointer(-1)` in the `contrlAction` field of the control's control record. (As previously explained, the Control Manager sends the `initCntl` message to your control definition function after initializing the fields of a new control record.) Then, when your application passes `Pointer(-1)` in the `actionProc` parameter to the `TrackControl` function, `TrackControl` calls your control definition function with the `autoTrack` message. The `param` parameter specifies the part code of the part where the mouse-down event occurs. Your control definition function should then use this information to respond as an action procedure would.

Note

For the `autoTrack` message, the high-order word of the `param` parameter may contain undefined data; therefore, evaluate only the low-order word of this parameter. u

ASSEMBLY-LANGUAGE INFORMATION

The function's entry point must be at the beginning.

SEE ALSO

The `TrackControl` function is described on page 5-90; creating an action procedure is described in the next section.

Defining Your Own Action Procedures

When a mouse-down event occurs in a control, the `TrackControl` function responds as appropriate by highlighting the control or dragging the indicator as long as the user holds down the mouse button. You can define other actions to be performed repeatedly during this interval. To do so, define your own action procedure and point to it in the `actionProc` parameter of the `TrackControl` function.

When calling your action procedure for a control part other than an indicator, `TrackControl` passes your action procedure (1) a handle to the control and (2) the control's part code. Your action procedure should then respond as appropriate. For example, if the user is working in a text document and holds down the mouse button while the cursor is in the lower scroll arrow, your application should scroll continuously one line at a time until the user releases the mouse button or reaches the end of the document.

For a control part other than an indicator, you declare an action procedure that takes two parameters: a handle to the control in which the mouse-down event occurred and an integer that represents the part of the control in which the mouse-down event occurred. Such an action procedure is declared as `MyAction` in the following section.

If the mouse-down event occurs in an indicator, your action procedure should take no parameters, because the user may move the cursor outside the indicator while dragging it. Such an action procedure, declared here as `MyIndicatorAction`, is described on page 5-117.

Because it will be called with either zero or two parameters, according to whether the mouse-down event occurred in an indicator or elsewhere, your action procedure can be defined for only one case or the other. The only way to specify actions in response to all mouse-down events in a control, regardless of whether they're in an indicator, is to define your own control definition function, as described in "Defining Your Own Control Definition Function" beginning on page 5-109.

MyAction

Here's how to declare an action procedure for a control part other than an indicator if you were to name the procedure `MyAction`:

```
PROCEDURE MyAction (theControl: ControlHandle; partCode: Integer);
```

`theControl`	A handle to the control in which the mouse-down event occurred.
`partCode`	When the cursor is still in the control part where mouse-down event first occurred, this parameter contains that control's part code. When the user drags the cursor outside the original control part, this parameter contains 0.

DESCRIPTION

Your procedure can perform any action appropriate for the control part. For example, when a mouse-down event occurs in a scroll arrow or gray area of a scroll bar, `TrackControl` calls your action procedure and passes it the part code and a handle to the scroll bar. Your action procedure should examine the part code to determine the part of the control in which the mouse-down event occurred. Your action procedure should then scroll up or down a line or page as appropriate and then call the `SetControlValue` procedure to change the control's setting and redraw the scroll box.

ASSEMBLY-LANGUAGE INFORMATION

If you store a pointer to a procedure in the global variable `DragHook`, your procedure is called repeatedly (with no parameters) as long as the user holds down the mouse button. The `TrackControl` function invokes the Window Manager function `DragGrayRgn`, which calls the `DragHook` procedure. The `DragGrayRgn` function uses the pattern stored in the global variable `DragPattern` for the dragged outline of the indicator.

SEE ALSO

Listing 5-19 on page 5-59 illustrates a pair of action procedures for scrolling through a text document. As an alternative to passing a pointer to your action procedure in a parameter to `TrackControl`, you can use the `SetControlAction` procedure to store a pointer to the action procedure in the `contrlAction` field in the control record. When you pass `Pointer(-1)` instead of a procedure pointer to `TrackControl`, `TrackControl` uses the action procedure pointed to in the control record.

MyIndicatorAction

Here's how to declare an action procedure for an indicator if you were to name the procedure `MyIndicatorAction`:

```
PROCEDURE MyIndicatorAction;
```

DESCRIPTION

Your procedure can perform any action appropriate for the control part. For example, if your application plays music while displaying a volume control slider, your application should change the volume in response to the user's action in the slider switch.

SEE ALSO

See the `MyAction` procedure described on page 5-116 for other considerations.

Resources

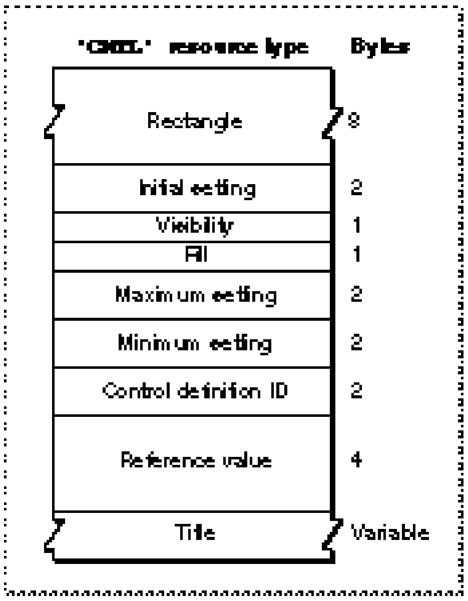
This section describes the control (`'CNTL'`) resource and the control color table (`'cctb'`) resource. You can use the control resource to define a control and use the control color table resource to change the default colors of a control's parts.

The Control Resource

You can use a control resource to define a control. A control resource is a resource of type `'CNTL'`. All control resources must have resource ID numbers greater than 128. Use the `GetNewControl` function (described on page 5-81) to create a control defined in a control resource. The Control Manager uses the information you specify to create a control record in memory. (The control record is described on page 5-73.)

This section describes the structure of this resource after it is compiled by the Rez resource compiler, available from APDA. The format of a Rez input file for a control resource differs from its compiled output form, which is illustrated in Figure 5-25. If you are concerned only with creating a control resource, see "Creating and Displaying a Control" beginning on page 5-15.

Figure 5-25 Structure of a compiled control (`'CNTL'`) resource



The compiled version of a control resource contains the following elements:

- The rectangle, specified in coordinates local to the window, that encloses the control; this rectangle encloses the control and thus determines its size and location.
- The initial setting for the control.
  - For controls—such as buttons—that don't retain a setting, this value should be 0.
  - For controls—such as checkboxes or radio buttons—that retain an on-or-off setting, a value of 0 in this element indicates that the control is initially off; a value of 1 indicates that the control is initially on.
  - For controls—such as scroll bars and dials—that can take a range of settings, whatever initial value is appropriate within that range is specified in this element.

- For pop-up menus, a combination of values instructs the Control Manager where and how to draw the control title. Appropriate values, along with the constants used to specify them in a Rez input file, are listed here:

```
CONST popupTitleBold       = $00000100;   {boldface font style}
      popupTitleItalic     = $00000200;   {italic font style}
      popupTitleUnderline  = $00000400;   {underline font }
                                          { style}
      popupTitleOutline    = $00000800;   {outline font style}
      popupTitleShadow     = $00001000;   {shadow font style}
      popupTitleCondense   = $00002000;   {condensed text}
      popupTitleExtend     = $00004000;   {extended text}
      popupTitleNoStyle    = $00008000;   {monostyle text}
      popupTitleLeftJust   = $00000000;   {place title left }
                                          { of pop-up box}
      popupTitleCenterJust = $00000001;   {center title over }
                                          { pop-up box}
      popupTitleRightJust  = $000000FF;   {place title right }
                                          { of pop-up box}
```

- The visibility of the control. If this element contains the value `TRUE`, `GetNewControl` draws the control immediately, without using the application's standard updating mechanism for windows. If this element contains the value `FALSE`, the application must use the `ShowControl` procedure (described on page 5-86) when it's prepared to display the control.
- Fill. This should be set to 0.
- The maximum setting for the control.
  - For controls—such as buttons—that don't retain a setting, this value should be 1.
  - For controls—such as checkboxes or radio buttons—that retain an on-or-off setting, this element should contain the value 1 (meaning "on").
  - For controls—such as scroll bars and dials—that can take a range of settings, this element can contain whatever maximum value is appropriate; when the application makes the maximum setting of a scroll bar equal to its minimum setting, the control definition function automatically makes the scroll bar inactive, and when the application makes the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.
  - For pop-up menus, this element contains the width, in pixels, of the control title.
- The minimum setting for the control.
  - For controls—such as buttons—that don't retain a setting, this value should be 0.
  - For controls—such as checkboxes or radio buttons—that retain an on-or-off setting, the value 0 (meaning "off") should be set in this element.
  - For controls—such as scroll bars and dials—that can take a range of settings, this element contains whatever minimum value is appropriate.
  - For pop-up menus, this element contains the resource ID of the `'MENU'` resource that describes the menu items.

- The control definition ID, which the Control Manager uses to determine the control definition function for this control. “Defining Your Own Control Definition Function” beginning on page 5-109 describes how to create control definition functions and their corresponding control definition IDs. The following list shows the control definition ID numbers—and the constants that represent them in Rez input files—for the standard controls.

```
CONST
   pushButProc          = 0;     {button}
   checkBoxProc         = 1;     {checkbox}
   radioButProc         = 2;     {radio button}
   useWFont             = 8;     {when added to above, shows }
                                 { title in the window font}
   scrollBarProc        = 16;    {scroll bar}
   popupMenuProc        = 1008;  {pop-up menu}
   popupFixedWidth      = $0001; {add to popupMenuProc to }
                                 { use fixed-width control}
   popupUseAddResMenu   = $0004; {add to popupMenuProc to }
                                 { specify a value of type }
                                 { ResType in the contrlRfCon }
                                 { field of the control }
                                 { record; Menu Manager }
                                 { adds resources of this }
                                 { type to the menu}
   popupUseWFont        = $0008; {if added to popupMenuProc, }
                                 { shows title in window font}
```

Note

The title of a button, checkbox, radio button, or pop-up menu normally appears in the system font, which in Roman script systems is 12-point Chicago. Do not use a smaller font; some script systems, such as KanjiTalk, require 12-point fonts. You should generally use the system font in your controls; doing so will simplify localization effort. However, if you absolutely need to display a control title in the font currently associated with the window’s graphics port, you can add the `popupUseWFont` constant to the pop-up menu control definition ID or add the `useWFont` constant to the other standard control definition IDs. u

- The control’s reference value, which is set and used only by the application (except when the application adds the `popupUseAddResMenu` variation code to the `popupMenuProc` control definition ID, as described in “Creating a Pop-Up Menu” beginning on page 5-25).

- For controls—such as buttons, checkboxes, radio buttons, and pop-up menus—that need a title, the string for that title; for controls that don't use titles, an empty string.

After you use `GetNewControl` to create the control, you can change the current setting, the maximum setting, the minimum setting, the reference value, and the title by using, respectively, the `SetControlValue`, `SetControlMaximum`, `SetControlMinimum`, `SetControlReference`, and `SetControlTitle` routines. You can use the `MoveControl` and `SizeControl` procedures to change the control's rectangle. You can use the `GetControlValue`, `GetControlMaximum`, `GetControlMinimum`, `GetControlReference`, and `GetControlTitle` routines to determine the control values.

The Control Color Table Resource

On color monitors, the Control Manager automatically draws control parts so that they match the colors of the controls used by system software.

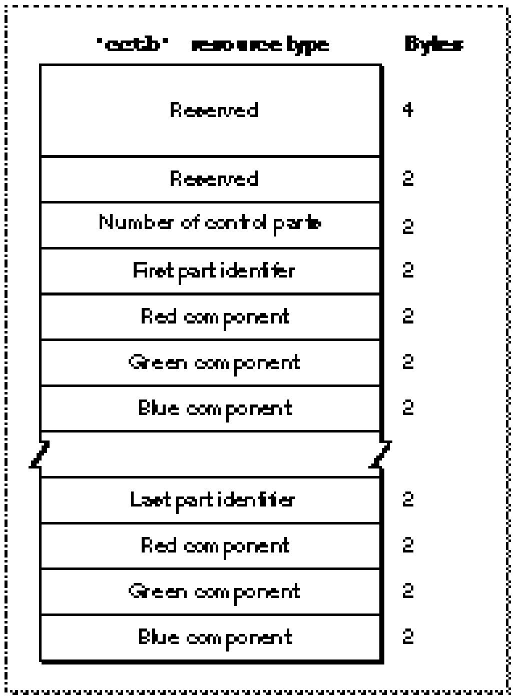
If you feel absolutely compelled to use nonstandard colors, the Control Manager allows you to do so. Your application can specify these by creating a control color table (`'cctb'`) resource; you must give the control color table resource for a control the same resource ID as its control (`'CNTL'`) resource, which is described on page 5-118. When you call the `GetNewControl` function to create the control, the Control Manager automatically attempts to load a control color table resource with the same resource ID as the control resource specified to `GetNewControl`. The Control Manager also creates an auxiliary control record for the control; the auxiliary control record is described on page 5-76.

Note
Using nonstandard colors in your controls may initially confuse your users. u

Generally, you use a control color table resource for a control that you define in a control resource. To change a control's colors, or to use nonstandard colors in a control you create using `NewControl`, create a control color table record and use the `SetControlColor` procedure. The control color table record is described on page 5-77; the `SetControlColor` procedure is described on page 5-101.

A control color table resource is of type `'cctb'`. All control color table resources must have resource ID numbers greater than 128. Figure 5-26 on the next page shows the format of a control color table resource. Note that `DisposeControl` does not delete a control color table resource; therefore, you should make each control color table resource purgeable.

Figure 5-26 Structure of a compiled control color table (`'cctb'`) resource



You define a control color table resource by specifying these elements in a resource with the `'cctb'` resource type:

- Reserved. Should always be set to 0.
- Reserved. Should always be set to 0.
- Number of control parts. For standard controls other than scroll bars, this should be set to 3, because these controls consist of a frame, a control body, and text. For scroll bars, this should be set to 12. A scroll bar consists of a frame, a body, and scroll box; each part of a scroll bar has various highlight and tinge colors associated with it. To create a control with more parts, you must create your own control definition function (as described in "Defining Your Own Control Definition Function" beginning on page 5-109) that recognizes additional parts.
- First part identifier. A value or constant that identifies the control's part to color. The part identifiers can be listed in any order. The scroll bar control definition function may use more than one part identifier to produce the actual colors used for each part of the scroll bar.

```
CONST
   cFrameColor       = 0;  {frame color; for scroll bars, used to produce }
                           { foreground color for scroll arrows & gray area}
   cBodyColor        = 1;  {body color; for scroll bars, used to produce }
                           { colors in the scroll box}
   cTextColor        = 2;  {text color; unused for scroll bars}
```

```
cArrowsColorLight = 5;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cArrowsColorDark  = 6;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cThumbLight       = 7;  {Used to produce colors in scroll box}
cThumbDark        = 8;  {Used to produce colors in scroll box}
cHiliteLight      = 9;  {Use same value as wHiliteColorLight in 'wctb'}
cHiliteDark       = 10; {Use same value as wHiliteColorDark in 'wctb'}
cTitleBarLight    = 11; {Use same value as wTitleBarLight in 'wctb'}
cTitleBarDark     = 12; {Use same value as wTitleBarDark in 'wctb'}
cTingeLight       = 13; {Use same value as wTingeLight in 'wctb'}
cTingeDark        = 14; {Use same value as wTingeDark in 'wctb'}
```

- Red component. An integer that represents the intensity of the red component of the color to use when drawing this part of the control. In this and the next two elements, use 16-bit unsigned integers to give the intensity values of three additive primary colors.
- Green component. An integer that represents the intensity of the green component of the color to use when drawing this part of the control.
- Blue component. An integer that represents the intensity of the blue component of the color to use when drawing this part of the control.
- Part identifier and red, green, and blue components for the next control part. You can list parts in any order in this resource. If the application specifies a part identifier that cannot be found, the Control Manager uses the colors for the control's first identifiable part. If a part is not listed in the control color table, the Dialog Manager draws it in its default color.

The Control Definition Function

The resource type for a control definition function is `'CDEF'`. The resource data is the compiled or assembled code of the function. See "Defining Your Own Control Definition Function" beginning on page 5-109 for information about creating a control definition function.

Summary of the Control Manager

Pascal Summary

Constants

```
CONST
   {control definition IDs}
   pushButProc          = 0;     {button}
   checkBoxProc         = 1;     {checkbox}
   radioButProc         = 2;     {radio button}
   useWFont             = 8;     {add to above to display control title in }
                                 { the window font}
   scrollBarProc        = 16;    {scroll bar}
   popupMenuProc        = 1008;  {pop-up menu}
   popupMenuCDEFproc    = popupMenuProc;  {synonym for compatibility}

   {pop-up menu CDEF variation codes}
   popupFixedWidth      = $0001;    {add to popupMenuProc to use }
                                    { fixed-width control}
   popupUseAddResMenu   = $0004;    {add to popupMenuProc to specify a }
                                    { value of type ResType in the }
                                    { contrlRfCon field of the control }
                                    { record; Menu Manager adds }
                                    { resources of this type to the menu}
   popupUseWFont        = $0008;    {add to popupMenuProc to show control }
                                    { title in the window font}

   {part codes}
   inButton             = 10;    {button}
   inCheckBox           = 11;    {checkbox or radio button}
   inUpButton           = 20;    {up arrow for a vertical scroll bar, }
                                 { left arrow for a horizontal scroll bar}
   inDownButton         = 21;    {down arrow for a vertical scroll bar, }
                                 { right arrow for a horizontal scroll bar}
   inPageUp             = 22;    {gray area above scroll box for a }
                                 { vertical scroll bar, gray area to }
                                 { left of scroll box for a horizontal }
                                 { scroll bar}
```

```
inPageDown          = 23;    {gray area below scroll box for a }
                             { vertical scroll bar, gray area to }
                             { right of scroll box for a horizontal }
                             { scroll bar}
inThumb             = 129;   {scroll box (or other indicator)}

{pop-up title characteristics}
popupTitleBold      = $00000100;   {boldface font style}
popupTitleItalic    = $00000200;   {italic font style}
popupTitleUnderline = $00000400;   {underline font style}
popupTitleOutline   = $00000800;   {outline font style}
popupTitleShadow    = $00001000;   {shadow font style}
popupTitleCondense  = $00002000;   {condensed characters}
popupTitleExtend    = $00004000;   {extended characters}
popupTitleNoStyle   = $00008000;   {monostyled text}
popupTitleLeftJust  = $00000000;   {place title left of pop-up box}
popupTitleCenterJust = $00000001;  {center title over pop-up box}
popupTitleRightJust = $000000FF;   {place title right of pop-up box}

{axis constraints for DragControl procedure}
noConstraint        = 0;  {no constraint}
hAxisOnly           = 1;  {drag along horizontal axis only}
vAxisOnly           = 2;  {drag along vertical axis only}

{constants for the message parameter in a control definition function}
drawCntl            = 0;  {draw the control or its part}
testCntl            = 1;  {test where mouse button is pressed}
calcCRgns           = 2;  {calculate region for control or indicator in }
                          { 24-bit systems}
initCntl            = 3;  {peform any additional control initialization}
dispCntl            = 4;  {take any additional disposal actions}
posCntl             = 5;  {move indicator and update its setting}
thumbCntl           = 6;  {calculate parameters for dragging indicator}
dragCntl            = 7;  {perform any custom dragging of control or }
                          { its indicator}
autoTrack           = 8;  {execute action procedure specified by your }
                          { function}
calcCntlRgn         = 10; {calculate region for control}
calcThumbRgn        = 11; {calculate region for indicator}

{part identifiers for ColorSpec records in a control color table resource}
cFrameColor         = 0;  {frame color; for scroll bars, also fore- }
                          { ground color for scroll arrows and gray area}
```

```
   cBodyColor          = 1;  {for scroll bars, background color for }
                             { scroll arrows and gray area; for other }
                             { controls, the fill color for body of control}
   cTextColor          = 2;  {text color; unused for scroll bars}
   cThumbColor         = 3;  {Reserved}
```

Data Types

```
TYPE  ControlPtr         = ^ControlRecord;
      ControlHandle      = ^ControlPtr;

      ControlRecord =
      PACKED RECORD
         nextControl:   ControlHandle; {next control}
         contrlOwner:   WindowPtr;     {control's window}
         contrlRect:    Rect;          {rectangle}
         contrlVis:     Byte;          {255 if visible}
         contrlHilite:  Byte;          {highlight state}
         contrlValue:   Integer;       {control's current setting}
         contrlMin:     Integer;       {control's minimum setting}
         contrlMax:     Integer;       {control's maximum setting}
         contrlDefProc: Handle;        {control definition function}
         contrlData:    Handle;        {data used by contrlDefProc}
         contrlAction:  ProcPtr;       {action procedure}
         contrlRfCon:   LongInt;       {control's reference value}
         contrlTitle:   Str255;        {control's title}
      END;

      AuxCtlPtr      = ^AuxCtlRec;
      AuxCtlHandle   = ^AuxCtlPtr;

      AuxCtlRec =
      RECORD
         acNext:        AuxCtlHandle;  {handle to next AuxCtlRec}
         acOwner:       ControlHandle; {handle to this record's control}
         acCTable:      CCTabHandle;   {handle to color table record}
         acFlags:       Integer;       {reserved}
         acReserved:    LongInt;       {reserved for future use}
         acRefCon:      LongInt;       {for use by application}
      END;
```

```
CCTabPtr       = ^CtlCTab;
CCTabHandle    = ^CCTabPtr;

CtlCTab =
RECORD
   ccSeed:        LongInt;       {reserved; set to 0}
   ccRider:       Integer;       {reserved; set to 0}
   ctSize:        Integer;       {number of ColorSpec records in next }
                                 { field; 3 for standard controls}
   ctTable:       ARRAY[0..3] OF ColorSpec;
END;
```

Control Manager Routines

Creating Controls

```
FUNCTION GetNewControl      (controlID: Integer; owner: WindowPtr)
                             : ControlHandle;
FUNCTION NewControl         (theWindow: WindowPtr; boundsRect: Rect;
                             title: Str255; visible: Boolean;
                             value: Integer; min: Integer; max: Integer;
                             procID: Integer; refCon: LongInt)
                             : ControlHandle;
```

Drawing Controls

```
{UpdateControls is also spelled as UpdtControl}
PROCEDURE ShowControl       (theControl: ControlHandle);
PROCEDURE UpdateControls    (theWindow: WindowPtr; updateRgn: RgnHandle);
PROCEDURE DrawControls      (theWindow: WindowPtr);
PROCEDURE Draw1Control      (theControl: ControlHandle);
```

Handling Mouse Events in Controls

```
FUNCTION FindControl        (thePoint: Point; theWindow: WindowPtr;
                             VAR theControl: ControlHandle): Integer;
FUNCTION TrackControl       (theControl: ControlHandle; thePoint: Point;
                             actionProc: ProcPtr): Integer;
FUNCTION TestControl        (theControl: ControlHandle; thePt: Point)
                             : Integer;
```

Changing Control Settings and Display

```
{some routines have 2 spellings——see Table 5-1 for the alternate spellings}
PROCEDURE SetControlValue   (theControl: ControlHandle; theValue: Integer);
PROCEDURE SetControlMinimum (theControl: ControlHandle; minValue: Integer);
PROCEDURE SetControlMaximum (theControl: ControlHandle; maxValue: Integer);
PROCEDURE SetControlTitle   (theControl: ControlHandle; title: Str255);
PROCEDURE HideControl       (theControl: ControlHandle);
PROCEDURE MoveControl       (theControl: ControlHandle; h: Integer;
                             v: Integer);
PROCEDURE SizeControl       (theControl: ControlHandle; w: Integer; h:
                             Integer);
PROCEDURE HiliteControl     (theControl: ControlHandle;
                             hiliteState: Integer);
PROCEDURE DragControl       (theControl: ControlHandle; startPt: Point;
                             limitRect: Rect; slopRect: Rect;
                             axis: Integer);
PROCEDURE SetControlColor   (theControl: ControlHandle; newColorTable:
                             CCTabHandle);
PROCEDURE SetControlAction  (theControl: ControlHandle;
                             actionProc: ProcPtr);
```

Determining Control Values

```
{some routines have 2 spellings——see Table 5-1 for the alternate spellings}
FUNCTION GetControlValue    (theControl: ControlHandle): Integer;
FUNCTION GetControlMinimum  (theControl: ControlHandle): Integer;
FUNCTION GetControlMaximum  (theControl: ControlHandle): Integer;
PROCEDURE GetControlTitle   (theControl: ControlHandle; VAR title: Str255);
FUNCTION GetControlReference
                            (theControl: ControlHandle): LongInt;
PROCEDURE SetControlReference
                            (theControl: ControlHandle; data: LongInt);
FUNCTION GetControlAction   (theControl: ControlHandle): ProcPtr;
FUNCTION GetControlVariant  (theControl: ControlHandle): Integer;
FUNCTION GetAuxiliaryControlRecord
                            (theControl: ControlHandle;
                             VAR acHndl: AuxCtlHandle): Boolean;
```

Removing Controls

```
PROCEDURE DisposeControl    (theControl: ControlHandle);
PROCEDURE KillControls      (theWindow: WindowPtr);
```

Application-Defined Routines

Defining Your Own Control Definition Function

```
FUNCTION MyControl          (varCode: Integer; theControl: ControlHandle;
                             message: Integer; param: LongInt) : LongInt;
```

Defining Your Own Action Procedures

```
PROCEDURE MyAction          (theControl: ControlHandle; partCode: Integer);
PROCEDURE MyIndicatorAction;
```

C Summary

Constants

```
enum {
      /*control definition IDs*/
      pushButProc          = 0,     /*button*/
      checkBoxProc         = 1,     /*checkbox*/
      radioButProc         = 2,     /*radio button*/
      useWFont             = 8,     /*add to above to display control */
                                    /* title in the window font*/
      scrollBarProc        = 16,    /*scroll bar*/
      popupMenuProc        = 1008,  /*pop-up menu*/

      /*pop-up menu CDEF variation codes*/
      popupFixedWidth  = 1 << 0,    /*add to popupMenuProc to use */
                                    /* use fixed-width control*/
      popupUseAddResMenu = 1 << 2,  /*add to popupMenuProc to specify a */
                                    /* value of type ResType in the */
                                    /* contrlRfCon field of the control */
                                    /* record; Menu Manager adds */
                                    /* resources of this type to the menu*/
      popupUseWFont  = 1 << 3       /*add to popupMenuProc to display */
                                    /* control title in the window font*/
};
```

```
enum {
      /*part codes*/
      inButton             = 10, /*button*/
      inCheckBox           = 11, /*checkbox or radio button*/
      inUpButton           = 20, /*up arrow for a vertical scroll bar, */
                                 /* left arrow for a horizontal scroll bar*/
      inDownButton         = 21, /*down arrow for a vertical scroll bar, */
                                 /* right arrow for a horizontal scroll bar*/
      inPageUp             = 22, /*gray area above scroll box for a */
                                 /* vertical scroll bar, gray area to */
                                 /* left of scroll box for a horizontal */
                                 /* scroll bar*/
      inPageDown           = 23, /*gray area below scroll box for a */
                                 /* vertical scroll bar, gray area to */
                                 /* right of scroll box for a horizontal */
                                 /* scroll bar*/
      inThumb              = 129 /*scroll box (or other indicator)*/
};

enum {
      /*pop-up title characteristics*/
      popupTitleBold  = 1 << 8,        /*boldface font style*/
      popupTitleItalic = 1 << 9,       /*italic font style*/
      popupTitleUnderline = 1 << 10,   /*underline font style*/
      popupTitleOutline = 1 << 11,     /*outline font style*/
      popupTitleShadow = 1 << 12,      /*shadow font style*/
      popupTitleCondense = 1 << 13,    /*condensed text*/
      popupTitleExtend = 1 << 14,      /*extended text*/
      popupTitleNoStyle = 1 << 15      /*monostyled text*/
};

enum {
      /*pop-up title characteristics*/
      popupTitleLeftJust = 0x00000000,    /*place title left of pop-up box*/
      popupTitleCenterJust = 0x00000001,  /*center title over pop-up box*/
      popupTitleRightJust = 0x000000FF,   /*place title right of pop-up box*/

   /*axis constraints for DragControl procedure*/
      noConstraint   = 0,  /*no constraint*/
      hAxisOnly      = 1,  /*constrain movement to horizontal axis only*/
      vAxisOnly      = 2,  /*constrain movement to vertical axis only*/
/*constants for the message parameter in a control definition function*/
      drawCntl       = 0,  /*draw the control or control part*/
      testCntl       = 1,  /*test where mouse button was pressed*/
      calcCRgns      = 2,  /*calculate region for control or indicator in */
                           /* 24-bit systems*/
      initCntl       = 3,  /*do any additional control initialization*/
      dispCntl       = 4,  /*take any additional disposal actions*/
      posCntl        = 5,  /*move indicator and update its setting*/
      thumbCntl      = 6,  /*calculate parameters for dragging indicator*/
      dragCntl       = 7,  /*peform any custom dragging of control or */
                           /* its indicator*/
      autoTrack      = 8,  /*execute action procedure specified by your */
                           /* function*/
      calcCntlRgn    = 10, /*calculate region for control*/
      calcThumbRgn   = 11, /*calculate region for indicator*/

/*part identifiers for ColorSpec records in a control color table resource*/
      cFrameColor    = 0,  /*frame color; for scroll bars, also foreground */
                           /* color for scroll arrows and gray area*/
      cBodyColor     = 1,  /*for scroll bars, background color for scroll */
                           /* arrows and gray area; for other controls, */
                           /* the fill color for body of control*/
      cTextColor     = 2,  /*text color; for scroll bars, unused*/
      cThumbColor    = 3   /*Reserved*/
};
```

Data Types

```
struct ControlRecord {
      struct ControlRecord **nextControl;    /*next control*/
      WindowPtr      contrlOwner;   /*control's window*/
      Rect           contrlRect;    /*rectangle*/
      unsigned char  contrlVis;     /*255 if visible*/
      unsigned char  contrlHilite;  /*highlight state*/
      short          contrlValue;   /*control's current setting*/
      short          contrlMin;     /*control's minimum setting*/
      short          contrlMax;     /*control's maximum setting*/
      Handle         contrlDefProc; /*control definition function*/
      Handle         contrlData;    /*data used by contrlDefProc*/
      ProcPtr        contrlAction;  /*action procedure*/
      long           contrlRfCon;   /*control's reference value*/
      Str255         contrlTitle;   /*control's title*/
   };
```

```
typedef struct ControlRecord ControlRecord;
typedef ControlRecord *ControlPtr, **ControlHandle;

struct AuxCtlRec {
      Handle         acNext;        /*handle to next AuxCtlRec*/
      ControlHandle  acOwner;       /*handle to this record's control*/
      CCTabHandle    acCTable;      /*handle to color table record*/
      short          acFlags;       /*reserved*/
      long           acReserved;    /*reserved for future use*/
      long           acRefCon;      /*for use by application*/
   };

typedef struct AuxCtlRec AuxCtlRec;
typedef AuxCtlRec *AuxCtlPtr, **AuxCtlHandle;

struct CtlCTab {
      long           ccSeed;        /*reserved; set to 0*/
      short          ccRider;       /*reserved; set to 0*/
      short          ctSize;        /*number of ColorSpec records in next */
                                    /* field; 3 for standard controls*/
      ColorSpec      ctTable[4];
   };
typedef struct CtlCTab CtlCTab;
typedef CtlCTab *CCTabPtr, **CCTabHandle;
```

Control Manager Routines

Creating Controls

```
pascal ControlHandle GetNewControl
                             (short controlID, WindowPtr owner);
pascal ControlHandle NewControl
                             (WindowPtr theWindow, const Rect *boundsRect,
                              ConstStr255Param title, Boolean visible,
                              short value, short min, short max,
                              short procID, long refCon);
```

Drawing Controls

```
/*UpdateControls is also spelled as UpdtControl*/
pascal void ShowControl      (ControlHandle theControl);
pascal void UpdateControls   (WindowPtr theWindow, RgnHandle updateRgn);
pascal void DrawControls     (WindowPtr theWindow);
pascal void Draw1Control     (ControlHandle theControl);
```

Handling Mouse Events in Controls

```
pascal short FindControl    (Point thePoint, WindowPtr theWindow,
                             ControlHandle *theControl);
pascal short TrackControl   (ControlHandle theControl, Point thePoint,
                             ProcPtr actionProc);
pascal short TestControl    (ControlHandle theControl, Point thePt);
```

Changing Control Settings and Display

```
/*some routines have 2 spellings——see Table 5-1 for the alternate spellings*/
pascal void SetControlValue (ControlHandle theControl, short theValue);
pascal void SetControlMinimum
                            (ControlHandle theControl, short minValue);
pascal void SetControlMaximum
                            (ControlHandle theControl, short maxValue);
pascal void SetControlTitle (ControlHandle theControl,
                             ConstStr255Param title);
pascal void HideControl     (ControlHandle theControl)
pascal void MoveControl     (ControlHandle theControl, short h, short v);
pascal void SizeControl     (ControlHandle theControl, short w, short h);
pascal void HiliteControl   (ControlHandle theControl, short hiliteState);
pascal void DragControl     (ControlHandle theControl, Point startPt,
                             const Rect *limitRect,
                             const Rect *slopRect, short axis);
pascal void SetControlAction(ControlHandle theControl, ProcPtr actionProc)
pascal void SetControlColor (ControlHandle theControl,
                             CCTabHandle newColorTable);
```

Determining Control Values

```
/*some routines have 2 spellings——see Table 5-1 for the alternate spellings*/
pascal short GetControlValue
                            (ControlHandle theControl);
pascal short GetControlMinimum
                            (ControlHandle theControl);
pascal short GetControlMaximum
                            (ControlHandle theControl);
pascal void GetControlTitle (ControlHandle theControl, Str255 title);
pascal long GetControlReference
                            (ControlHandle theControl);
pascal void SetControlReference
                            (ControlHandle theControl, long data);
pascal ProcPtr GetControlAction
                             (ControlHandle theControl);
```

```
pascal short GetControlVariant
                            (ControlHandle theControl);
pascal Boolean GetAuxiliaryControlRecord
                            (ControlHandle theControl,
                             AuxCtlHandle *acHndl);
```

Removing Controls

```
pascal void DisposeControl  (ControlHandle theControl);
pascal void KillControls    (WindowPtr theWindow);
```

Application-Defined Routines

Defining Your Own Control Definition Function

```
pascal long MyControl       (short varCode, ControlHandle theControl,
                             short message, long param);
```

Defining Your Own Action Procedures

```
pascal void MyAction        (ControlHandle theControl, short partCode);
pascal void MyIndicatorAction;
```

Assembly-Language Summary

Data Structures

ControlRecord Data Structure

0	`nextControl`	long	handle to next control in control list
4	`contrlOwner`	long	pointer to this control’s window
8	`contrlRect`	8 bytes	control’s rectangle
16	`contrlVis`	1 byte	value of 255 if control is visible
17	`contrlHilite`	1 byte	highlight state
18	`contrlValue`	word	control’s current setting
20	`contrlMin`	word	control’s minimum setting
22	`contrlMax`	word	control’s maximum setting
24	`contrlDefProc`	long	handle to control definition function
28	`contrlData`	long	data used by control definition function
32	`contrlAction`	long	address of action procedure
36	`contrlRfCon`	long	control’s reference value
40	`contrlTitle`	256 bytes	control title (preceded by length byte)

AuxCtlRec Data Structure

0	`acNext`	long	handle to next `AuxCtlRec` record in control list
4	`acOwner`	long	handle to this record's control
8	`acCTable`	long	handle to color table for this control
12	`acFlags`	word	miscellaneous flags
14	`acReserved`	long	reserved for use by Apple Computer, Inc.
18	`acRefCon`	long	for use by application

Global Variables

`AuxCtlHead`	First in a linked list of auxiliary control records
`AuxWinHead`	Contains a pointer to the linked list of auxiliary control records
`DragHook`	Address of procedure to execute during `TrackControl` and `DragControl`
`DragPattern`	Pattern of dragged region's outline (8 bytes)

C H A P T E R 6

Dialog Manager

Contents

This chapter describes how your application can use the Dialog Manager to alert users to unusual situations and to solicit information from users. For example, in some situations your application might not be able to carry out a command normally, and in other situations the user must specify multiple parameters before your application can execute a command. For circumstances like these, the Macintosh user interface includes these two features:

- **alerts**—including alert sounds and alert boxes—which warn the user whenever an unusual or potentially undesirable situation occurs within your application
- **dialog boxes,** which allow the user to provide additional information or to modify settings before your application carries out a command

Read this chapter to learn how and when to implement alerts and dialog boxes. For example, your application can use the Dialog Manager to ask the user whether to save new or altered documents before quitting and, if the situation arises, to inform the user that there is insufficient disk space to save the file.

Virtually all applications need to implement alerts and dialog boxes. To avoid needless development effort, use the Dialog Manager to implement alerts and to create most dialog boxes. It is possible, however—and sometimes desirable—to bypass the Dialog Manager and instead use Window Manager, Control Manager, QuickDraw, and Event Manager routines to create or respond to events in complex dialog boxes. Even if you decide not to use the Dialog Manager, read this chapter for information about effective human interface design and localization issues regarding dialog boxes.

To use this chapter, you should be familiar with resources, the Event Manager, the Window Manager, and the Control Manager.

You typically use resources to specify the items you wish to display in alert boxes and dialog boxes; for example, you specify the size, location, and appearance of a dialog box in a dialog resource—a resource of type `'DLOG'`. See the chapter “Introduction to the Macintosh Toolbox” in this book for general information about resources; detailed information about the Resource Manager and its routines is provided in the chapter “Resource Manager” of *Inside Macintosh: More Macintosh Toolbox*.

The Dialog Manager offers routines that handle most of the events relating to alerts and dialog boxes, but your application still needs to handle a few additional events as described in “Writing an Event Filter Function for Alert and Modal Dialog Boxes” beginning on page 6-86. See the chapter “Event Manager” in this book for general information about events and event handling.

The Dialog Manager uses the Window Manager to display your alert boxes and dialog boxes. Although the Dialog Manager uses most of the Window Manager routines necessary to activate and update your alert and dialog boxes, your application needs to use Window Manager routines if it creates certain types of dialog boxes—such as modeless dialog boxes—as explained in this chapter. See the chapter “Window Manager” in this book for general information about windows.

The Dialog Manager uses the Control Manager to create and display buttons, radio buttons, checkboxes, and pop-up menus and to handle events in them. Generally, you shouldn’t use any other controls—such as scroll bars—in your dialog boxes. If you need to implement a more complex control, see the chapter “Control Manager” in this book. Buttons are the only controls you should use in alert boxes.

If you include editable text items in your dialog boxes, the Dialog Manager uses TextEdit to handle associated editing tasks. For general information on TextEdit, see the chapter “TextEdit” in *Inside Macintosh: Text*.

This chapter provides a brief introduction to the concepts and functions of alerts and dialog boxes, and then it discusses how you can

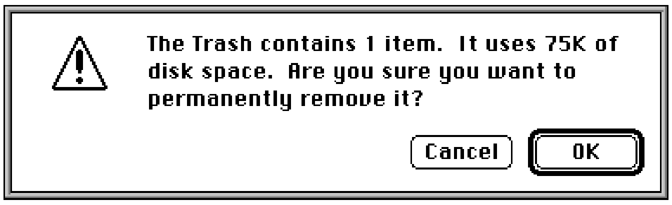
- create and display alerts and dialog boxes
- include controls, informative text, editable text fields, and similar items in your alert boxes and dialog boxes
- respond to events in your alert boxes and dialog boxes

Introduction to Alerts and Dialog Boxes

The behaviors and uses of alerts differ from those of dialog boxes. Important distinctions also exist between different types of alerts and between different types of dialog boxes. You choose among these according to the user’s current situation.

Your application should give an alert to report an error or to issue a warning to the user. An alert can simply play a sound (called an **alert sound**) for the user, it can display an alert box that contains a message and requires an acknowledgment from the user, or it can play an alert sound and simultaneously display an alert box. **Alert boxes** are special windows that contain informative text, buttons, and, generally, icons. They may also contain pictures. As shown in Figure 6-1, an alert box typically consists of text describing why the alert appears and buttons requiring the user to acknowledge or rectify the problem.

Figure 6-1 An alert box used by the Finder

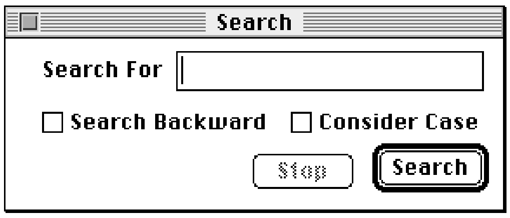

The Trash contains 1 item. It uses 75K of disk space. Are you sure you want to permanently remove it?

By requiring the user to click a button, an alert box obliges the user to acknowledge the alert box before proceeding. To assist the user who isn’t sure how to respond when an alert box appears, your application specifies a preferred button—which invokes a preferred action—for every alert box. The Dialog Manager draws a bold outline around the preferred button so that it stands out from the other buttons in the alert box. The outlined button is also the alert box’s **default button**; if the user presses the Return key or the Enter key, the Dialog Manager acts as if the user had clicked this preferred button. For example, if the user presses the Return or Enter key in response to the alert box shown in Figure 6-1, the Dialog Manager inverts the OK button for 8 ticks and informs the Finder that the OK button has been selected; then the Finder responds by deleting the item contained in the Trash.

Use a dialog box when your application needs more information to carry out a command. Commands in menus normally act on only one object. If the user chooses a command that your application cannot perform until the user supplies more information, use a dialog box to elicit the information from the user. If a command brings up a dialog box, indicate this to your user by placing three ellipsis points (...) after the command's name in the menu.

A dialog box is a special window that typically resembles a form on which the user checks boxes and fills in blanks. Figure 6-2 shows a typical dialog box.

Figure 6-2 A typical dialog box



Although an alert typically requires only an acknowledgment to proceed from the user, a dialog box ordinarily requires the user to supply information—for instance, by entering text or by clicking a checkbox—necessary for completing the command. When you create a dialog box that carries out a command, you normally provide OK and Cancel buttons. When the user clicks the OK button, your application should perform the command according to the information that the user supplied in the dialog box. When the user clicks the Cancel button, your application should revoke the command and retract all of its actions as though the user had never given the command. Instead of using an OK button, you might use a button that describes the action to be performed; for example, you might use a Search button in a Search command's dialog box or a Remove button in a Remove command's dialog box. For simplicity, this chapter refers to the button that performs the action described in the dialog box as the *OK button*. You may even provide more than one button that performs the command, each in a slightly different way. For example, in a Change command's dialog box, you might include a Change Selection button to replace only the current selection and a Change All button to replace all occurrences throughout the entire document.

You can use any or all of the following elements in the dialog boxes you create:

- informative or instructional text
- rectangles in which text may be entered (initially blank or containing default text that can be edited)

- controls
- graphics (icons or QuickDraw pictures)
- other items as defined by your application

Types of Alerts

Every user of every application is liable to do something that the application won't understand or can't cope with in a normal manner. Alerts give your application a way to respond to these situations in a consistent manner. There are two major categories of alerts: alert sounds and alert boxes.

The **system alert sound** is a sound resource stored in the System file. This sound is played whenever system software or your application uses the Sound Manager procedure `SysBeep`. The Sound control panel allows the user to select which sound is played as the system alert sound. You can also provide your own alert sound to use in place of the system alert sound.

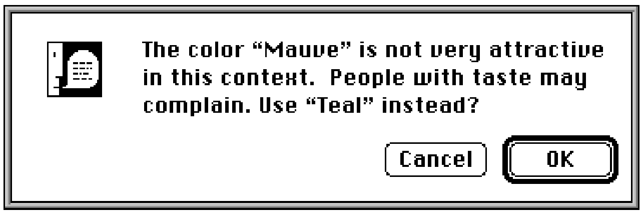
Use an alert sound for errors that are both minor and immediately obvious. For example, if the user tries to backspace past the left boundary of a text field, your application might play the alert sound instead of displaying an alert box. Your application can base its response on the number of consecutive times an alert condition recurs; the first time, your application might simply play a sound, and thereafter it might present an alert box. Your application can define different responses for each one of four alert stages.

An alert box is primarily a one-way communication from your application to the user; the only way the user can respond is by clicking buttons. Therefore, your alert boxes should contain buttons, but usually they should not contain editable text fields, radio buttons, or checkboxes—items that are typically displayed in dialog boxes.

There are three standard kinds of alert boxes: note alerts, caution alerts, and stop alerts. They are distinguished by the icons displayed in their upper-left corners.

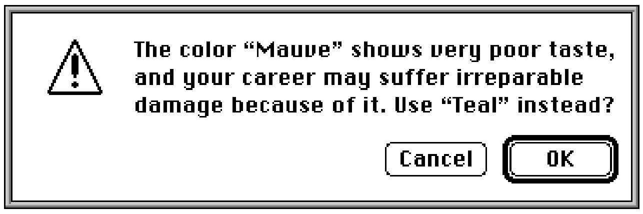
Use a **note alert** to inform users of a situation that won't have any disastrous consequences if left as is. Usually this type of alert simply offers information, and the user responds by clicking the OK button. Occasionally, as shown in Figure 6-3, a note alert may ask a simple question and provide a choice of responses.

Figure 6-3 A note alert


The color "Mauve" is not very attractive in this context. People with taste may complain. Use "Teal" instead?

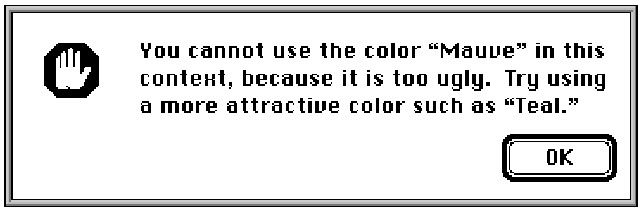
Use a **caution alert** to alert the user to an operation that may have undesirable results if it's allowed to continue. As shown in Figure 6-4, you should give the user the choice of whether to continue the action (by clicking the OK button) or to stop the action (by clicking the Cancel button).

Figure 6-4 A caution alert


The color "Mauve" shows very poor taste, and your career may suffer irreparable damage because of it. Use "Teal" instead?

Use a **stop alert** to inform the user that a problem or situation is so serious that the action cannot be completed. Stop alerts, as illustrated in Figure 6-5, typically have only a single button (OK), because all the user can do is acknowledge that the action cannot be completed.

Figure 6-5 A stop alert


You cannot use the color "Mauve" in this context, because it is too ugly. Try using a more attractive color such as "Teal."

You can also create **custom alert boxes** containing in the upper-left corners either your own icons or blank spaces. Plate 2 at the front of this book illustrates an alert box that the SurfWriter application displays when the user chooses the About command from the Apple menu. After reading the information in this alert box, the user clicks the OK button to dismiss it.

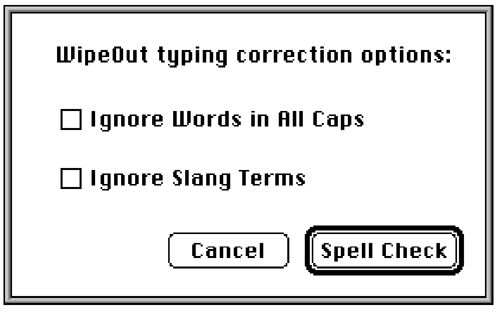
Types of Dialog Boxes

Dialog boxes should always require information from the user as well as communicate information to the user. That is, the purpose of a dialog box is to carry on a dialog between the user and your application—typically, in preparation for the execution of a command. Your dialog boxes can include editable text fields and controls such as checkboxes and radio buttons. With these, the user supplies the information your application needs to carry out the command. There are three types of dialog boxes: modal dialog boxes, movable modal dialog boxes, and modeless dialog boxes. These are described in the next three sections.

Modal Dialog Boxes

Before allowing the user to proceed with any other work, many dialog boxes require the user to click a button. The only response a user receives when clicking outside the dialog box is an alert sound. This type is called a **modal dialog box** because it puts the user in the state or "mode" of being able to work only inside the dialog box. Also called a fixed-position modal dialog box (to differentiate it from a movable modal dialog box), this type of dialog box looks like an alert box that includes other types of controls in addition to buttons. Figure 6-6 shows the modal dialog box that SurfWriter displays after the user chooses the Spell Check command.

Figure 6-6 A modal dialog box


WipeOut typing correction options:
☐ Ignore Words in All Caps
☐ Ignore Slang Terms
Cancel
Spell Check

IMPORTANT

Because the user must explicitly dismiss a modal dialog box before doing anything else, you should use a modal dialog box only when it's essential for the user to complete an operation before performing any other work. Fixed-position modal dialog boxes restrict the user's freedom of action; therefore, use them sparingly. As a rule of thumb, use a modeless dialog box whenever possible, use a movable modal dialog box whenever you can't use a modeless dialog box, and use a fixed-position modal dialog box only when you can't implement the dialog box as modeless or movable. s

A modal dialog box usually has at least two buttons: OK and Cancel. When the user clicks the OK button, your application should perform the command according to the information provided by the user and then remove the modal dialog box. You can give the OK button a more descriptive title if you wish. When the user clicks the Cancel button, your application should revoke any actions it took since displaying the modal dialog box, and then it should remove the modal dialog box. *Always* label this button "Cancel." Your dialog boxes can have additional buttons as well; these may or may not dismiss the dialog box.

Every dialog box you create should have a default button—that is, one whose action is invoked when the user presses the Return or Enter key. Unless you provide your own event filter function, the Dialog Manager treats the first item you specify in a description of a dialog box as the default button (that is, so long as the first item is a button). You use an **event filter function,** described in “Writing an Event Filter Function for Alert and Modal Dialog Boxes” beginning on page 6-86, to supplement the Dialog Manager’s ability to handle events; for example, an event filter function can also test for disk-inserted events and can allow background applications to receive update events. If you provide your own event filter function, it should test for key-down events involving the Return and Enter keys and respond as if the default button were clicked. The default button should invoke the preferred action, and you should try to design the preferred action to be safe—that is, so that it doesn’t cause loss of data.

Although the Dialog Manager draws bold outlines around default buttons in alert boxes, it does not draw bold outlines around those in dialog boxes. To indicate the preferred action, your application should outline the default button. “Using an Application-Defined Item to Draw the Bold Outline for a Default Button” beginning on page 6-56 shows a method you can use to outline a button. If you don’t outline a button in a dialog box, none should be the default button, and you must ensure in your event filter function that pressing the Return or Enter key has no effect.

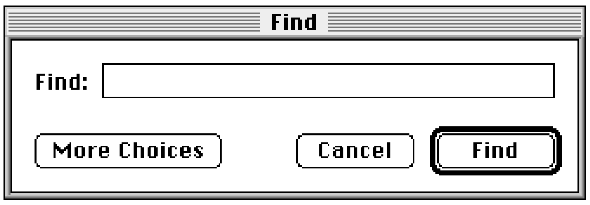
Movable Modal Dialog Boxes

The user sometimes needs to see windows obscured by an overlying modal dialog box. In this case, you should use a movable modal dialog box instead of a fixed-position modal dialog box. The **movable modal dialog box** is a modal dialog box that has a title bar so that the user can move the box by dragging its title bar.

The movable modal dialog box contains no close box and should contain no zoom box. These visual clues indicate that the user can move the dialog box, but that the dialog box is modal—that is, the user must respond to the dialog box before performing any other work in your application. If the user clicks another window belonging to your application, it should play the system alert sound. Your application removes a movable modal dialog box only after the user clicks one of its buttons. Unlike regular modal dialog boxes, however, this type of dialog box allows the user to bring another application to the front by clicking one of its windows or by choosing the application name from the Application or Apple menu.

Figure 6-7 shows the movable modal dialog box that the Finder displays after the user chooses the Find command from the File menu.

Figure 6-7 A movable modal dialog box



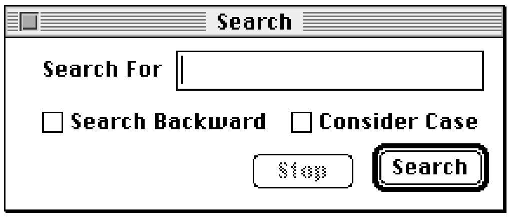
It’s important to consider whether you can use a modeless dialog box instead of a modal or a movable modal dialog box—especially to preserve the user’s ability to perform any task in any order.

Movable modal dialog boxes should generally respond like modal dialog boxes. Note, however, that users should be able to switch between your application and another application (thereby sending your application to the background) when you display a movable modal dialog box—an action users cannot perform with modal dialog boxes. For example, Macintosh system software uses several movable modal dialog boxes to show that the Finder is busy with a time-consuming operation (such as file copying), yet a user can still switch the Finder to the background.

Modeless Dialog Boxes

Other dialog boxes do not require the user to respond before doing anything else; these are called **modeless dialog boxes.** Whenever possible, you should try to implement your dialog boxes as modeless. As shown in Figure 6-8, a modeless dialog box looks like a document window. The user should be able to move it, make it inactive and active again, and close it like any document window. Unlike a document window, it consists mostly of buttons and other controls instead of text, and it contains no scroll bars and no size box. (A modeless dialog box should not have a size box or scroll bars; if you need these features, use the Window Manager to create a window.)

Figure 6-8 A modeless dialog box



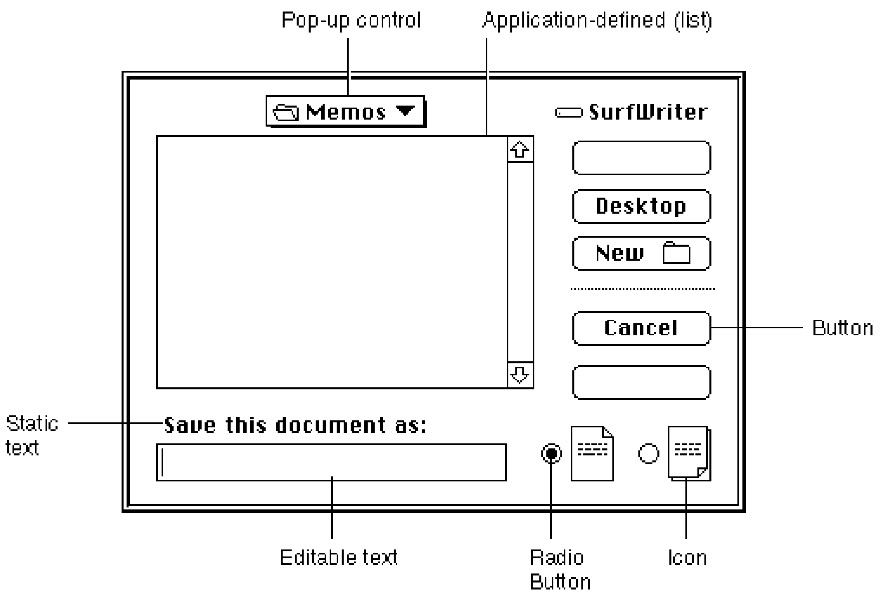
When you display a modeless dialog box, you must allow the user to perform other operations—such as working in document windows—without dismissing the dialog box. When a user clicks a button in a modeless dialog box, your application should *not* remove it; instead, the dialog box should remain on the desktop so that the user can perform the command again. Because of the difficulty in revoking the last action invoked from a modeless dialog box, it typically does not have a Cancel button, although it may have a Stop button. A Stop button in a modeless dialog box is useful for halting long printing or searching operations, for example.

When finished with a modeless dialog box, the user can click its close box or choose Close from the File menu (when the dialog box is the active window). Your application should then remove the modeless dialog box. A modeless dialog box is also dismissed implicitly when the user chooses Quit. It’s usually helpful to the user for your application to remember the contents of the dialog box after it’s dismissed. This way, when the user invokes the dialog box again, even after the user closes and reopens your application, you can restore the dialog box exactly as it was.

Items in Alert and Dialog Boxes

All dialog boxes and alert boxes contain items—such as icons, text, controls, and QuickDraw pictures. You use resources called **item lists** to specify which items you want to appear in your alert boxes and dialog boxes. You can even define your own items—for example, a picture whose appearance changes. Figure 6-9 illustrates most of these item types.

Figure 6-9 Typical items in a dialog box



Your application enables or disables the items it includes in its dialog and alert boxes. An **enabled item** is one for which the Dialog Manager reports user events involving that item; for example, the Dialog Manager reports to the application when a user clicks the enabled Cancel button shown in Figure 6-9. A **disabled item** is one for which the Dialog Manager does not report events. For example, the Dialog Manager does not report to the application when the user clicks or drags the static text item "Save this document as" in Figure 6-9 because that item is disabled.

Don't confuse a *disabled item* with an *inactive control.* When you don't want the Control Manager to display visual responses to mouse events in a control, you make it inactive by using the Control Manager procedure `HiliteControl`. For example, until the user types a filename, the Save button in Figure 6-9 is inactive. The Control Manager displays an inactive control in a way (such as by dimming it) that shows it's inactive. The Dialog Manager makes no visual distinction between a disabled item and an enabled item; the Dialog Manager simply doesn't inform your application when the user clicks a disabled item.

You should use `HiliteControl` to dim a control in dialog box whenever the user can't use that control. For example, Figure 6-8 shows a modeless dialog box with a dimmed

Stop button. The Stop button is dimmed because it has no effect until the user clicks the Search button. When the user initiates the search operation by clicking the Search button, the Stop button becomes active, and the Search button is dimmed.

You should use the Control Manager procedure `HiliteControl` to make the buttons and other controls inactive in a modeless or movable modal dialog box when you deactivate it. The `HiliteControl` procedure dims inactive buttons, radio buttons, checkboxes, and pop-up menus to indicate to the user that clicking these items has no effect while the dialog box is in the background. When you activate the dialog box again, use `HiliteControl` to make the controls active again.

You store information about all dialog or alert box items in an item list resource. When you use Dialog Manager routines to invoke alert boxes or create dialog boxes, the Dialog Manager gets most of the descriptive information about them from resources. The Dialog Manager calls the Resource Manager to read into memory what it needs from the resource file.

Events in Alert and Dialog Boxes

Handling events in an alert box is very simple: after you invoke an alert box, the Dialog Manager handles most events for you by automatically calling the `ModalDialog` procedure.

To handle events in a modal dialog box, your application must explicitly call the `ModalDialog` procedure after displaying the dialog box.

In either case, when an enabled item is clicked, the Dialog Manager returns the item number. You'll then do whatever is appropriate in response to that click. For mouse-down events outside the alert box or modal dialog box, the `ModalDialog` procedure plays the system alert sound and gets the next event.

The Dialog Manager automatically removes an alert box when the user clicks any enabled item. For a modal dialog box, your application should continue calling `ModalDialog` until the user selects the OK or Cancel button, and then—after responding appropriately to the user's selection—your application should remove the dialog box.

When it receives an event, `ModalDialog` passes the event to an event filter function before handling the event itself. You should provide an event filter function as a secondary event-handling loop for events that `ModalDialog` doesn't handle. For both alert and modal dialog boxes, you should provide a simple event filter function that performs the following tasks:

- return TRUE and the item number for the default button if the user presses the Return or Enter key
- return TRUE and the item number for the Cancel button if the user presses the Esc key or the Command-period key combination
- update your windows in response to update events (this also allows background applications to receive update events) and return FALSE
- return FALSE for all events that your event filter function doesn't handle

You can also use the event filter function to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor within an application-defined item.

For your application's modeless and movable modal dialog boxes, you can pass events to the `IsDialogEvent` function, or you can use your own event-handling code to learn whether the events need to be handled as part of a dialog box. If they do, call the `DialogSelect` function to assist you in handling them instead of calling the `ModalDialog` procedure. Your application should not remove a modeless dialog box unless the user clicks its close box or chooses Close from the File menu when the modeless dialog box is the active window. Your application should remove a movable modal dialog box only after the user clicks one of its enabled buttons.

Instead of using the `IsDialogEvent` or `DialogSelect` function to handle events within modeless and movable modal dialog boxes, you can use Control Manager, Window Manager, and TextEdit routines (such as `FindWindow`, `BeginUpdate`, `EndUpdate`, `FindControl`, `TrackControl`, and `TEClick`) to handle these events without the aid of the Dialog Manager.

Alert Boxes, Dialog Boxes, and the Window Manager

The Dialog Manager uses the Window Manager to draw your alert boxes and dialog boxes. You can use Window Manager or QuickDraw routines to manipulate an alert box or a dialog box just like any other window—showing it, hiding it, moving it, and resizing it.

The Dialog Manager gets most of the descriptive information about alerts and dialog boxes from resources in a resource file. An **alert resource** is a resource that describes an alert, and a **dialog resource** is a resource that describes a dialog box. Both are analogous to a window resource. (In addition to providing information that the Dialog Manager passes to the Window Manager, you also include in your alert resources and dialog resources additional information that the Dialog Manager alone uses. These resources are described more fully in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 and "Creating Dialog Boxes" beginning on page 6-23.)

When you create an alert box, the Dialog Manager always passes to the Window Manager the `dBoxProc` window definition ID for the alert box; this is so that all alert boxes have the same standard appearance and behavior. The Window Manager always displays an alert box in front of all other windows. Because an alert box requires the user to respond before doing anything else, and the response dismisses the alert box, your application typically won't need to use any Window Manager or QuickDraw routines to manipulate an alert box.

The `GetNewDialog` function for creating dialog boxes is similar to the Window Manager function `GetNewWindow`. When you call `GetNewDialog` to create a dialog box, you supply the same information as when you create a window with `GetNewWindow`. For example, you use a resource to specify the window definition ID, which determines how the dialog box looks and behaves, and a rectangle that defines the dimensions of the dialog box's graphics port. As for any window, you specify the plane of the dialog box (which, by convention, should initially be frontmost), and you specify whether it is initially visible or invisible. If you create a dialog box that is initially invisible—for example, if you need to set a control's value before displaying it—you use the Window Manager procedure `ShowWindow` to display the dialog box.

The Dialog Manager creates the dialog window by calling the Window Manager function `NewCWindow` and then setting the window class in the window record to indicate that it's a dialog box. The Dialog Manager procedures for disposing of a dialog box, `CloseDialog` and `DisposeDialog`, are analogous to the Window Manager procedures `CloseWindow` and `DisposeWindow`.

When you create a dialog box (as described in "Creating Dialog Boxes" beginning on page 6-23), use the window definition ID of `dBoxProc` for modal dialog boxes. Use the `noGrowDocProc` window definition ID for modeless dialog boxes. (If your dialog box absolutely needs a size box or scroll bars, you should use the Window Manager to create the window instead of using the Dialog Manager.) And finally, use the `movableDBoxProc` window definition ID to create movable modal dialog boxes.

The Dialog Manager provides routines for handling most events in alert boxes and dialog boxes. For example, your application does not need to use such routines as the Window Manager function `FindWindow` and the Control Manager function `TrackControl` to determine when and where a mouse-down event occurs within an alert box's buttons. The Dialog Manager tells you which button the user clicks, and your application needs only to respond appropriately to the click. The Dialog Manager also automatically handles update and activate events for your alert boxes and dialog boxes. "Handling Events in Alert and Dialog Boxes" beginning on page 6-77 describes in detail how to use the Dialog Manager to help your application handle events.

About the Dialog Manager

The Dialog Manager greatly simplifies the task of creating alert boxes and simple modal dialog boxes. Whenever you need to create an alert box, you'll save yourself much effort by relying on the Dialog Manager. (If you need only to play the system alert sound without ever displaying an alert box for an error condition, you can use the Sound Manager procedure `SysBeep` instead of using the Dialog Manager. See *Inside Macintosh: Sound* for more information about the `SysBeep` procedure.)

You may find, however, that the advantages of using the Dialog Manager begin to diminish for dialog boxes if you make them very complex. For complex modal dialog boxes (particularly those containing multipart controls or multiple application-defined items) and for many movable modal and modeless dialog boxes, you may find it more convenient to implement your own dialog boxes using the Window Manager to create standard windows and using the Control Manager, QuickDraw, and the Event Manager to handle the tasks assumed by the Dialog Manager.

There are two main issues to consider when deciding whether to use the Dialog Manager:

- whether to use the Window Manager and the Control Manager instead of the Dialog Manager to create a dialog box

- whether to use the Event Manager, Window Manager, Control Manager, and TextEdit instead of the Dialog Manager to handle events

You may, for example, want to create complex dialog boxes by using the Dialog Manager, but then use the Event Manager, Window Manager, Control Manager, and TextEdit to handle events inside your normal event loop. With regard to movable modal and modeless dialog boxes, the sample code in this chapter illustrates such a hybrid approach: it uses the Dialog Manager to create the dialog boxes, but it uses normal event-handling code to determine an appropriate action according to which type of window is frontmost. When a modeless or movable modal dialog box is in front, this chapter illustrates how to take actions specific to that dialog box.

If you draw your own dialog box in a standard window without using the Dialog Manager, you won't be able to use Dialog Manager routines to help handle events, but in return you'll be able to update the window more quickly and extend its event handling more easily. Here are some situations that tend to diminish the advantages of using the Dialog Manager to create dialog boxes or handle events involving them:

- The dialog box contains more than 20 items.
- You need a multipart control, such as a scroll bar.
- You need to move items offscreen and onscreen.
- You need to display a moving indicator, such as a progress indicator.
- You need to display a list in the dialog box. (For more information on lists, see the chapter "List Manager" in *Inside Macintosh: More Macintosh Toolbox*.)
- You need to display text in a font other than the system font.
- Your application must respond to events other than mouse-down events, key-down events inside editable text items, and a few key-down events for keyboard equivalents when your application displays the dialog box.

If none of these situations applies to the dialog box you want to create, then you should definitely use the Dialog Manager. If only one situation applies, you should probably use the Dialog Manager. If two or more of these situations apply, you may find that it is better to create and manage a standard window that operates like a dialog box instead of using the Dialog Manager to create or manage it.

Using the Dialog Manager

You can use the Dialog Manager to

- alert users to critical situations
- carry on a dialog with users when your application needs their input

With Dialog Manager routines, you invoke alert boxes or create dialog boxes in windows whose contents are, in turn, managed by the Dialog Manager. The Dialog Manager automatically handles update events, activate events, cursor tracking, and most text-editing tasks for your alert and dialog boxes.

To implement alerts and dialog boxes, you generally

- create an alert resource or a dialog resource in a resource file
- create another resource to specify a list of items—such as controls, informative text, and pictures—to be displayed in the alert box or dialog box
- create and display the alert box or dialog box
- respond as appropriate to events relating to your alert or dialog box
- close the dialog box when you are finished with it (for alert boxes, the Dialog Manager automatically performs this for you)

These tasks are explained in greater detail in the rest of this chapter.

Before using the Dialog Manager, you must initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit, in that order. Then initialize the Dialog Manager by using the `InitDialogs` procedure.

The Dialog Manager uses the system alert sound for signaling the user during various alert stages. If you want to use alert sounds other than the system alert sound, write your own sound procedure (as illustrated in Listing 6-3 on page 6-22) and call the `ErrorSound` procedure to make it the current sound procedure.

If you want to display static text or editable text in a font other than the system font, you can use the `SetDialogFont` procedure. However, there are a number of caveats regarding this procedure. For descriptions of these caveats, see “Special Considerations” in the description of `SetDialogFont` on page 6-105.

System 7 and earlier versions of the Communications Toolbox add several new routines (namely, `AppendDITL`, `ShortenDITL`, and `CountDITL`) that make it easier for you to add items to, remove items from, and count the number of items in a dialog box. Before calling these routines, you should make sure that they are available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit field indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `AppendDITL`, `ShortenDITL`, and `CountDITL` are available.

```
CONST gestaltDITLExtPresent= 0;  {if this bit is set, then }
                                 { AppendDITL, ShortenDITL, }
                                 { & CountDITL are available}
```

The `Gestalt` function is described in the chapter “Gestalt Manager” of *Inside Macintosh: Operating System Utilities.*

Creating Alert Sounds and Alert Boxes

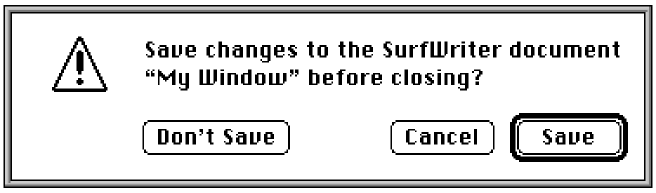
To create an alert, use one of these functions: `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert`. Icons associated with the first three functions appear in the upper-left corner of the alert boxes, as previously shown in Figure 6-3, Figure 6-4, and Figure 6-5. The `Alert` function allows you to display your own icon or to have no icon at all in the upper-left corner of the alert box.

These functions take descriptive information about the alert from an alert resource that you provide. An alert resource has the resource type `'ALRT'`. When you call one of these functions, you pass it the resource ID of the alert resource and a pointer to an event filter function. These functions create and display an alert box. When the user clicks a button in an alert box, these functions return the button's item number and close the alert box, at which time you respond appropriately to the user's click, as described in "Responding to Events in Alert Boxes" beginning on page 6-81.

Here's an example of how to create the caution alert shown in Figure 6-10.

```
VAR
   myAlertItem:      Integer;
myAlertItem := CautionAlert(kSaveAlertID, @MyEventFilter);
```

Figure 6-10 An alert box to save changes to a document


Save changes to the SurfWriter document
"My Window" before closing?

You should specify a pointer to an event filter function when you call the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. You should provide an event filter function as a secondary event-handling loop for events that `ModalDialog` doesn't handle. In this example, a pointer to `MyEventFilter` is specified for the event filter function. You can use the standard event filter function by passing `NIL` in this parameter. The standard event filter function allows users to press the Return or Enter key in lieu of clicking the default button. As described in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86, your application should provide a simple event filter function that also allows background applications to receive update events. You can use the same event filter function in most or all of your alert boxes and modal dialog boxes.

Continuing with the previous example, an application-defined constant (`kSaveAlertID`) specifies the resource ID of an alert resource in a parameter to the `CautionAlert` function. Listing 6-1 shows how this alert resource appears in Rez input format. (Rez is the resource compiler provided with Apple's Macintosh Programmer's Workshop [MPW], available from APDA.)

Listing 6-1 Rez input for an alert resource

```
resource 'ALRT' (kSaveAlertID, purgeable) {  /*alert resource*/
   {94, 80, 183, 438},        /*rectangle for alert box*/
   kSaveAlertDITL,            /*use the 'DITL' with res ID 200*/
   {                          /*alert stages, starting with #4; at each */
                              /* stage, make OK the default, display the */
                              /* alert box, & play the system alert sound*/
   OK, visible, sound1,       /*4th consecutive error*/
   OK, visible, sound1,       /*3rd consecutive error*/
   OK, visible, sound1,       /*2nd consecutive error*/
   OK, visible, sound1,       /*1st error*/
   },
   alertPositionParentWindow  /*place over document window*/
};
```

An alert resource contains the following information:

- a rectangle, given in global coordinates, that determines the alert box's dimensions and, optionally, its position; these coordinates specify the upper-left and lower-right corners of the alert box
- the resource ID of the item list for the alert box
- the actions to be taken at each of four alert stages
- as an option, a constant (either `alertPositionParentWindow`, `alertPositionMainScreen`, or `alertPositionParentWindowScreen`) that tells the Dialog Manager where to position the alert box (available only to applications running in System 7)

In Listing 6-1, the coordinates (94,80,183,438) specify the dimensions of the alert box, and the `alertPositionParentWindow` constant causes the Dialog Manager to place the alert box just below the title bar of the user's document window. If you don't supply a positioning constant, the Dialog Manager places the alert box at the global coordinates you specify for the alert box's rectangle. The positioning constants for alert boxes are explained in "Positioning Alert and Dialog Boxes" beginning on page 6-62.

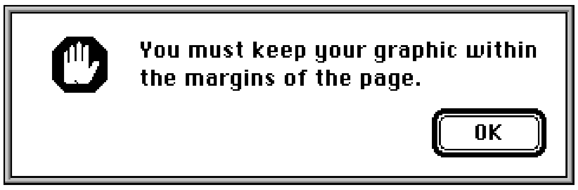
In Listing 6-1, the application-defined constant `kSaveAlertDITL` represents the resource ID for the item list resource. "Providing Items for Alert and Dialog Boxes" beginning on page 6-26 describes how to create an item list resource.

Your application can base its response on the number of consecutive times an alert condition recurs. In Listing 6-1, the alert resource specifies that each consecutive time the user repeats the action that invokes this caution alert, the Dialog Manager should perform the following: outline the OK button and treat it as the default button, display the alert box (that is, make it "visible"), and play a single system alert sound.

Your application can define different responses for each of four stages of an alert. This is most appropriate for stop alerts—those that signify that an action cannot be completed—especially when that action has a high probability of being accidental (for example, when the user chooses the Cut command when no text is selected). Under such a circumstance, your application might simply play the system alert sound the first two times the user makes the mistake, and for subsequent mistakes it might also present an alert box. Every consecutive occurrence of the mistake after the fourth alert stage is treated as a fourth-stage alert.

For example, a user might try to paste a graphic outside the page margins of a simple page-layout program; the first time the user tries this, the application—using the Dialog Manager—may simply play the system alert sound for the user. If the user repeats the mistake, the application may play the system alert sound again. But when the user repeats the error for the third consecutive time, the application may display an alert box like the one shown in Figure 6-11. If the user makes the same mistake immediately after dismissing this alert box, the alert box reappears, and it continues doing so until the user corrects or abandons the improper action.

Figure 6-11 An alert box displayed only during the third and fourth alert stages


You must keep your graphic within the margins of the page.

Listing 6-2 shows the alert resource used to specify the stop alert displayed in Figure 6-11. Notice that the fourth alert stage is listed first, and the first alert stage is listed last. At the third alert stage, the application displays an alert box but does not play the system alert sound. If the user repeats the mistake a fourth consecutive time, the application plays the system alert sound and displays the alert box as well.

Listing 6-2 Specifying different alert responses according to alert stage

```
resource 'ALRT' (kStopAlertID, purgeable) {  /*alert resource*/
   {40, 40, 127, 353},     /*rectangle for alert box*/
   kStopAlertDITL,         /*use the 'DITL' with res ID 300*/
   {                       /*alert stages, starting with #4*/
   OK, visible, sound1,    /*4th err: show alert box, play alert sound*/
   OK, visible, silent,    /*3rd err: show alert box, don't play sound*/
   OK, invisible, sound1,  /*2nd err: play sound, don't show alert box*/
   OK, invisible, sound1,  /*1st err: play sound, don't show alert box*/
   },
   alertPositionParentWindow  /*place over document window*/
};
```

The actions for each alert stage are specified by the following three pieces of information:

- Which button is the default button—the OK button (that is, the first item in the item list resource) or the Cancel button (that is, the second item in the item list resource). The Dialog Manager automatically draws a bold outline around the default button, and when the user presses the Return or Enter key, the Dialog Manager treats—or your event filter function should treat—that keyboard event as a click in the default button. The OK and Cancel buttons are described in detail in “Providing Items for Alert and Dialog Boxes” beginning on page 6-26. At each alert stage, you can change the default button, although it’s difficult to imagine a scenario where changing the default button would be helpful to the user. In the previous example, the OK button is the default.

- Whether the alert box is to be displayed. If you specify the `visible` constant for an alert stage, the alert box is displayed; if you specify the `invisible` constant, it is not. In Listing 6-2, the alert box is not displayed the first two consecutive times the user repeats the mistake, but it is displayed for all subsequent consecutive times.
- Which of four possible sounds (if any) should be emitted at this stage of the alert. In the previous example, the first, second, and fourth alert stages play a single system alert sound, but the third stage plays no sound.

By default, the Dialog Manager uses the system alert sound. The `sound1` constant, used in Listing 6-2, tells the Dialog Manager to play the system alert sound once; you can also specify the `sound2` and `sound3` constants, which cause the Dialog Manager to play the system alert sound two and three times, respectively, each time at the same pitch and with the same duration. The volume of the sound depends on the current speaker volume setting, which the user can adjust in the Sound control panel. If the user has set the speaker volume to 0, the menu bar blinks once in place of each sound that the user would otherwise hear.

If you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure and then call `ErrorSound` and pass it a pointer to your sound procedure. The `ErrorSound` procedure (described on page 6-104) makes your sound procedure the current sound procedure. For example, you might create a sound procedure named `MyAlertSound`, as shown in Listing 6-3.

Listing 6-3 Creating your own sound procedure for alerts

```
PROCEDURE MyAlertSound (soundNo: Integer);
BEGIN
   CASE soundNo OF
      0: PlayMyWhisperAlert;  {sound for silent constant in alert resources}
      1: PlayMyBellAlert;     {sound for sound1 constant in alert resources}
      2: PlayMyDrumAlert;     {sound for sound2 constant in alert resources}
      3: PlayMyTrumpetAlert;  {sound for sound3 constant in alert resources}
   OTHERWISE   ;
   END;                       {of CASE}
END;
```

For each of the four alert stages that can be reported in the `soundNo` parameter, your procedure can emit any sound that you define.

As previously explained, the dimensions of the rectangle you specify in the alert resource determine the dimensions of the alert box. You can also let the rectangle coordinates serve as global coordinates that position the alert box, or you can let the Dialog Manager automatically locate it for you according to three standard positions. Listing 6-2 on page 6-21, for example, uses the `alertPositionParentWindow` constant to position the alert box over the document window where the user is working. For details about these standard positions, see "Positioning Alert and Dialog Boxes" beginning on page 6-62.

Creating Dialog Boxes

To create a dialog box, use the `GetNewDialog` or `NewDialog` function. You should usually use `GetNewDialog`, which takes information about the dialog box from a dialog (`'DLOG'`) resource in a resource file. Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages. The rest of this section describes how to use `GetNewDialog`. Although it's generally not recommended, you can also use the `NewDialog` or `NewColorDialog` function and pass it the necessary descriptive information in individual parameters instead of using a dialog resource. See page 6-118 for a description of `NewDialog` and page 6-115 for a description of `NewColorDialog`.

The `GetNewDialog` function creates a data structure (called a **dialog record**) of type `DialogRecord` from the information in the dialog resource and returns a pointer to it. A dialog record includes a window record. When you use `GetNewDialog`, the Dialog Manager sets the `windowKind` field in the window record to `dialogKind`. As explained in "Displaying Alert and Dialog Boxes" beginning on page 6-61, you can use this pointer with Window Manager or QuickDraw routines to display and manipulate the dialog box.

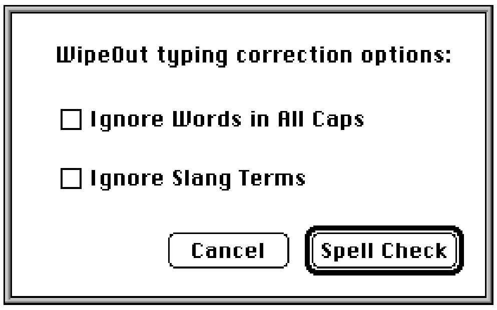
When you use `GetNewDialog`, you pass it the resource ID of the dialog resource, an optional pointer to the memory to use for the dialog record, and the window pointer `Pointer(-1)`, which causes the Window Manager to display the dialog box in front of all other windows.

If you pass `NIL` for the memory pointer, the dialog record is allocated in your application's heap. Passing `NIL` is appropriate for modal dialog boxes and movable modal dialog boxes, but—if you are creating a modeless dialog box—this can cause your heap to become fragmented. In the case of modeless dialog boxes, therefore, you should allocate your own memory as you would for a window; allocating window memory is described in the chapter "Window Manager" in this book.

Here's an example of how to create the dialog box shown in Figure 6-12.

```
VAR
   theDialog:        DialogPtr;
theDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
```

Figure 6-12 A simple modal dialog box



This example uses an application-supplied constant (`kSpellCheckID`) to specify the resource ID number of a dialog resource. Listing 6-4 shows how this dialog resource appears in Rez input format.

Listing 6-4 Rez input for a dialog resource

```
resource 'DLOG' (kSpellCheckID, purgeable) { /*dialog resource*/
   {62, 184, 216, 448},       /*rectangle for dialog box*/
   dBoxProc,                  /*window definition ID for modal dialog box*/
   visible,                   /*display this dialog box immediately*/
   noGoAway,                  /*don't draw a close box*/
   0x0,                       /*initial refCon value of 0*/
   kSpellCheckDITL,           /*use item list with res ID 400*/
   "Spellcheck Options",      /*title if this were a modeless dialog box*/
   alertPositionParentWindow  /*place over document window*/
};
```

The dialog resource contains the following information:

- a rectangle, given in global coordinates, that determines the dialog box's dimensions and, optionally, position; these coordinates specify the upper-left and lower-right corners
- the window definition ID, which specifies the window definition function and variation code for the type of dialog box
- a constant (either `visible` or `invisible`) that specifies whether the dialog box should be drawn on the screen immediately
- a constant (either `noGoAway` or `goAway`); use `goAway` only to specify a close box in the title bar of a modeless dialog box
- a reference value of type `LongInt`, which your application may use for any purpose
- the resource ID of the item list resource for the dialog box

- a text string used for the title of a modeless or movable modal dialog box
- as an option, a constant (either `alertPositionParentWindow`, `alertPositionMainScreen`, or `alertPositionParentWindowScreen`) that tells the Dialog Manager how to position the dialog box (available only to applications running in System 7)

In the example, a rectangle with coordinates (62,184,216,448) specifies the dimensions of the dialog box, and the `alertPositionParentWindow` constant causes the Dialog Manager to place the dialog box just below the title bar of the user's document window. If you don't supply a positioning constant, the Dialog Manager places the dialog box at the global coordinates you specify for the dialog box's rectangle. Positioning constants for dialog boxes are explained in "Positioning Alert and Dialog Boxes" beginning on page 6-62.

In the example, the `dBoxProc` window definition ID is used. Use the following window definition IDs for specifying dialog box types:

Window definition ID	Dialog box type
`dBoxProc`	Modal dialog box
`movableDBoxProc`	Movable modal dialog box
`noGrowDocProc`	Modeless dialog box

In each case, the Dialog Manager uses the Window Manager to draw the appropriate window frame. Figure 6-6 on page 6-10 shows an example of a modal dialog box drawn with the `dBoxProc` window definition ID, Figure 6-7 on page 6-11 shows an example of a movable modal dialog box drawn with the `movableDBoxProc` window definition ID, and Figure 6-8 on page 6-12 illustrates a modeless dialog box drawn with the `noGrowDocProc` window definition ID.

Listing 6-4 specifies the `visible` constant so that the dialog box is drawn immediately. If you use the `invisible` constant, the dialog box is not drawn until your application uses the Window Manager procedure `ShowWindow` to display the dialog box.

Use the `goAway` constant only with modeless dialog boxes. For modal dialog boxes and movable modal dialog boxes, use the `noGoAway` constant, as shown in the example.

Notice that because the example does not make use of the reference constant, 0 (`0x0`) is provided as a filler. However, you may wish to make use of this constant. For example, your application can store a number that represents a dialog box type, or it can store a handle to a record that maintains state information about the dialog box or other window types, as explained in the chapter "Window Manager" in this book. You can use the Window Manager procedure `SetWRefCon` at any time to change this value in the dialog record for a dialog box, and you can use the `GetWRefCon` function to determine its current value.

Listing 6-4 uses an application-defined constant that specifies the resource ID for the item list. The next section, "Providing Items for Alert and Dialog Boxes," describes how to create an item list.

Supply a text string in your dialog resource when you want a modeless dialog box or a movable modal dialog box to have a title. You can specify an empty string for a title bar that contains no text. The example specifies the string “Spellcheck Options” for code readability but, because the example creates a modal dialog box, no title bar is displayed.

You can let the Dialog Manager automatically locate the dialog box according to three standard positions. Listing 6-4 on page 6-24, for example, specifies the `alertPositionParentWindow` constant to position the dialog box over the document window where the user is working. For details on these standard positions, see “Positioning Alert and Dialog Boxes” beginning on page 6-62.

Providing Items for Alert and Dialog Boxes

You use an item list (`'DITL'`) resource to store information about all the items (text, controls, icons, or pictures) in an alert or dialog box. As described in “Creating Alert Sounds and Alert Boxes” beginning on page 6-18 and “Creating Dialog Boxes” beginning on page 6-23, you specify the resource ID of the item list resource in the alert (`'ALRT'`) resource or dialog (`'DLOG'`) resource.

Within an item list resource for an alert box or a dialog box, you specify its static text, buttons, and the resource IDs of icons and QuickDraw pictures. In addition, you can specify checkboxes, radio buttons, editable text, and the resource IDs of other types of controls (such as pop-up menus) in an item list resource for a dialog box.

Figure 6-13 shows an example of an alert box displayed by the SurfWriter application when the user chooses the About command from the Apple menu. To display its own icon in the upper-left corner of the alert box, the application uses the `Alert` function. An alert resource with resource ID 128 is passed in a parameter to the `Alert` function. The alert resource in turn specifies an item list resource with resource ID 128. The item list resource specifies an OK button, some static text, and an icon, whose resource ID is 128. (It’s customary to assign the same resource ID to the item list resource and to either its alert resource or dialog resource, but it’s not necessary to do so.)

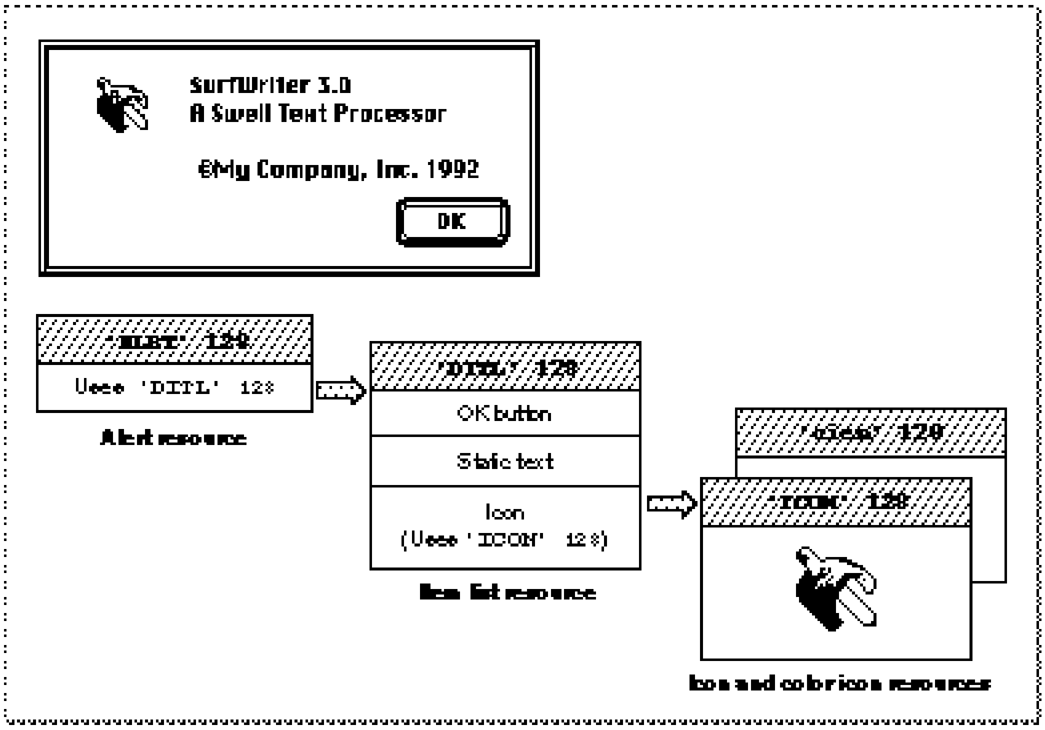
In this example, when the user chooses the About command, the SurfWriter application uses the `Alert` function to display the alert.

```
itemHit := Alert(kAboutBoxID, @MyEventFilter);
```

The `Alert` function in this example displays the alert box defined by the alert resource represented by the `kAboutBoxID` resource ID. As explained in “Responding to Events in Alert Boxes” beginning on page 6-81, the `Alert` function handles most user actions while the alert box is displayed. When the user clicks any button in an alert box, `Alert` removes the alert box and returns to your application the item number of the selected button. The application-defined function `MyEventFilter` that is pointed to in this example allows background applications to receive update events for their windows.

Listing 6-5 shows the resources, in Rez input format, that the `Alert` function uses to display the alert box shown in Figure 6-13.

Figure 6-13 Relationship of various resources to an alert box



Listing 6-5 Rez input for providing an alert box with items

```
# define kAboutBoxID 128      /*resource ID for About SurfWriter alert box*/
# define kAboutBoxDITL 128          /*resource ID for item list*/
# define kSurfWriterIconID 128      /*resource ID for 'ICON' resource*/
# define kSurfWriterColorIconID 128 /*resource ID for 'cicn' resource*/
# define kAboutBoxHelp 128          /*resource ID for 'hdlg' resource*/

resource 'ALRT' (kAboutBoxID, purgeable) {   /*About SurfWriter alert box*/
   {40, 40, 156, 309},            /*rectangle for alert box*/
   kAboutBoxDITL,                 /*use item list resource with ID 128*/
   {  /*identical alert stage responses*/
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent,
   },
   alertPositionMainScreen    /*display on the main screen*/
};
```

```
resource 'DITL' (kAboutBoxDITL, purgeable) { /*items for About SW alert box*/
      /*ITEM NO. 1*/
   {  {86, 201, 106, 259},    /*display rectangle for item*/
      Button {                /*the item is a button*/
            enabled,          /*enable item to return click*/
            "OK"              /*title for button is "OK"*/
      },
      /*ITEM NO. 2*/
      {10, 20, 42, 52},       /*display rectangle for item*/
      Icon {                  /*the item is an icon*/
            disabled,         /*don't return clicks on this item*/
            kSurfWriterIconID /*use 'ICON' & 'cicn' resources */
                              /* with resource IDs of 128*/
      },
      /*ITEM NO. 3*/
      {10, 78, 74, 259},      /*display rectangle for the item*/
      StaticText {            /*the item is static text*/
            disabled,         /*don't return clicks on this item*/
            "SurfWriter 3.0\n"/*text string to display*/
            "A Swell Text Processor \n\n "
            "©My Company, Inc. 1992"
      },
      /*ITEM NO. 4*/
      {0, 0, 0, 0},           /*help items get an empty rectangle*/
      HelpItem {              /*invisible item for reading in help balloons*/
            disabled,         /*don't return clicks on this item*/
            HMScanhdlg        /*scan resource type 'hdlg' for help balloons*/
            {kAboutBoxHelp}   /*get 'hdlg' with resource ID 128*/
      }
   }
};
data 'ICON' (kSurfWriterIconID, purgeable)      {
      /*icon data for black-and-white icon for About SurfWriter goes here*/
};
data 'cicn' (kSurfWriterColorIconID, purgeable)      {
      /*icon data for color icon for About SurfWriter goes here*/
};
```

Items are usually referred to by their positions in the item list resource. For example, the `Alert` function returns 1 when the user clicks the OK button in the alert box created in Listing 6-5. The Dialog Manager returns 1 because the OK button is the first item in the list. (Responding to the item numbers returned by `Alert` and other Dialog Manager functions is explained in "Handling Events in Alert and Dialog Boxes" beginning on page 6-77.) Similarly, when you use a Dialog Manager routine to manipulate an item, you specify it by its **item number,** an integer that corresponds to an item's position in its item list resource.

IMPORTANT
Item list resources should always be marked as purgeable. s

The Dialog Manager also calls the Resource Manager to read in any individual items as necessary.

When you create a dialog box or an alert box, the Dialog Manager creates a *dialog record.* The Dialog Manager then reads in the item list resource and stores a handle to it in the `items` field of the dialog record. Because the Dialog Manager always makes a copy of the item list resource and uses that copy, several independent dialog boxes may share the same item list resource. As explained in “Adding Items to an Existing Dialog Box” beginning on page 6-51, you can use the `AppendDITL` and `ShortenDITL` procedures to modify or customize copies of a shared item list resource for use in individual dialog boxes.

As an alternative to using a dialog resource, you can read in a dialog’s item list resource directly and then pass a handle to it along with other information to `NewDialog`, which creates the dialog box. Remember, however, that it is easier to localize your application if you use a dialog resource and the `GetNewDialog` function.

An item list resource contains the following information for each item:

- a display rectangle
- the type of item (as described in the next section)
- a constant (either `enabled` or `disabled`) that instructs the Dialog Manager whether to report events for that item
- either a text string or a resource ID, as appropriate for the type of item

The **display rectangle** determines the size and location of the item. Specify the display rectangle in coordinates local to the alert or dialog box. For example, in Listing 6-5 the first item is displayed in a rectangle specified by the coordinates (86,201,106,259), which place the item in the lower-right corner of this alert box. More information about display rectangles and their placement is provided in “Display Rectangles” beginning on page 6-32.

In an item list resource, you can specify controls, text, icons, pictures, and other items that you define. In Listing 6-5, the first item’s type is specified by the `Button` constant. Item types and their constants are described in the next section.

For each item in the item list resource, you must also instruct the Dialog Manager whether to report to your application when the item is clicked. In Listing 6-5, the first item is enabled, because the `enabled` constant is specified. “Enabled and Disabled Items” on page 6-36 explains when and how to enable items.

Depending on the type of item in the list, you usually provide a text string or a resource ID for the item. In Listing 6-5, the string `OK` is specified as the button title for the first item. “Resource IDs for Items” beginning on page 6-36 explains what titles and resources are appropriate for the various item types.

The information that you provide in an item list resource is described more fully in the next several sections.

Item Types

The following list shows the types of items you can include in alert and dialog boxes and the constants for specifying them in a Rez input file.

Constant	Description
`Button`	A button control
`CheckBox`	A checkbox control (use in dialog boxes only)
`Control`	A control defined in a `'CNTL'` resource file (use in dialog boxes only)
`EditText`	An editable text item (use in dialog boxes only)
`HelpItem`	An invisible item that makes the Help Manager associate help balloons with the other items defined in the item list resource
`Icon`	An icon whose black-and-white version is stored in an `'ICON'` resource and whose color version is stored in a `'cicn'` resource with the same resource ID as the `'ICON'` resource
`Picture`	A QuickDraw picture stored in a `'PICT'` resource
`RadioButton`	A radio button control (use in dialog boxes only)
`StaticText`	Static text; that is, text that cannot be edited
`UserItem`	An application-defined item (use in dialog boxes only)

The chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox* describes how to create and use items of type `HelpItem` to provide help balloons for your alert and dialog boxes. When you specify a help item, make it the last item in the list, as shown in Listing 6-5 on page 6-27.

The chapter “Finder Interface” in this book describes icon (`'ICON'`) resources and color icon (`'cicn'`) resources. *Inside Macintosh: Imaging* describes `'PICT'` resources.

The chapter “Control Manager” in this book describes how to create a control with a `'CNTL'` resource. Pop-up menus are easily implemented as controls. “Pop-Up Menus as Items” beginning on page 6-42 illustrates how to include pop-up menus in your dialog boxes.

Be aware that alert boxes should contain only buttons (which the user clicks to dismiss the alert box), static text, icons, and pictures. If you need to present other items, you should create a dialog box.

The first item in an alert box’s item list resource should be the OK button; if a Cancel button is necessary, it should be the second item. The Dialog Manager provides these constants for the first two item numbers:

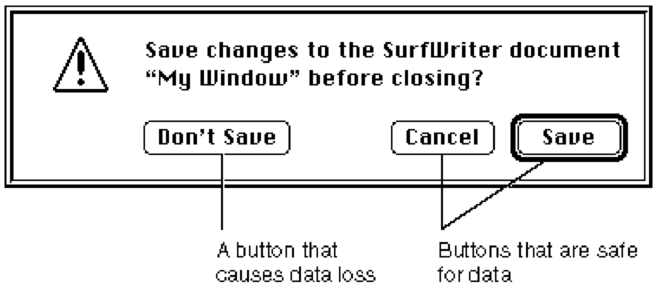
```
CONST ok       = 1;
      cancel   = 2;
```

As described in “Creating Alert Sounds and Alert Boxes” beginning on page 6-18, you define within the alert resource whether the OK or the Cancel button is the default button for each alert stage. The Dialog Manager automatically draws a bold outline around the button that you specify and, if the user presses the Return key or Enter key, the Dialog Manager responds—or your event filter function should respond—as if the default button were clicked. ("Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86 describes event filter functions.)

The Dialog Manager does not draw a bold outline around the default button for dialog boxes. "Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 shows how your application can outline the default button in a dialog box. You should normally give every dialog box a default button—that is, one whose action is invoked when the user presses the Return or Enter key. "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86 shows how to test for these key-down events and respond as if the user had clicked the default button. If you don't provide your own event filter function, the Dialog Manager treats the first item in an item list resource as the default button. That is, although the Dialog Manager doesn't draw a bold outline around the button in a dialog box, the Dialog Manager does return its item number to your application when the user presses the Return or Enter key.

Don't set a default button to perform a dangerous action—for example, one that causes a loss of user data. If none of the possible actions is safe, don't display a default border around any button and provide an event filter function that ignores the Return and Enter keys. This protects users from accidentally damaging their work by pressing Return or Enter out of habit. However, you should try to design a safe action that the user can invoke with a default button, such as a Save button. Figure 6-14 illustrates an alert box that provides a default button for a safe action.

Figure 6-14 A safe default button in an alert box


Save changes to the SurfWriter document "My Window" before closing?

Don't Save
Cancel
Save
A button that causes data loss
Buttons that are safe for data

Provide a Cancel button whenever you can, and in your event filter function, map the Command-period key combination and the Esc (Escape) key to the Cancel button.

Don't display a bold outline around any button if you use the Return key in editable text items, because using the same key for two different purposes confuses users and makes the interface less predictable.

Display Rectangles

As previously mentioned, the display rectangle determines the location of an item within an alert box or a dialog box. Use the alert or dialog box's local coordinates to specify the display rectangle.

For controls, the display rectangle becomes the control's enclosing rectangle. To match a control's enclosing rectangle to its display rectangle, specify an enclosing rectangle in the control resource that is identical to the one you specify for the display rectangle in the item list resource. (The control resource is described in the chapter "Control Manager" in this book.)

Note

Note that, when an item is a control defined in a control (`'CNTL'`) resource, the rectangle added to the update region is the rectangle defined in the control resource, not the display rectangle defined in the item list resource. u

For an editable text item, the display rectangle becomes the TextEdit destination rectangle and its view rectangle. Word wrapping occurs within display rectangles that are large enough to contain multiple lines of text, and the text is clipped if there's more than will fit in the rectangle. The Dialog Manager uses the QuickDraw procedure `FrameRect` to draw a rectangle three pixels outside the display rectangle. For more detailed information about TextEdit, see the chapter "TextEdit" in *Inside Macintosh: Text*.

For a static text item, the Dialog Manager draws the text within the display rectangle just as it draws editable text items, except that the Dialog Manager doesn't draw a frame rectangle outside the display rectangle.

For an icon or a QuickDraw picture larger than its display rectangle, the Dialog Manager scales the icon or picture to fit the display rectangle.

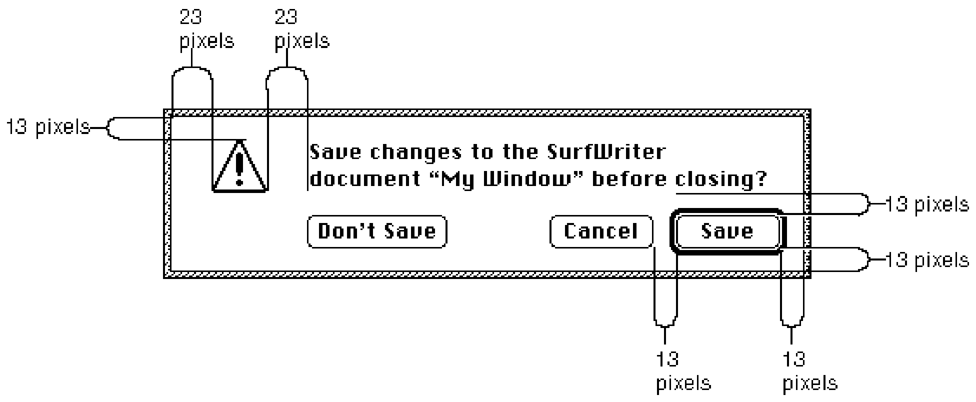
Although the procedure for an application-defined item can draw outside the item's display rectangle, this is not recommended, because if the Dialog Manager receives an update event involving an area outside the display rectangle but inside the area where you draw your application-defined item, the Dialog Manager won't call your draw procedure.

Note

A click anywhere in the display rectangle is considered a click in that item. If display rectangles overlap, a click in the overlapping area is considered a click in whichever item appears first in the item list resource. u

You should display items in functional and consistent locations, both within your application and across all applications that you develop. In alert boxes and in most dialog boxes, place the OK button in the lower-right corner and place the Cancel button to its left. Figure 6-15 shows the recommended location of buttons and text in an alert box.

Figure 6-15 The consistent spacing of buttons and text in an alert box



Generally, you should use distances of 13 and 23 white pixels to separate the items in dialog boxes and alert boxes. (When separating the default button from the window edges and other items, don't count the pixels that make up its bold outline.) However, be aware that the Window Manager adds 3 white pixels inside the window frame when it draws alert boxes and modal dialog boxes. Therefore, specify display rectangle locations as follows when you use tools like Rez and ResEdit:

- Place the display rectangle for the lower-right button 10 pixels from the right edge and 10 pixels from the bottom edge of the alert or modal dialog box; align the display rectangles for other bottommost and rightmost items with this button.
- Place the display rectangle for the upper-left icon (or similar item) 10 pixels from the top edge and 20 pixels from the left of the alert or modal dialog box; align the display rectangles for other topmost and leftmost items with this item. The Dialog Manager automatically places the caution, note, and stop alert icons in this position when you use the `CautionAlert`, `NoteAlert`, and `StopAlert` functions. When you use the `Alert` function, you must specify the icon and the location.
- Place the other elements in the alert or modal dialog box 13 or 23 pixels apart, as shown in Figure 6-15.

For example, the rectangle for the alert box in Figure 6-15 has a specified height of 85 pixels. The display rectangle for the Save button has a bottom coordinate of 75, and the display rectangle for the static text item has a top coordinate of 10. The Window Manager adds 3 white pixels at the top of the alert box and 3 pixels at the bottom, so the alert box contains 13 white pixels below the Save button and 13 white pixels above the static text display rectangle. Listing 6-6 shows how the locations for these display rectangles are specified in a Rez input file.

Listing 6-6 Rez input for consistent spacing of display rectangles

```
resource 'DITL' (200, purgeable) {
   {  {55, 288, 75, 348},  Button      {enabled, "Save"},
      {55, 215, 75, 275},  Button      {enabled, "Cancel"},
      {55, 72,  75, 156},  Button      {enabled, "Don't Save"},
      {10, 75,  42, 348},  StaticText  {disabled,
         "Save changes to the SurfWriter document “^0” before"
         " closing?"}
   }
};
```

When specifying display rectangle locations for items in movable modal and modeless dialog boxes, use the full distance of either 13 or 23 pixels to separate items from the window edges. For example, if the items in Figure 6-15 were placed in a modeless dialog box, the top coordinate of the Save button's display rectangle should be 52 instead of 55, and its bottom coordinate would be 72 instead of 75.

As explained in the previous section, the default button can be any button; its assignment is secondary to the consistent placement of buttons. This rule keeps the OK button and the Cancel button consistently placed. Otherwise, the buttons would keep changing location depending on the default choice.

The Western reader's eye tends to move from the upper-left area of the alert or dialog box to the lower-right area. For Western versions of your software, use the upper-left area for elements (such as the alert icon) that convey the initial impression that you want to make. Place the buttons that a user clicks in the lower-right area.

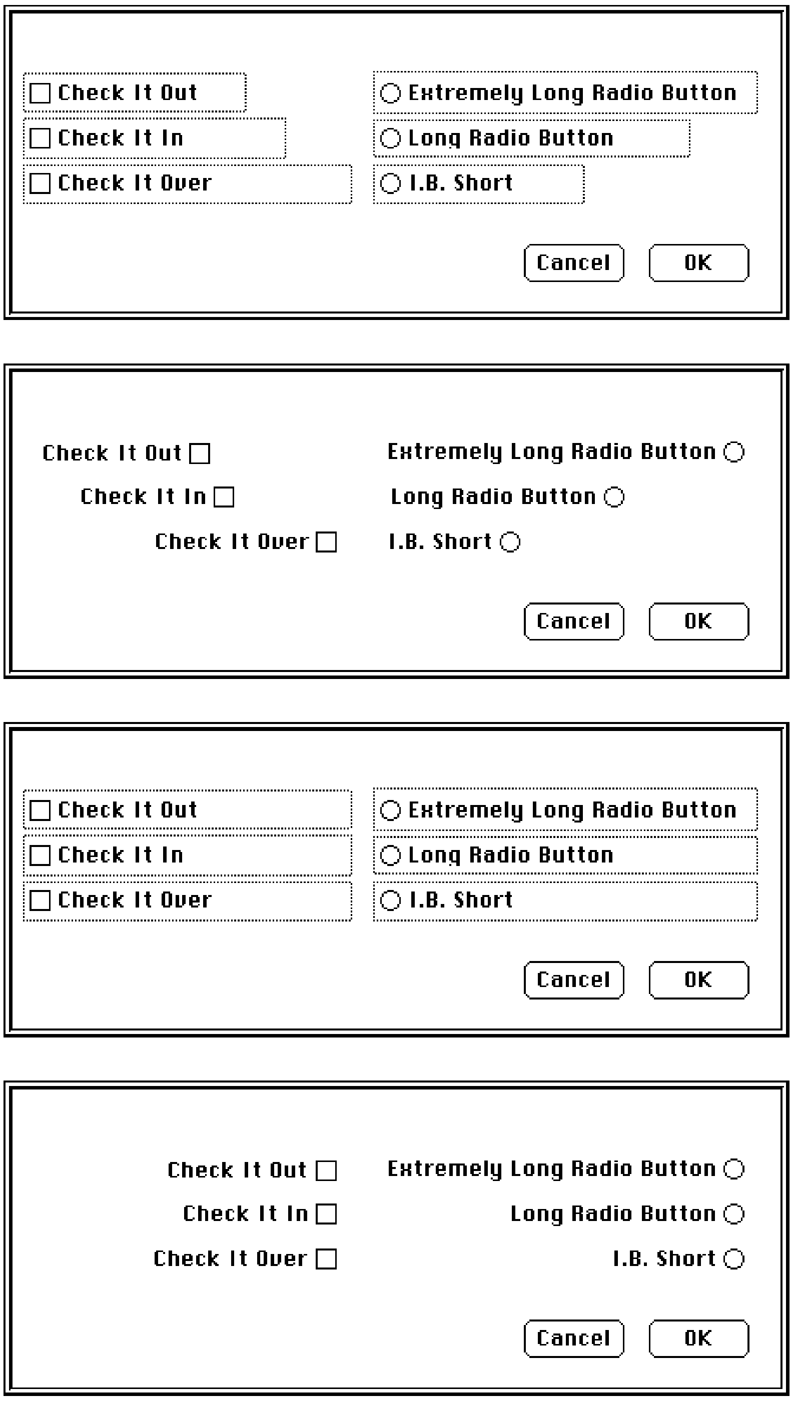
The alignment of the items in an alert box or a dialog box may vary with localization. Although in Roman script systems these items are generally aligned left to right, items in Arabic or Hebrew script systems should generally be aligned right to left, because Arabic and Hebrew are written from right to left. The TextEdit procedure `TESetJust`, described in the chapter “TextEdit” in *Inside Macintosh: Text*, controls the alignment of interface elements.

When line alignment is right to left, as in Hebrew and Arabic, the Control Manager transposes checkboxes—and radio buttons—and their titles. That is, checkboxes and radio buttons appear to the right of the text instead of to the left, as in Roman script systems. Therefore, when you create checkboxes, radio buttons, and static text items that need to align, make sure that their display rectangles are the same size.

The dialog box at the top of Figure 6-16 shows several checkboxes and radio buttons with display rectangles of different sizes. The next dialog box in the figure illustrates what happens to the alignment of these items after the Control Manager transposes the controls with their titles.

The bottom two dialog boxes in Figure 6-16 illustrate how the Control Manager displays properly sized items when transposing the controls with their titles.

Figure 6-16 Incorrectly and correctly sized display rectangles for alternate script systems



A dialog box containing display rectangles of different sizes

The same dialog box after the Control Manager transposes checkboxes and radio buttons with their titles

A dialog box containing display rectangles of the same sizes

The same dialog box after the Control Manager transposes checkboxes and radio buttons with their titles

Enabled and Disabled Items

For each item in an item list resource, include one of these two constants to specify in a Rez input file whether the Dialog Manager should inform your application of events involving that item:

Constant	Description
`enabled`	Informs your application about events involving this item
`disabled`	Doesn't inform your application about events involving this item

Generally, you should enable only controls. In particular, you should enable buttons, radio buttons, and checkboxes so that your application knows when they've been clicked. You typically disable editable text and static text items. You normally disable editable text items because you use the Dialog Manager function `GetDialogItemText` to read the information in the items only after the user clicks the OK button. (Listing 6-12 on page 6-49 illustrates how to use the `GetDialogItemText` function for this purpose.) You should use static text items only for providing information; users don't expect to click them. Likewise, you typically disable icons and pictures that merely provide information; if you use an icon or a picture as a buttonlike control to receive input, however, you must enable it. If you create an application-defined item such as a moving indicator to display information, you typically disable it. If you create an application-defined item such as a buttonlike control to receive input, you must enable it.

Don't confuse disabling an item with using the Control Manager procedure `HiliteControl` to make a control inactive. When you don't want the Control Manager to respond to clicks in a control, you make it *inactive*; when you don't want the Dialog Manager to report clicks in a control, you make it *disabled*.

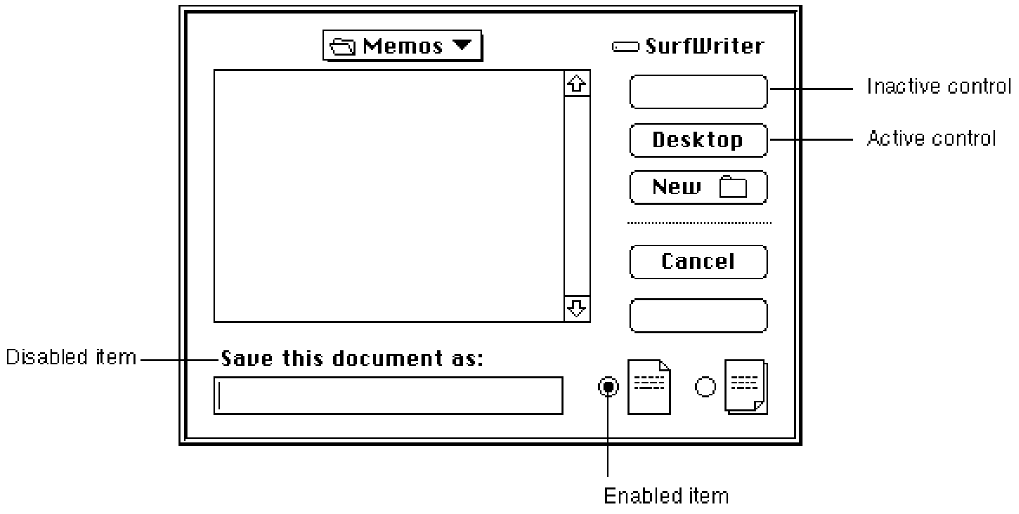
The Control Manager displays an inactive control in a way (dimmed, for example) that shows it's inactive, whereas the Dialog Manager makes no visual distinction between a disabled item and an enabled item. Figure 6-17 shows the difference between an inactive and an active control. The Control Manager procedure `HiliteControl` has been used to dim the inactive Eject button. If a user clicks this button, the Control Manager does not respond. However, when a user clicks the active Desktop button, the Control Manager inverts the button for 8 ticks.

Buttons and other controls are generally enabled, and disabling them does not alter their appearance; the enabled radio button in Figure 6-17 would appear the same if it were disabled. Because the static text reading "Save this document as" in Figure 6-17 is not a control, the application doesn't need to respond clicks in the text. Therefore, the application has disabled it; however, the static text would have the same appearance if the application were to enable it.

Resource IDs for Items

The final element for an item in an item list resource is usually either a text string or a resource ID. The choice depends on the type of item.

Figure 6-17 Inactive controls and disabled items



Provide a resource ID for icons, QuickDraw pictures, and controls other than buttons, checkboxes, and radio buttons. For an icon, provide the ID of an `'ICON'` resource; for a QuickDraw picture, the ID of a `'PICT'` resource; and for a control (including a pop-up menu), the ID of a `'CNTL'` resource. In Listing 6-5 on page 6-27, the resource ID of 128 specifies which `'ICON'` (and `'cicn'`) resources to use for the second item in the item list resource.

For a button, checkbox, radio button, static text item, and editable text item, supply a text string as the final element for the item in its item list resource. The next several sections provide guidelines for the text that you should provide.

For your own application-defined items, supply neither a title nor a resource ID. Listing 6-15 on page 6-57 shows an item list resource that includes an application-defined item.

Titles for Buttons, Checkboxes, and Radio Buttons

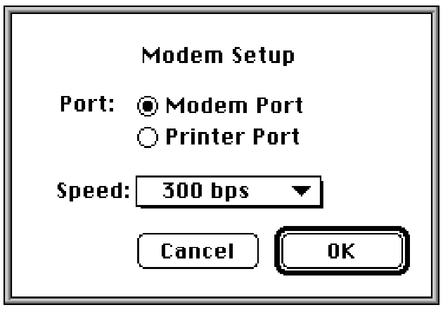
For a button, checkbox, or radio button, provide a text string for the control's title as the final element for the item when you specify it in the item list. In Listing 6-5 on page 6-27, the string `OK` specifies the button title for the first item in the item list resource.

Use book-title capitalization for these items. In general, this means that you capitalize one-word titles and, in multiple-word titles, words of four or more letters. Usually you don't capitalize words such as *in*, *an*, or *and*. The rules for capitalization of titles appear in the *Apple Publications Style Guide*, which is available from APDA.

As explained in the chapter "Control Manager" in this book, the title of a checkbox should reflect two clearly opposite states, because a checkbox should allow the user to turn a particular setting either on or off. The opposites should be expressed in an equal sense in either state. If you can't devise a checkbox title that clearly implies its opposite state, you might be better off using radio buttons. With radio buttons, you can use two titles, thereby clarifying the states. Radio buttons should represent related, but not necessarily opposite, choices. Give each radio button a title consisting of a word or a phrase that identifies what the button does. Remember that, as described in the chapter "Control Manager" in this book, the radio buttons in a group are mutually exclusive: only one button in that group can be on at one time.

Whenever possible, title a button with a verb describing the action that the button performs. A user typically reads the text in an alert box or a dialog box until it becomes familiar and then relies on visual cues, such as button titles or positions, to respond. Buttons such as Save, Quit, or Erase Disk allow users to identify and click the correct button quickly. These titles are often more clear and precise than OK, Yes, and No. If the action can't be condensed into a word or two, OK and Cancel or Yes and No may serve the purpose. If you use these generic titles, be sure to phrase the wording in the dialog box so that the action the button initiates is clear. Figure 6-18 shows a dialog box with appropriate OK and Cancel buttons.

Figure 6-18 A dialog box with OK and Cancel buttons



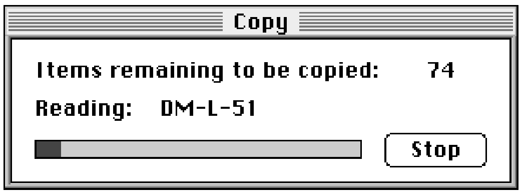
Cancel means "dismiss this operation with no side effects." It does not mean "I've read this dialog box" or "stop what you're doing regardless." When users click the Cancel button in your alert boxes, modal dialog boxes, and movable modal dialog boxes, your application should revoke any actions it took since displaying the alert or dialog box and then remove the box.

Your application should not remove a modeless dialog box when the user clicks a button; rather, you should remove the dialog box when the user clicks its close box or chooses Close from the File menu while the modeless dialog box is active.

When it is impossible to return to the state that existed before an operation began, don't use a Cancel button. You can use Stop or OK, which are useful in different situations. A Stop button may leave the results of a partially complete task intact, whereas a Cancel button always returns the application and its documents to their previous state. Use OK for a button that closes the alert box, modal dialog box, or movable modal dialog box and accepts any changes made while the dialog box was displayed.

Because of the difficulty in revoking the last action invoked from a modeless dialog box, these dialog boxes typically don't have Cancel buttons, although they may have Stop buttons. For example, the movable modal dialog box shown in Figure 6-19 uses a Stop button; clicking the button halts the current file copy operation but leaves intact the copies that were previously made.

Figure 6-19 A movable modal dialog box with a Stop button



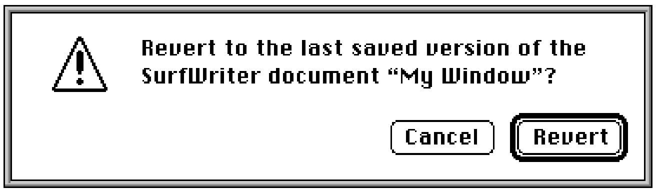
Copy

Items remaining to be copied: 74

Reading: DM-L-51

In an alert box that requires confirmation, use a button title that describes the result of accepting the message in the alert box. For example, if an alert box asks “Revert to the last saved version of the document,” use a Revert button rather than an OK button. Try to use a verb in the button title; as in the Revert button in Figure 6-20, use the same verb that you use in your alert message.

Figure 6-20 An alert box with a Revert button



Revert to the last saved version of the SurfWriter document “My Window”?

If the alert box presents the user with a situation in which no alternative actions are available, give the box a single button that’s titled OK. You should interpret the user’s clicking this button to mean “I’ve read the alert box.”

A modal dialog box usually cuts the user off from the task. That is, when making choices in a modal dialog box, the user can’t see the area of the document that changes. The user sees the changes only after dismissing the dialog box. If the changes aren’t appropriate, then the user has to repeat the entire operation. To provide better feedback to the user, you need to give the user a way to see what the changes will be. Therefore, any selection made in a modal dialog box should immediately update the document contents, or you should provide a sample area in the dialog box that reflects how the user’s selections will change the document. In the case of immediate document updating, the OK button means “accept this change” and the Cancel button means “undo all changes made through this dialog box.”

The Dialog Manager displays button titles (as well as all other control titles) in the system font. To make it easier to localize your application, you should not change the font.

Text Strings for Static Text and Editable Text Items

For an editable text item, if you want the item to display only a blinking cursor, specify an empty string as the item's final element in the item list resource or specify a string if you want to display some default text.

For a static text item, supply a text string as its final element when you specify it in the item list resource. In the third item in Listing 6-5, the text string `SurfWriter 3.0 \nA Swell Text Processor \n\n©My Company, Inc. 1992` specifies the alert box's message.

Whenever you provide static text items in alert and dialog boxes, ensure that the messages make sense to the user. Use simple, nontechnical language and don't provide system-oriented information to which the user can't respond.

Whenever applicable, state the name of the document or application in your alert or dialog box. For example, the alert box in Figure 6-20 on page 6-39 shows both the name of the application (SurfWriter) and the name of the document (My Window). This kind of message helps users who are working with several documents or applications at once to make decisions about each one individually. "Changing Static Text" beginning on page 6-46 describes how to use the `ParamText` procedure to supply the names of document windows to your alert and dialog boxes dynamically.

Use icons and pictures whenever possible. Images can describe some error situations better than words, and familiar icons help users distinguish their alternatives better. However, because experience has shown that it is nearly impossible to create icons that are comprehensible or inoffensive across all international markets, you should be prepared to localize any icons or pictures you use. See the chapter "Icons" in *Macintosh Human Interface Guidelines* for more information about creating appropriate icons.

For your static text items, it's generally better to be polite than abrupt, even if it means lengthening your message. Your message should be helpful, and it may offer constructive suggestions, but it should not appear to give orders. Its focus should be to help the user perform the task, not to give an interesting but academic description of the task itself.

When you localize your application for use with other languages, the text may become longer or shorter. Translated text is often 50 percent longer than U.S. English text. You may need to resize your display rectangles and your alert and dialog boxes to accommodate the translated text.

By default, the Dialog Manager displays static text items in the system font. To make it easier to localize your application, you should not change the font. Do not use a smaller font, such as 9-point Geneva; some script systems such as KanjiTalk require 12-point fonts. You will save yourself future localization effort by leaving all the text in your alert and dialog boxes in the system script.

In alert boxes, try to include information that tells the user how to resolve the problem at hand. Never refer the user to external documentation for further clarification.

Stop alerts typically report errors to the user. A good error message explains what went wrong, why it went wrong, and what the user can do about it. Express this information in the user's vocabulary, not in your programming vocabulary.

Figure 6-21 shows an example of a very poor alert message—the information is expressed in the programmer's vocabulary, and the user is offered no clue about how to remedy the problem.

Figure 6-21 An obscure and useless alert message

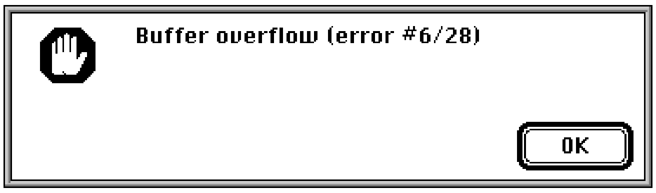

Buffer overflow (error #6/28)

Figure 6-22 shows a somewhat better alert message. Although the vocabulary is less technical, no remedy to the problem is offered.

Figure 6-22 A less obscure alert message

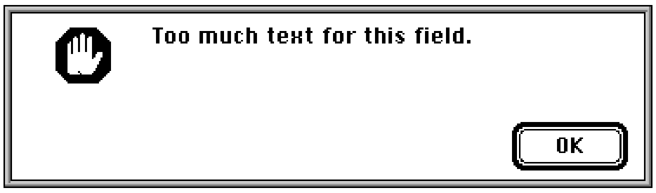
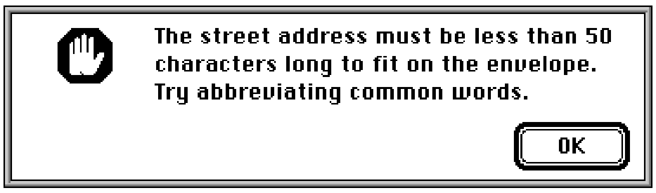

Too much text for this field.

Figure 6-23 illustrates a good alert message. The message is specific, it's expressed in nontechnical terms, it explains why the error occurred, and it suggests a solution to the problem.

Figure 6-23 A clear and helpful alert message

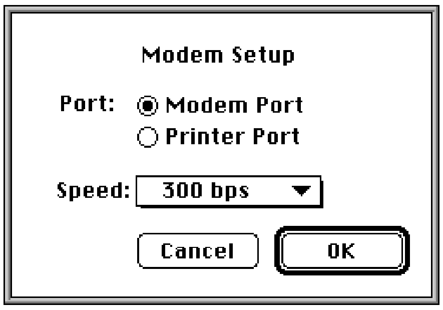

The street address must be less than 50 characters long to fit on the envelope. Try abbreviating common words.

The best way to make an alert message understandable is to think carefully through the error condition itself. Can the application handle this without an error? Is the message specific enough so that the user can fix the situation? What are the recommended solutions?

Pop-Up Menus as Items

You can use pop-up menus to present the user with a list of mutually exclusive choices in a dialog box. Figure 6-24 illustrates a typical use of pop-up menus in a dialog box. As explained in the chapter "Control Manager" in this book, pop-up menus are especially useful as an alternative to radio buttons when the user must make one choice from a list of many or set a specific value, or when you must present a variable list of choices. The pop-up menu in Figure 6-24 allows the application to present a choice of modem speeds that vary according to the modem type in the user's computer.

Figure 6-24 A pop-up menu in a dialog box



In System 7, pop-up menus are implemented as controls. To display a pop-up menu in a dialog box, you

- define specific features of the pop-up menu in a control that uses the standard pop-up control definition function (described in the chapter "Control Manager" in this book)
- define the menu items of a pop-up menu just as you define items in other menus (using `GetMenu` or `NewMenu`, as described in the chapter "Menu Manager" in this book)
- specify the pop-up menu in the dialog box's item list resource

Using the pop-up control definition function, the Dialog Manager automatically draws the pop-up box and its drop shadow, inserts the text into the pop-up box, draws a downward-pointing triangle, and draws the pop-up menu's title. When the user moves the cursor to a pop-up menu and presses the mouse button, the pop-up control definition function highlights the pop-up menu title, displays the pop-up menu, and handles all user interaction until the user releases the mouse button. When the user releases the mouse button, the pop-up control definition function closes the pop-up box, draws the user's choice in the pop-up box (or restores the previous item if the user doesn't make a new choice), and removes the highlighting from the pop-up menu title. The control definition function then sets the value of the control to the item selected by the user. Your application can use the Control Manager function `GetControlValue` to get the number of the currently selected item.

The modal dialog box shown in Figure 6-24 is created by defining a dialog resource that describes the dialog box and by defining an item list resource that describes the dialog items, including a control whose `'CNTL'` resource uses the standard pop-up control definition function. Listing 6-7 shows the dialog resource and item list resource for this modal dialog box.

Listing 6-7 Rez input for a dialog resource and an item list resource for a dialog box that includes a pop-up menu

```
resource 'DLOG' (kModemDialog, purgeable) {
   {62, 184, 216, 416}, dBoxProc, visible, noGoAway, 0x0,
   kModemDialogDITL, "", alertPositionMainScreen
};
resource 'DITL' (kModemDialogDITL, purgeable) {
   {  {123, 152, 144, 222},   Button      {enabled, "OK"},
      {123, 69, 144, 139},    Button      {enabled, "Cancel"},
      {13, 70, 33, 204},      StaticText  {enabled, "Modem Setup"},
      {41, 23, 61, 64},       StaticText  {enabled, "Port:"},
      {41, 67, 59, 186},      RadioButton {enabled, "Modem Port"},
      {59, 67, 77, 186},      RadioButton {enabled, "Printer Port"},
      {90,18,109,198},        Control     {disabled, kPopUpCNTL},
      {123, 152, 144, 222},   UserItem    {disabled}  /*outline OK button*/
      {0,0,0,0},              HelpItem    {disabled, HMScanhdlg{kModemHelp}}
                                                       /*Balloon Help*/
   }
};
```

Listing 6-8 shows the `'CNTL'` and `'MENU'` resources for the Speed pop-up menu shown in Figure 6-24. Notice that the display rectangle specified for the control in the item list resource is the same as the enclosing rectangle specified in the control resource. See the chapter “Control Manager” in this book for a complete description of how to specify values for a pop-up menu’s control resource.

Listing 6-8 Rez input for a control resource and a menu resource for a pop-up menu

```
resource 'CNTL' (kPopUpCNTL, preload, purgeable) {
   {90, 18, 109, 198},  /*enclosing rectangle of control*/
   popupTitleLeftJust,  /*title position*/
   visible,             /*make control visible*/
   50,                  /*pixel width of title*/
   kPopUpMenu,          /*'MENU' resource ID*/
   popupMenuCDEFProc,   /*pop-up control definition ID*/
   0,                   /*reference value*/
   "Speed:"             /*control title*/
};
```

```
resource 'MENU' (kPopUpMenu, preload, purgeable) {
   mPopUp, textMenuProc,
   0b1111111111111111111111111111111,
   enabled, "Speed",
   {
      "300 bps",     noicon, nokey, nomark, plain;
      "1200 bps",    noicon, nokey, nomark, plain;
      "2400 bps",    noicon, nokey, nomark, plain;
      "9600 bps",    noicon, nokey, nomark, plain;
      "19200 bps",   noicon, nokey, nomark, plain
   }
};
```

Keyboard Navigation Among Items

Your dialog boxes may have several items, such as editable text items and scrolling lists, that can accept input from the keyboard. You need to give users a visual cue indicating which item is currently accepting input from the keyboard. Each item type has its own distinct indicator. The Dialog Manager automatically displays a blinking cursor in an editable text item to indicate that it is accepting keyboard input. You can also use the `SelectDialogItemText` procedure (explained on page 6-131) to indicate a selected text range within an editable text item.

When a scrolling list is accepting keyboard input, you should indicate it by a rectangular border of two black pixels, separated from the list by one pixel of white space. In Figure 6-25, the AppleTalk Zones scrolling list is the item currently accepting keyboard input in the Chooser dialog box. See the chapter “List Manager” in *Inside Macintosh: More Macintosh Toolbox* for details about creating lists in dialog boxes.

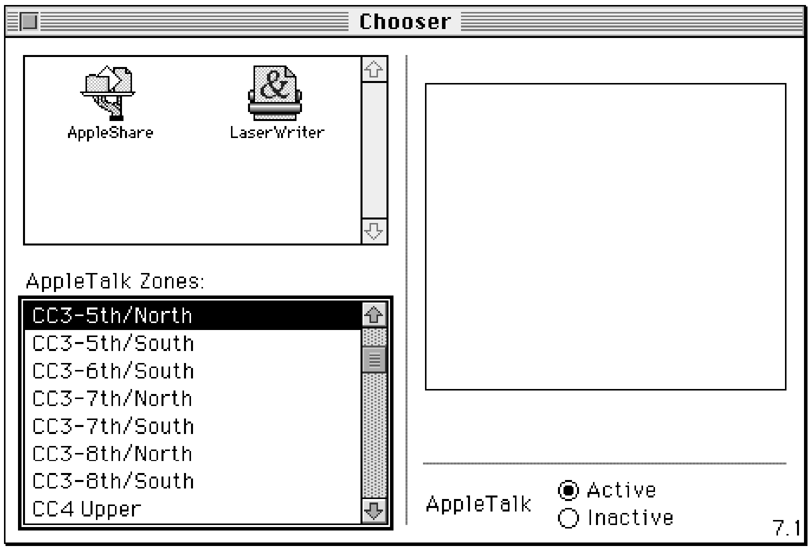
Because all typing goes to the active window, there should be only one active area and only one indicator at any time. If only one element in a dialog box can accept keyboard input and that element is a scrolling list, it’s not necessary to place a border around it.

The Dialog Manager automatically handles mouse-down events and keyboard events for the Tab key. Thus, the user can select any item that accepts keyboard input by clicking the desired item or by pressing the Tab key to cycle through the available items. When the user presses the Tab key, the Dialog Manager accepts the changes made to the current item and selects the next item—as listed in the item list—that accepts keyboard input. When the user clicks another item, the Dialog Manager accepts the changes made to the current item and selects the newly clicked item.

Manipulating Items

In many cases, you won’t have to make any changes to alerts or dialog boxes after you define them in your resource file. However, if you should want to modify an item, you can use several Dialog Manager routines to do so.

Figure 6-25 A selected scrolling list



For example, you can use the `ParamText` procedure to supply text strings (such as document titles) to alert and dialog boxes dynamically. For most other types of item manipulation, you must first call the `GetDialogItem` procedure to get the information about the item. You then use other routines to manipulate that item. For example, you can use the `SetDialogItem` procedure to change the item, or—to get a text string that the user has entered in an editable text item after clicking the OK button—you can use the `GetDialogItemText` procedure.

The Dialog Manager routines for manipulating items are summarized in the following list.

Routine	Description
`AppendDITL`	Adds items to a dialog box.
`CountDITL`	Counts the number of items in a dialog box.
`FindDialogItem`	Finds an item that contains a specified point within a dialog box.
`GetAlertStage`	Returns the stage of the last occurrence of an alert.
`GetDialogItem`	Returns the item type, the display rectangle, and the control handle or application-defined procedure of a given item in a dialog box.
`GetDialogItemText`	Returns the text of a given editable text or static text item.
`HideDialogItem`	Hides the given item.
`ParamText`	Substitutes up to four different text strings in static text items.
`ResetAlertStage`	Resets the stage of the last occurrence of an alert.

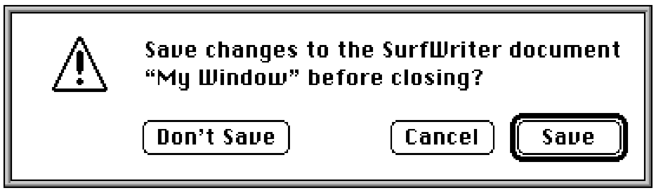
Routine	Description
`SelectDialogItemText`	Selects the text of an editable text item.
`SetDialogItem`	Sets the item type and the display rectangle of an item, or (for application-defined items) the draw procedure of an item.
`ShortenDITL`	Removes items from a dialog box.
`ShowDialogItem`	Redisplays the item previously hidden by `HideDialogItem`.

The next several sections describe the most frequently used of these routines. The next section, "Changing Static Text," explains the use of the `ParamText` procedure to manipulate the text in static text items. "Getting Text From Editable Text Items" beginning on page 6-48 describes how to use the `GetDialogItemText` procedure to determine what the user types in an editable text item. Using the `AppendDITL` procedure is explained in "Adding Items to an Existing Dialog Box" beginning on page 6-51. "Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 describes how to use `SetDialogItem` to install application-defined items. For additional information about all of the previously listed routines, see "Manipulating Items in Alert and Dialog Boxes" beginning on page 6-120 and "Handling Text in Alert and Dialog Boxes" beginning on page 6-129.

Changing Static Text

As previously explained, it is often useful to state the name of a document in an alert box or a dialog box. For example, Figure 6-26 shows an alert box that an application might display when the user closes a window that contains unsaved changes.

Figure 6-26 An alert box that displays a document name


Save changes to the SurfWriter document "My Window" before closing?

You can use the `ParamText` procedure to supply the names of document windows to your alert and dialog boxes dynamically, as illustrated in the application-defined routine `MyCloseDocument` shown in Listing 6-9.

Listing 6-9 Using the `ParamText` procedure to substitute text strings

```
PROCEDURE MyCloseDocument (myData: MyDocRecHnd);
VAR
   title:      Str255;
   item:       Integer;
   docWindow:  WindowPtr;
   event:      EventRecord;   {dummy parameter for calling DialogSelect}
   myErr:      OSErr;
BEGIN
   docWindow := FrontWindow;     {point to active window}
   IF (myData^^.windowDirty) THEN   {document has been changed}
   BEGIN
      GetWTitle(docWindow, title);  {get title of window}
      MyStringCheck(title);
      ParamText(title, '', '', ''); {pass the title in 1st parameter}
      DoActivate(docWindow, FALSE, event);   {deactivate the active window}
      item := CautionAlert(kSaveAlertID, @MyEventFilter); {display alert box}
      IF item = kCancel THEN
         Exit(MyCloseDocument);
      IF item = kSave THEN
         DoSaveCmd;        {save the document}
      myErr := DoCloseFile(myData);    {close the file}
   END;                 {let click in Don't Save fall through}
   CloseWindow(docWindow);
   DisposPtr(Ptr(docWindow));
END;
```

In this example, the Window Manager function `FrontWindow` returns a pointer to the active window. Another Window Manager function, `GetWTitle`, returns the title of that window. The `MyCloseDocument` routine passes this string to the `ParamText` procedure, which takes four text strings as parameters. In this example, only one string is needed (the window title), which is passed in the first parameter; empty strings are passed for the remaining three parameters.

You can use `ParamText` to supply up to four text strings for a single alert or dialog box. In the item list resource for the alert or dialog box, specify where each of these strings should go by inserting the special characters `^0` through `^3` in any of the items where you can specify text. The `ParamText` procedure dynamically replaces `^0` with the string you pass in its first parameter, `^1` with the string in the second parameter, and so forth, when you display the alert or dialog box.

IMPORTANT
To avoid recursion problems in versions of system sofware earlier than 7.1, you have to ensure that you do not include the characters ^0 through ^3 in any strings you pass to ParamText. This is why MyCloseDocument uses another application-defined routine, MyStringCheck, to filter these characters out of the window titles passed to ParamText. s

Listing 6-10 shows a portion of an item list resource. When the application calls CautionAlert, the Dialog Manager uses the first parameter passed previously to the ParamText procedure to replace the characters ^0 in the static text with the title of the document window.

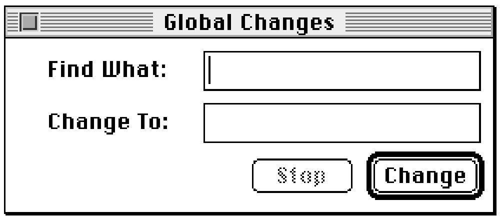
Listing 6-10 Specifying where ParamText should substitute text in an alert box message

```
resource 'DITL' (kSaveAlertID, purgeable) {
   {        /*Save button information goes here*/
            /*Cancel button information goes here*/
            /*Don't Save button information goes here*/
      {10, 75, 42, 348},
      StaticText {      /*ask the user to save changes to the document--*/
            disabled,   /* filename inserted with ParamText*/
            "Save changes to the SurfWriter document “^0” before closing?"
      },
            /*help item information goes here*/
   }
};
```

Getting Text From Editable Text Items

The application displaying the modeless dialog box shown in Figure 6-27 uses the GetDialogItem and GetDialogItemText procedures after the user clicks the Change button.

Figure 6-27 Two editable text items in a modeless dialog box



This dialog box prompts the user for two text strings: one to search for and another to take the place of the first string. Listing 6-11 shows the item list resource for this dialog box. The fifth item in the list is the editable text item where the user enters the text string being sought; the sixth item is the item where the user enters the replacement text string.

Listing 6-11 Specifying editable text items in an item list

```
resource 'DITL' (kGlobalChangesDITL, purgeable) {
   {  /*ITEM NO. 1*/
      {70, 213, 90, 271},  Button      {enabled,"Change"},
      /*ITEM NO. 2*/
      {70, 142, 90, 200},  Button      {enabled,"Stop"},
      /*ITEM NO. 3*/
      {10, 23, 27, 98},    StaticText  {disabled, "Find What:"},
      /*ITEM NO. 4*/
      {40, 23, 57, 98},    StaticText  {disabled,"Change To:"},
      /*ITEM NO. 5*/
      {10, 117, 27, 271},  EditText    {disabled, ""},
      /*ITEM NO. 6*/
      {40, 117, 57, 271},  EditText    {disabled, ""}
      /*ITEM NO. 7: for drawing outline around Change button*/
      {63, 205, 97, 278},  UserItem    {disabled, },
      /*ITEM NO. 8: help item goes here*/
   }
};
```

Listing 6-12 shows how the application handles a click in the Change button. (Subsequent sections of this chapter explain how to handle events in a modeless dialog box.)

Listing 6-12 Getting the text entered by the user in an editable text item

```
PROCEDURE MyHandleModelessDialogs(theEvent: EventRecord);
VAR
   myDialog:            DialogPtr;
   itemHit, itemType:   Integer;
   searchStringHandle:  Handle;
   replaceStringHandle: Handle;
   searchString:        Str255;
   replaceString:       Str255;
   itemRect:            Rect;
```

```
BEGIN
   {use DialogSelect, then determine whether the event occurred }
   { in the Global Changes dialog box; if so, respond to mouse }
   { clicks as follows}
   CASE itemHit OF
      kChange:      {user clicked the Change button}
         BEGIN
            GetDialogItem(myDialog, kFind, itemType,
                          searchStringHandle, itemRect);
            GetDialogItemText(searchStringHandle, searchString);
            GetDialogItem(myDialog, kReplace, itemType,
                          replaceStringHandle, itemRect);
            GetDialogItemText(replaceStringHandle, replaceString);
            {get a handle to the Stop button}
            GetDialogItem(myDialog, kStop, itemType,
                          itemHandle, itemRect);
            {make the Stop button active during the operation}
            HiliteControl(ControlHandle(itemHandle), 0);
            {get a handle to the Change button}
            GetDialogItem(myDialog, kChange, itemType,
                          itemHandle, itemRect);
            {make the Change button inactive during the operation}
            HiliteControl(ControlHandle(itemHandle), 255);
            DoReplace(searchString, replaceString);
            {when the operation is complete, dim Stop and make }
            { Change active here}
         END;
      kStop:     {user clicked the Stop button}
         BEGIN
               {cancel operation, then make Stop button }
               { inactive and Change button active again}
         END;
   END;
END;
```

In Listing 6-12, when the user clicks the Change button, the `GetDialogItem` procedure returns a handle to the item containing the search string. Because this is a handle to an editable text item, the application can pass the handle to the `GetDialogItemText` procedure, which then returns the item's text string in its second parameter. These two procedures are then used to get the string in the item containing the replacement string. These two strings are then passed to an application-defined routine that replaces all instances of the first string with the characters of the second string. Note that when the user clicks Change, the Control Manager procedure `HiliteControl` is used to make the Stop button active and to make the Change button inactive—that is, dimmed. This indicates that the user can use the Stop button but not the Change button while the change operation is taking place.

Adding Items to an Existing Dialog Box

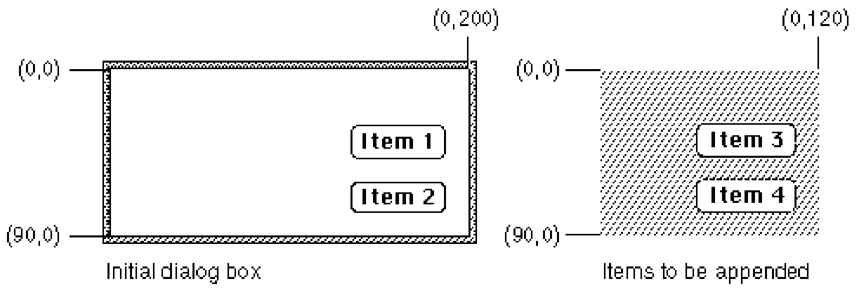
You can dynamically add items to and remove items from a dialog box by using the `AppendDITL` and `ShortenDITL` procedures. When you create a dialog box, the Dialog Manager creates a *dialog record.* The Dialog Manager then reads in the item list resource and stores a handle to it in the `items` field of the dialog record. Because every dialog box you create has its own dialog record, you can define dialog boxes whose items are defined by the same item list resource. The `AppendDITL` and `ShortenDITL` procedures are especially useful if several dialog boxes share the same item list resource and you want to add or remove items as appropriate for individual dialog boxes.

When you call the `AppendDITL` procedure, you specify a dialog box, and you specify a new item list resource to append to the dialog box's existing item list resource. You also specify where the Dialog Manager should display the new items. You can use one of these constants to designate where `AppendDITL` should display the appended items:

```
CONST overlayDITL       = 0;     {overlay existing items}
      appendDITLRight   = 1;     {append at right}
      appendDITLBottom  = 2;     {append at bottom}
TYPE DITLMethod = Integer;
```

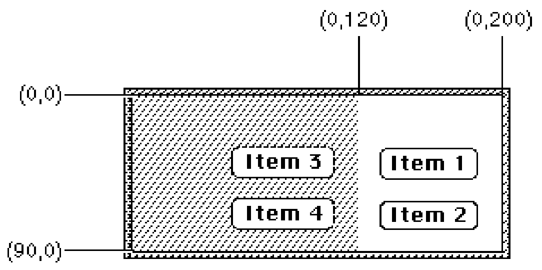
Figure 6-28 illustrates an existing dialog box and a pair of items to be appended.

Figure 6-28 An existing dialog box and items to append



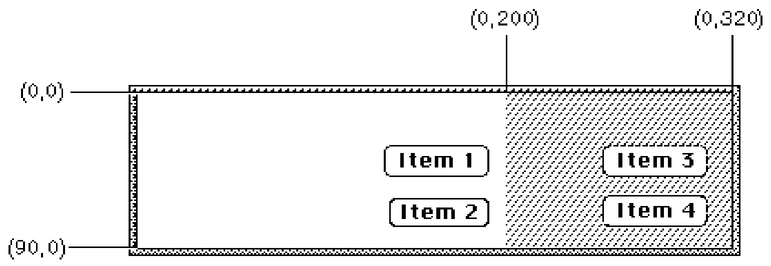
If you specify the `overlayDITL` constant, `AppendDITL` superimposes the appended items over the dialog box. That is, `AppendDITL` interprets the coordinates of the display rectangles for the appended items (as specified in their item list resource) as local coordinates within the dialog box. Figure 6-29 shows the result of overlaying the items upon the dialog box illustrated in Figure 6-28.

Figure 6-29 The dialog box after items are overlaid



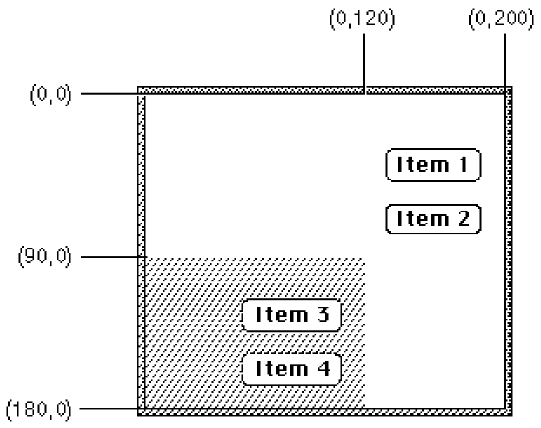
If you specify the `appendDITLRight` constant, `AppendDITL` appends the items to the right side of the dialog box, as illustrated in Figure 6-30, by positioning the display rectangles of the appended items relative to the upper-right coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.

Figure 6-30 The dialog box after items are appended to the right



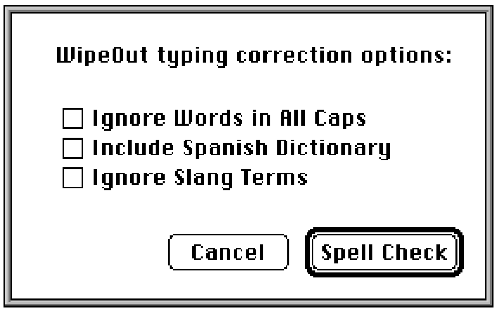
If you specify the `appendDITLBottom` constant, `AppendDITL` appends the items to the bottom of the dialog box, as illustrated in Figure 6-31, by positioning the display rectangles of the appended items relative to the lower-left coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.

Figure 6-31 The dialog box after items are appended to the bottom



As an alternative to passing the `overlayDITL`, `appendDITLRight`, or `appendDITLBottom` constant, you can pass a negative number to `AppendDITL`, which appends the items relative to an existing item in the dialog box. The absolute value of this number is interpreted as the item in the dialog box relative to which the new items are to be positioned. For example, if you pass –2 to `AppendDITL`, the display rectangles of the appended items are offset from the upper-left corner of item number 2 in the dialog box. Figure 6-12 on page 6-24 shows a simple dialog box with two checkboxes. Figure 6-32 shows the same dialog box after an additional item is appended relative to the first checkbox, so that the new item appears between the two existing checkboxes.

Figure 6-32 A dialog box with an item appended relative to an existing item


WipeOut typing correction options:

☐ Ignore Words in All Caps
☐ Include Spanish Dictionary
☐ Ignore Slang Terms

The application-defined routine called `DoSpellBoxWithSpanish`, which is shown in Listing 6-13 on the next page, illustrates the use of the `AppendDITL` procedure to add the new item.

Listing 6-13 Appending an item to an existing dialog box

```
FUNCTION DoSpellBoxWithSpanish: OSErr;
VAR
   theDialog:        DialogPtr;
   myNewItem:        Handle;
   docWindow:        WindowPtr;
   event:            EventRecord;
BEGIN
   theDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
   IF theDialog <> NIL THEN
   BEGIN
      myNewItem := GetResource('DITL', kSpanishDITL);
      IF myNewItem <> NIL THEN
      BEGIN
         AppendDITL(theDialog, myNewItem, kAppendItem);  {kAppendItem = -3}
         ReleaseResource(myNewItem);
         docWindow := FrontWindow;     {get the front window}
         {if there's a front window, deactivate it}
         IF docWindow <> NIL THEN
            DoActivate(docWindow, FALSE, event);
         ShowWindow(theDialog);  {show dialog box with appended item}
         MyAdjustMenus;             {adjust menus as needed}
         REPEAT
            ModalDialog(@MyEventFilter, itemHit);
               {handle clicks in checkboxes here}
         UNTIL ((itemHit = kSpellCheck) OR (itemHit = kCancel));
               {handle clicks in buttons here}
         DisposeDialog(theDialog);
         DoSpellBoxWithSpanish := kSuccess;
      END
      ELSE
         DoSpellBoxWithSpanish := kFailed;
   END
   ELSE DoSpellBoxWithSpanish := kFailed;
END;
```

The `DoSpellBoxWithSpanish` routine uses `GetNewDialog` to create a dialog box. As you'll see in Listing 6-14, the dialog resource passed to `GetNewDialog` has a resource ID of 402, and this dialog resource in turn specifies an item list resource with resource ID 402. The `DoSpellBoxWithSpanish` routine then uses the Resource Manager function `GetResource` to obtain a handle to a second item list resource; this item list resource contains the "Include Spanish Dictionary" checkbox. By setting a value of –3 in the last parameter of `AppendDITL`, the `DoSpellBoxWithSpanish` routine appends the items in the second item list resource relative to item number 3 (the “Ignore Words in All Caps” checkbox) in the dialog box. Listing 6-14 shows the dialog resource for the dialog box, its regular item list resource, and the item list resource that `AppendDITL` adds to it.

Listing 6-14 Rez input for a dialog box and the item appended to it

```
# define kSpellCheckID 402   /*resource ID for Spell Check dialog box*/
# define kSpellCheckDITL 402 /*resource ID for item list resource*/
# define kSpanishDITL 257    /*resource ID for item list resource to append*/
# define kAppendHelp 257     /*resource ID for 'hdlg' for appended item*/

resource 'DLOG' (kSpellCheckID, purgeable) {/*Spell Check dialog box*/
   {62, 184, 216, 448},
   dBoxProc,              /*make it modal*/
   invisible,             /*make it initially invisible*/
   noGoAway, 0x0, kSpellCheckDITL, "Spellcheck Options",
   alertPositionParentWindow     /*place over the document window*/
};
resource 'DITL' (kSpellCheckDITL, purgeable) {
   /*items for Spell Check dialog box*/
   /*ITEM NO. 1, the "Spell Check" button, goes here*/
   /*ITEM NO. 2, the "Cancel" button, goes here*/
   /*ITEM NO. 3*/
   {48, 23, 67, 202}, CheckBox {enabled, "Ignore Words in All Caps"},
   /*ITEM NO. 4*/
   {83, 23, 101, 196}, CheckBox {enabled, "Ignore Slang Terms"},
      /*static text, help item, etc. go here*/
};
   /*add this item list resource to Spell Check dialog box only when */
   /* Spanish language dictionary is installed*/
resource 'DITL' (kSpanishDITL, purgeable) {
   {  {18, 0, 36, 209},CheckBox {enabled,"Include Spanish Dictionary"},
      {0,0,0,0}, HelpItem {disabled, HMScanAppendhdlg{kAppendHelp}} /*help*/
   }
};
```

The dialog resource specifies that the dialog box is invisible so that the application can add the new item to the dialog box before displaying it. In Listing 6-13, the `DoSpellBoxWithSpanish` routine uses the Window Manager procedure `ShowWindow` to display the dialog box after its new item has been appended. (“Displaying Alert and Dialog Boxes” beginning on page 6-61 describes more fully how to display dialog boxes.)

The appended item list resource includes a help item that causes the Help Manager to use the help resource associated with that item list resource in addition to the help resource originally associated with the dialog box. See the chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox* for information about using the `HMScanAppendhdlg` identifier in a help item.

Listing 6-13 uses the Resource Manager procedure `ReleaseResource`. The `AppendDITL` procedure modifies the contents of the dialog box (for instance, by enlarging it). To use an unmodified version of the dialog box at a later time, your application needs to use `ReleaseResource` to release the memory occupied by the appended item list. Otherwise, if your application calls `AppendDITL` to add items to that dialog box again, the dialog box will remain modified by your previous call—for example, it will still be longer at the bottom if you previously used the `appendDITLBottom` constant.

When you can call the `ShortenDITL` procedure to remove items from the end of a dialog item list, you specify a pointer to the dialog box and the number of items to remove from the end of the item list. Note that `ShortenDITL` does not automatically resize the dialog box; you can use the Window Manager procedure `SizeWindow` if you need to resize the dialog box. You can use the `CountDITL` function to determine the number of items in the item list resource for a dialog box.

Using an Application-Defined Item to Draw the Bold Outline for a Default Button

You can define your own type of item for dialog boxes. You might wish, for example, to display a clock with the current time in a dialog box. You can also use application-defined items to draw a bold outline around the default button in a dialog box.

You should not use application-defined items in an alert box because they add unnecessary programming complications. If you need an application-defined item, use a dialog box instead.

To define your own item, include an item of type `UserItem` in your item list resource; it should have a display rectangle, but no text and no resource ID associated with it. The dialog resource that uses this item list resource must specify the `invisible` constant. This makes the dialog box invisible while you install a draw procedure for your application-defined item. After installing the procedure that draws the application-defined item, you display the dialog box by using the Window Manager procedure `ShowWindow`.

For example, Figure 6-32 on page 6-53 illustrates a dialog box that outlines the default button (Spell Check). To outline the button, the application must add an item of type `UserItem` to the item list resource for that dialog box.

So that an application-defined drawing procedure can draw a border around the Spell Check button, the item list resource in Listing 6-15 specifies a larger display rectangle for the application-defined item than for the Spell Check button.

Listing 6-15 Rez input for an application-defined item in an item list

```
resource 'DITL' (kSpellCheckDITL, purgeable) {
   /*ITEM NO. 1: OK button--the default*/
   {  {123, 170, 144, 254}, Button {enabled,"Spell Check"},
   /*ITEMs 2-5 go here: Cancel button, two checkboxes, and static text*/
   /*ITEM NO. 6: application-defined item*/
      {115, 164, 152, 260},   /*6th item*/
      UserItem {              /*draw procedure for item draws an outline*/
            disabled,         /*1st item lies inside this--1st is enabled*/
      }
   }
};
```

The application-defined item is disabled because the Spell Check button, which lies within the application-defined item, is enabled. Because the Spell Check button is listed before the application-defined item in this item list resource, the Dialog Manager reports when the user clicks the Spell Check button. However, note that when application-defined items are enabled, the Dialog Manager reports their item numbers when the user clicks them.

Note

Although the draw procedure for an application-defined item can draw outside the item's display rectangle, this is not recommended because if the Dialog Manager receives an update event involving an area outside the display rectangle but inside the area where you draw your application-defined item, the Dialog Manager won't call your draw procedure. u

Listing 6-14 on page 6-55 shows the dialog resource for the dialog box. Notice that the `invisible` constant in the dialog resource specifies that the dialog box should initially be invisible.

You must provide a procedure that draws your application-defined item. Your draw procedure must have two parameters: a dialog pointer and an item number from the dialog box's item list resource. For example, this is how you should declare the draw procedure if you were to name it `MyDrawDefaultButtonOutline`:

```
PROCEDURE MyDrawDefaultButtonOutline (theDialog: DialogPtr;
                                      theItem: Integer);
```

The parameter `theDialog` is a pointer to the dialog box containing the application-defined item. (If your procedure draws in more than one dialog box, this parameter tells your procedure which dialog box to draw in.) The parameter `theItem` is a number corresponding to the position of an item in the item list resource for the dialog box. (In case the procedure draws more than one item, this parameter tells the procedure which one to draw.)

To install this draw procedure, use the `GetDialogItem` and `SetDialogItem` procedures. Use `GetDialogItem` to return a handle to the application-defined item specified in the item list resource. Then use `SetDialogItem` to replace this handle with a pointer to your draw procedure. When calling your draw procedure, the Dialog Manager sets the current port to the dialog box's graphics port. The Dialog Manager then calls your procedure to draw the application-defined item as necessary—for instance, when you display the dialog box and whenever the Dialog Manager receives an update event for the dialog box.

Listing 6-16 illustrates how to install the procedure that draws a bold outline. In this listing, `GetDialogItem` gets a handle to the application-defined item (which is the sixth item in the item list resource from Listing 6-15). The procedure pointer `@MyDrawDefaultButtonOutline`, which is coerced to a handle, is then passed to `SetDialogItem`, which sets the draw procedure into the dialog record.

Listing 6-16 Installing the draw procedure for an application-defined item

```
FUNCTION DisplayMyDialog (VAR theDialog: DialogPtr): OSErr;
VAR
   itemType:      Integer;
   itemHandle:    Handle;
   itemRect:      Rect;
   docWindow:     WindowPtr;
   event:      EventRecord;
BEGIN
   {begin by creating an invisible dialog box}
   theDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
   IF theDialog <> NIL THEN
   BEGIN
      {get a handle to the application-defined item (i.e., userItem)}
      GetDialogItem(theDialog, kUserItem, itemType, itemHandle, itemRect);
      {install the drawing procedure for the application-defined item}
      SetDialogItem(theDialog, kUserItem, itemType,
                    Handle(@MyDrawDefaultButtonOutline), itemRect);
      docWindow := FrontWindow;     {get the front window}
         {if there's a front window, deactivate it}
      IF docWindow <> NIL THEN
         DoActivate(docWindow, FALSE, event);
      ShowWindow(theDialog);     {display the dialog box}
      MyAdjustMenus;             {adjust menus as needed}
      DisplayMyDialog := kSuccess;
   END
   ELSE DisplayMyDialog := kFailed;
   {call ModalDialog and handle events in dialog box here}
END;
```

Use the Window Manager procedure `ShowWindow` to display the previously invisible dialog box. When `ShowWindow` is called in this example, a bold outline is drawn inside the application-defined item and around the Spell Check button.

Listing 6-17 shows a procedure that draws a bold outline around a button of any size and shape. This procedure can be used to draw the outline around the Spell Check button from the previous example.

Listing 6-17 Creating a draw procedure that draws a bold outline around the default button

```
PROCEDURE MyDrawDefaultButtonOutline(theDialog: DialogPtr; theItem: Integer);
CONST
   kButtonFrameInset =  -4;
   kButtonFrameSize =   3;
   kCntrActivate =      0;
VAR
   itemType:      Integer;    {returned item type}
   itemRect:      Rect;       {returned display rectangle}
   itemHandle:    Handle;     {returned item handle}
   curPen:        PenState;
   buttonOval:    Integer;
   fgSaveColor:   RGBColor;
   bgColor:       RGBColor;
   newfgColor:    RGBColor;
   newGray:       Boolean;
   oldPort:       WindowPtr;
   isColor:       Boolean;
   targetDevice:  GDHandle;
BEGIN
   {get the default button & draw a bold border around it}
   GetDialogItem(theDialog, kDefaultButton, itemType, itemHandle, itemRect);
   GetPort(oldPort);
   SetPort(ControlHandle(itemHandle)^^.contrlOwner);
   GetPenState(curPen);
   PenNormal;
   InsetRect(itemRect, kButtonFrameInset, kButtonFrameInset);
   FrameRoundRect(itemRect, 16, 16);
   buttonOval := (itemRect.bottom - itemRect.top) DIV 2 + 2;
   IF ((CGrafPtr(ControlHandle(itemHandle)^^.contrlOwner)^.portVersion) =
       kIsColorPort) THEN
      isColor := TRUE
   ELSE
      isColor := FALSE;
   IF (ControlHandle(itemHandle)^^.contrlHilite <> kCntrlActivate) THEN
   BEGIN    {control is dimmed, so draw gray default button outline}
      newGray := FALSE;
      IF isColor THEN
      BEGIN
         GetBackColor(bgColor);
         GetForeColor(fgSaveColor);
         newfgColor := fgSaveColor;
            {get the device on which this dialog box is displayed}
         targetDevice :=
               MyGetDeviceFromRect(ControlHandle(itemHandle)^^.contrlRect);
            {use the gray defined by the display device}
         newGray := GetGray(targetDevice, bgColor, newfgColor);
      END;
      IF newGray THEN
         RGBForeColor(newfgColor)
      ELSE
         PenPat(gray);
      PenSize(kButtonFrameSize, kButtonFrameSize);
      FrameRoundRect(itemRect, buttonOval, buttonOval);
      IF isColor THEN
         RGBForeColor(fgSaveColor);
   END
   ELSE  {control is active, so draw default button outline in black}
   BEGIN
      PenPat(black);
      PenSize(kButtonFrameSize, kButtonFrameSize);
      FrameRoundRect(itemRect, buttonOval, buttonOval);
   END;
   SetPenState(curPen);
   SetPort(oldPort);
END;
```

Listing 6-17 uses `GetDialogItem` to get the Spell Check button and then uses several QuickDraw routines to draw a black outline around that button's display rectangle when the button is active. If the button is inactive (that is, dimmed), `MyDrawDefaultButtonOutline` draws a gray outline.

Before drawing a gray outline, `MyDrawDefaultButtonOutline` determines whether the dialog box uses a color graphics port. As explained in "Including Color in Your Alert and Dialog Boxes" beginning on page 6-75, you can supply a dialog box with a color graphics port by creating a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource. If the dialog box uses a color graphics port, `MyDrawDefaultButtonOutline` uses the Color QuickDraw function `GetGray` to return a blended gray based on the foreground and background colors. Then

MyDrawDefaultButtonOutline uses this gray for outlining the dimmed default button. Otherwise, MyDrawDefaultButtonOutline uses the QuickDraw procedure PenPat to draw a gray outline on black-and-white monitors.

Displaying Alert and Dialog Boxes

You typically define alerts and dialog boxes in resources, as described in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 and in "Creating Dialog Boxes" beginning on page 6-23. To create an alert or a dialog box, you use a Dialog Manager function—such as Alert or GetNewDialog—that incorporates information from your item list resource and from your alert resource or dialog resource into a data structure, called a *dialog record,* in memory. The Dialog Manager creates a dialog record, which is a data structure of type DialogRecord, whenever your application creates an alert or a dialog box.

The Dialog Manager automatically displays alert boxes at the appropriate alert stages; it also automatically displays those dialog boxes that you specify as visible in their dialog resources. But you must use a Window Manager routine such as ShowWindow to display dialog boxes that you specify as invisible in their dialog resources.

When you use a function that creates an alert (namely, Alert, StopAlert, NoteAlert, orCautionAlert), the Dialog Manager automatically displays the alert box at the alert stages that you specify with the visible constant in your alert resource. You do not use any routines other than the Alert, StopAlert, NoteAlert, and CautionAlert functions to display an alert box.

When you specify the visible constant in a dialog resource, the Dialog Manager immediately displays the dialog box when you use the GetNewDialog function. If you instead specify the invisible constant so that the dialog box is initially invisible when you call GetNewDialog, use the Window Manager procedure ShowWindow to display it. This is useful if you need to manipulate a dialog item dynamically using GetDialogItem and SetDialogItem before you display the dialog box. For example, if you want to install an application-defined draw procedure for a dialog box, you specify the invisible constant in a dialog resource, pass the resource ID of that dialog resource in a parameter to GetNewDialog, use GetDialogItem and SetDialogItem to install the application-defined draw procedure, then call ShowWindow to display the dialog box, as previously shown in Listing 6-16 on page 6-58.

You should always specify Pointer(-1) as a parameter to GetNewDialog to display a dialog box as the active (that is, frontmost) window.

You should perform the following tasks in conjunction with displaying an alert box or a dialog box:

- Specify an appropriate screen position at which to display the alert box or dialog box.
- Deactivate the frontmost window (if one exists) before displaying an alert box or a modal dialog box.
- Determine whether you've already created a modeless dialog box and, if so, select it instead of creating a new instance of it.
- Adjust your menus appropriately for a modal dialog box with editable text items and for any movable modal and modeless dialog box you wish to display.

The `DialogSelect` function uses the QuickDraw procedure `SetPort` to make the alert or dialog box the current graphics port. The `ModalDialog` procedure and the functions that create alert boxes use `DialogSelect` to respond to update and activate events. You can also use `DialogSelect` to respond to update and activate events in your modeless and movable modal dialog boxes. In response to update events, you can instead use the `UpdateDialog` function, which also makes the dialog box the current graphics port. In these cases, it's generally not necessary for your application to call `SetPort` when displaying, updating, or activating alert boxes and dialog boxes. See *Inside Macintosh: Imaging* for more information about `SetPort`.

These and other related issues are explained in detail in the next several sections of this chapter.

Positioning Alert and Dialog Boxes

As previously described in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 and "Creating Dialog Boxes" beginning on page 6-23, you specify a rectangle in every alert resource and dialog resource. The dimensions of this rectangle determine the dimensions of the alert box or dialog box. You can also let the rectangle coordinates serve as the global coordinates that determine the position of the alert box or dialog box, or you can let the Dialog Manager automatically locate it for you according to three standard positions. To specify these standard positions in System 7, your application can use the following constants in the Rez input files for alert resources and dialog resources:

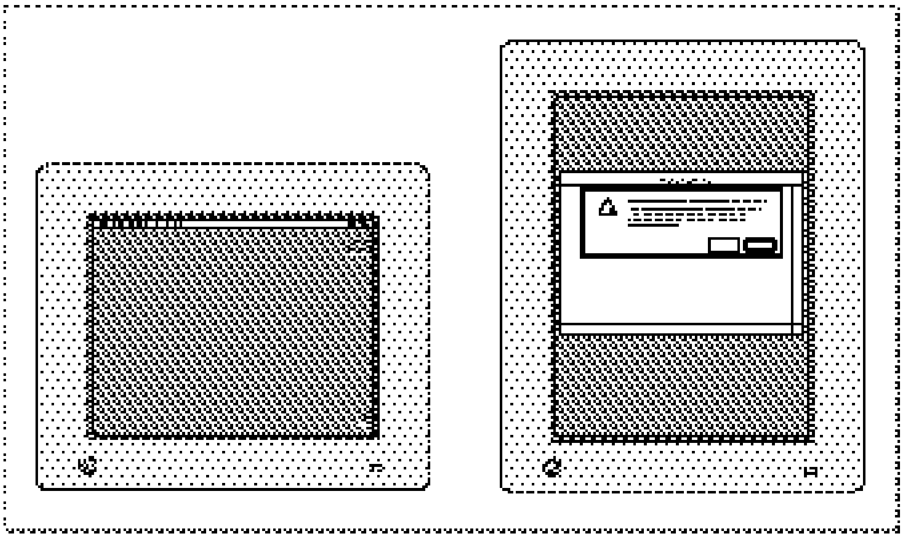
Constant	Description
`alertPositionParentWindow`	Position the alert or dialog box over the frontmost window
`alertPositionMainScreen`	Position the alert or dialog box on the main screen
`alertPositionParentWindowScreen`	Position the alert or dialog box on the screen containing the frontmost window

If your application positions alert or dialog boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager. If you do use these constants, use them to specify the positions of both alert boxes and dialog boxes.

The next three figures illustrate various alert boxes that might appear when the user is working on two monitors: a 12-inch monitor (the main screen) that displays the menu bar and a full-page monitor that displays a document window. These figures show where the Dialog Manager places an alert box according to the position specified in the alert resource.

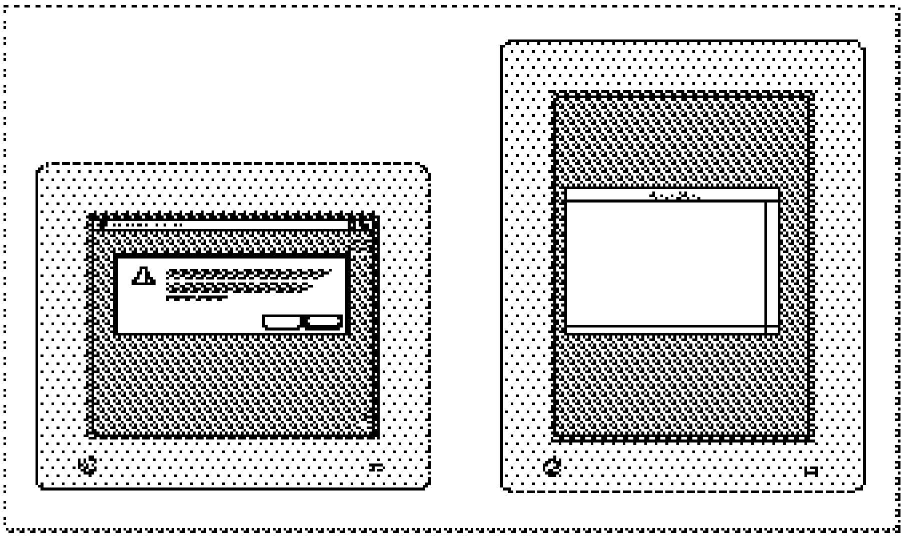
Figure 6-33 shows an alert box displayed in response to an error made by the user while working on a document; the alert resource specifies the `alertPositionParentWindow` constant, which tells the Dialog Manager to position the alert box over the frontmost window so that the window's title bar appears. This position is appropriate for an alert box or a dialog box that relates directly to the frontmost window. You should always try to position alert boxes and dialog boxes where the user is working.

Figure 6-33 An alert box in front of a document window



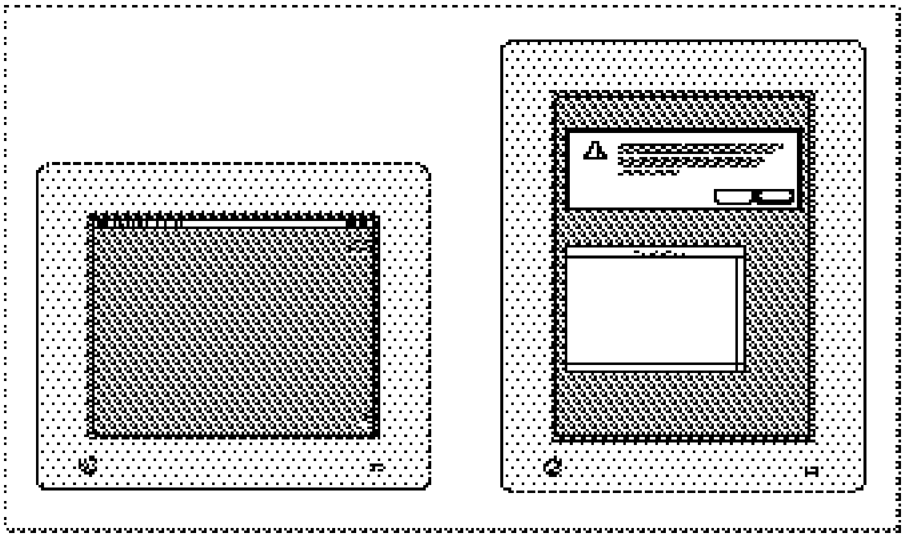
Not all alert boxes or dialog boxes relate to the frontmost window. Some may relate only to actions the user performs on the main screen. For example, Figure 6-34 illustrates an alert box displayed when the user chooses the About command from the Apple menu. For an alert box or dialog box such as this, you should specify the `alertPositionMainScreen` constant in the alert or dialog resource. Figure 6-34 shows how the Dialog Manager centers such an alert box near the top of the main screen.

Figure 6-34 An alert box on the main screen



Sometimes you may need to display an alert box or a dialog box that applies neither to the frontmost window nor to an action performed on the main screen. To catch the user's attention, you should position such an alert or dialog box on the screen where the user is working. For example, if you need to alert the user that available disk space is low, you should specify the `alertPositionParentWindowScreen` constant. Figure 6-35 shows how the Dialog Manager displays such an alert box or dialog box when a document window appears on a screen other than the main screen.

Figure 6-35 An alert box in the alert position of the document window screen



If you don't specify a positioning constant, the Dialog Manager uses the rectangle coordinates in your alert resource or dialog resource as global coordinates specifying where to position your alert or dialog box. If you wish to specify the position yourself in this manner, you should generally try to center alert and dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is most appropriate. If you don't use the positioning constants, you should also place the tops of alert and dialog boxes (including the title bars of modeless and movable modal dialog boxes) below the menu bar. You can use the `GetMBarHeight` function, described in the chapter "Menu Manager" in this book, to determine the height of the menu bar.

Deactivating Windows Behind Alert and Modal Dialog Boxes

For alert and modal dialog boxes, the `ModalDialog` procedure traps all events before they are passed to your event loop, which normally handles activate events for your windows. Thus, if a window is active, you must explicitly deactivate it before displaying an alert box or a modal dialog box.

Your modeless dialog boxes and movable modal dialog boxes never use the `ModalDialog` procedure. Therefore, you do not have to deactivate the frontmost window explicitly before displaying a modeless or a movable modal dialog box.

Instead, the Event Manager continues sending your application activate events for your windows as needed, which you typically handle in your normal event loop. (The chapters "Event Manager" and "Window Manager" in this book explain how to activate and deactivate windows.)

Plate 2 at the front of this book shows an alert box that an application displays when the user chooses the About command in the Apple menu. Listing 6-18 shows an application-defined routine, `ShowMyAboutBox`, that displays this alert box.

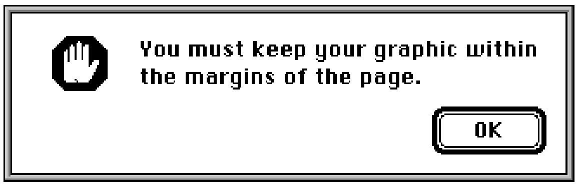
Listing 6-18 Deactivating the front window before displaying an alert box

```
PROCEDURE ShowMyAboutBox;
VAR
   itemHit:    Integer;
   docWindow:  WindowPtr;
   event:      EventRecord;
BEGIN
   docWindow := FrontWindow;     {get the front window}
   {if there's a front window, deactivate it}
   IF docWindow <> NIL THEN
      DoActivate(docWindow, FALSE, event);
   {then show the alert box}
   itemHit := Alert(kAboutBoxID, @MyEventFilter);
END;
```

The `ShowMyAboutBox` routine uses the Window Manager function `FrontWindow`. If `FrontWindow` returns a valid pointer, `ShowMyAboutBox` calls its `DoActivate` procedure to deactivate that window before calling the `Alert` function to display the alert box. When the user clicks the OK button, the alert box is dismissed. The Event Manager then sends the application update events so that it can update the contents of any windows as appropriate, and the Event Manager sends the application an activate event so that it can activate the previously frontmost window again. The application handles these events in its normal event loop.

If your application does not display an alert box during certain alert stages, use the `GetAlertStage` function to test for those stages before deactivating the active window. The `GetAlertStage` function returns the last occurrence of an alert as a number from 0 to 3. Figure 6-36 shows an alert box that appears only after the user repeats an error three consecutive times.

Figure 6-36 An alert box displayed only after the third alert stage


You must keep your graphic within the margins of the page.

Listing 6-19 shows how you might use `GetAlertStage` to determine if such an alert needs to be displayed before deactivating the document window.

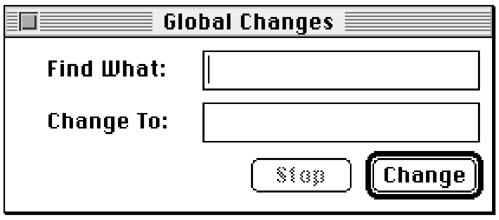
Listing 6-19 Using `GetAlertStage` to determine when to deactivate the front window

```
PROCEDURE MyAlert;
VAR
   itemHit:       Integer;
   alertStage:    Integer;
   docWindow:     WindowPtr;
   event:         EventRecord;
BEGIN
   docWindow := FrontWindow;
   alertStage := GetAlertStage;
   IF (alertStage >= 2) AND (docWindow <> NIL) THEN    {at 3rd alert stage, }
      DoActivate(docWindow, FALSE, event);      { deactivate front window & }
   itemHit := StopAlert(kStopAlertID, @MyEventFilter);   { display alert box}
END;
```

Displaying Modeless Dialog Boxes

For a modeless dialog box, check to make sure it isn't already open before you create and display it. For example, the modeless dialog box shown in Figure 6-37 should appear when the user chooses the Global Changes command. After invoking this command, the user may select another window, thereby deactivating the modeless dialog box.

Figure 6-37 A modeless dialog box for changing text in a document



So as not to create multiple versions of this dialog box whenever the user chooses the Global Changes command, the application-defined routine `DoGlobalChangesDialog`, shown in Listing 6-20, checks whether the dialog box already exists.

Listing 6-20 Ensuring that the modeless dialog box isn't already open before creating it

```
FUNCTION DoGlobalChangesDialog: OSErr;
BEGIN
   DoGlobalChangesDialog := kSuccess;  {assume success}
   IF gChangeDialogPtr = NIL THEN      {it doesn't exist, so create it}
   BEGIN
      gChangeDialogPtr := GetNewDialog(kGlobalChangesDlog, NIL, Pointer(-1));
      IF gChangeDialogPtr = NIL THEN   {handle failure}
      BEGIN
         DoGlobalChangesDialog := kFailed;
         EXIT(DoShowModelessFindDialogBox);
      END;
      {set window refCon to store value that identifies the dbox}
      SetWRefCon(gChangeDialogPtr, LongInt(kGlobalChangesDlog));
   END
   ELSE        {it does exist, so display and select it}
   BEGIN
      ShowWindow(gChangeDialogPtr); {it's hidden; so show it}
      SelectWindow(gChangeDialogPtr);{bring it to the front}
   END;
   MyAdjustMenus;                {adjust the menus}
END;
```

In this example, a pointer to the modeless dialog box is stored in a global variable. If the global variable does not contain a pointer, `DoGlobalChangesDialog` uses `GetNewDialog` to create and draw the dialog box. Later, if the user decides to close the modeless dialog box, the application merely hides it so that when the user needs it again, `DoGlobalChangesDialog` can display the dialog box in the same location and with the same text selected as when the user last used it. Hiding this dialog box is illustrated later in Listing 6-30 on page 6-94.

If the dialog box has already been created, `DoGlobalChangesDialog` uses the Window Manager procedures `ShowWindow` to make the dialog box visible and `SelectWindow` to make it active.

Finally, `DoGlobalChangesDialog` uses the application-defined routine `MyAdjustMenus` to adjust the menus as appropriate for the modeless dialog box.

Listing 6-34 on page 6-98 illustrates an application-defined routine, `DoActivateGlobalChangesDialog`, that handles activate events for this modeless dialog box. The `DoActivateGlobalChangesDialog` routine in turn uses `DialogSelect`, which sets the graphics port to the modeless dialog box whenever the user makes it active.

Adjusting Menus for Modal Dialog Boxes

The Dialog Manager and the Menu Manager interact to provide various degrees of access to the menus in your menu bar. For alert boxes and modal dialog boxes without editable text items, you can simply allow system software to provide the appropriate access to your menu bar.

When your application displays either an alert box or a modal dialog box (that is, a window of type `dBoxProc`), these actions occur:

1. System software disables all menu items in the Help menu, except the Show Balloons (or Hide Balloons) command, which system software enables.
2. System software disables all menu items in the Application menu.
3. If the Keyboard menu appears in the menu bar, system software enables that menu but disables the About Keyboards command.

When your application displays an alert box or calls the `ModalDialog` procedure for a modal dialog box (described in "Responding to Events in Modal Dialog Boxes" beginning on page 6-82), the Dialog Manager determines whether any of the following cases is true:

- Your application does not have an Apple menu.
- Your application has an Apple menu, but the menu is disabled when the dialog box is displayed.
- Your application has an Apple menu, but the first item in that menu is disabled when the dialog box is displayed.

If none of these cases is true, system software behaves as follows:

1. The Menu Manager disables all of your application's menus.
2. If the modal dialog box contains a visible and active editable text field—and if the menu bar contains a menu having commands with the standard keyboard equivalents Command-X, Command-C, and Command-V—then the Menu Manager enables those three commands and the menu that contains them. The user can then use either the menu commands or their keyboard equivalents to cut, copy, and paste text. (The menu item having keyboard equivalent Command-X must be one of the first five menu items.)

When your application displays alert boxes and modal dialog boxes with no editable text items, it can safely allow system software to handle menu bar access as described in steps 1 and 2.

However, because system software cannot handle the Undo or Clear command (or any other context-appropriate command) for you, your application should handle its own menu bar access for modal dialog boxes with editable text items by performing the following tasks:

- disable the Apple menu or the first item in the Apple menu (typically, your application's About command) in order to take control of its menu bar access when displaying a modal dialog box

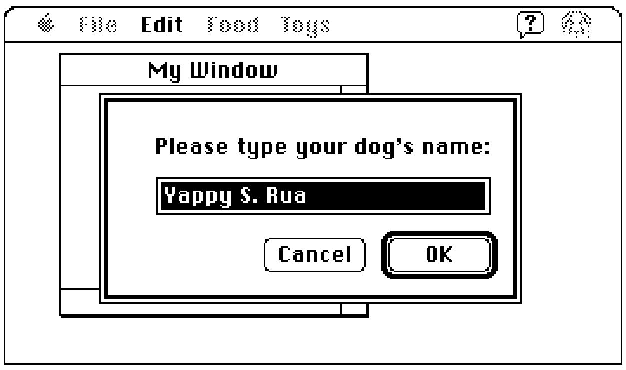
- disable all of its menus except the Edit menu, as well as any inappropriate commands in the Edit menu
- use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items
- provide your own code for supporting the Undo command
- enable your application's items in the Help menu as appropriate (system software disables all items except the Hide Balloons / Show Balloons command)

You don't need to do anything else for the system-handled menus—namely, Application, Keyboard, and Help. System software handles these menus for you automatically.

The `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` procedures are described beginning on page 6-132. Your application can test whether a dialog box is the front window when handling mouse-down events in the Edit menu and then call these routines as appropriate.

Figure 6-38 illustrates how an application disables all of its own menus except its Edit menu when displaying a modal dialog box containing editable text items. Access to the Edit menu benefits the user who instead of typing prefers copying from and pasting into editable text items.

Figure 6-38 Menu access when displaying a modal dialog box



Listing 6-21 on the next page shows an application-defined routine, `MyAdjustMenus`, that the SurfWriter application calls to adjust its menus after it displays a window or dialog box, but before it calls `ModalDialog` to handle events in a modal dialog box. When `MyAdjustMenus` determines that the frontmost window is a modal dialog box containing an editable text item, it calls another application-defined routine, `MyAdjustMenusForDialogs`, which adjusts the menus appropriately. Listing 6-22 on the next page shows the `MyAdjustMenusForDialogs` routine.

Listing 6-21 Adjusting menus for various windows

```
PROCEDURE MyAdjustMenus;
VAR
   window:     WindowPtr;
   windowType: Integer;
   menu:       MenuHandle;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:   {document window is in front}
      MyAdjustMenusForDocWindows;
   kMyDialogWindow:  {a dialog box is in front}
      MyAdjustMenusForDialogs;
   kDAWindow:  {adjust menus accordingly for a DA window}
      MyAdjustMenusForDA;
   kNil: {there isn't a front window}
      MyAdjustMenusNoWindows;
   END; {of CASE}
   DrawMenuBar;   {redraw menu bar}
END;
```

The `MyAdjustMenusForDialogs` routine in Listing 6-22 first determines what type of dialog box is in front: modal, movable modal, or modeless. For modal dialog boxes, `MyAdjustMenusForDialogs` disables the Apple menu so that the application can take control of its menus away from the Dialog Manager. The `MyAdjustMenusForDialogs` routine then uses the Menu Manager routines `GetMenuHandle` and `DisableItem` to disable all other application menus except the Edit menu. (To provide help balloons that explain why these menus are unavailable to the user, `MyAdjustMenusForDialogs` uses the Help Manager procedure `HMSetMenuResID` to reassign help resources to these menus; see the chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox* for more information.)

Listing 6-22 Disabling menus for a modal dialog box with editable text items

```
PROCEDURE MyAdjustMenusForDialogs;
   VAR
      window:     WindowPtr;
      windowType: Integer;
      myErr:      OSErr;
      menu:       MenuHandle;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyModalDialogs:
      BEGIN
         menu := GetMenuHandle(mApple);   {get handle to Apple menu}
         IF menu = NIL THEN
            EXIT(MyAdjustMenusForDialogs);
         DisableItem(menu, 0);   {disable Apple menu to get control of menus}
         myErr := HMSetMenuResID(mFile, kFileHelpID); {set up help balloons}
         menu := GetMenuHandle(mFile);       {get handle to File menu}
         IF menu = NIL THEN
            EXIT(MyAdjustMenusForDialogs);
         DisableItem(menu, 0);      {disable File menu}
         myErr := HMSetMenuResID(mFile, kFileHelpID); {set up help balloons}
         IF myErr <> NoErr THEN
            EXIT(MyAdjustMenusForDialogs);
         menu := GetMenuHandle(mTools);   {get handle to Tools menu}
         IF menu = NIL THEN
            EXIT(MyAdjustMenusForDialogs);
         DisableItem(menu, 0);         {disable Tools menu}
         myErr := HMSetMenuResID(mTools, kToolsHelpID);  {help balloons}
         IF myErr <> NoErr THEN
            EXIT(MyAdjustMenusForDialogs);
         MyAdjustEditMenuForModalDialogs;
      END;        {of kMyModalDialogs CASE}
      kMyGlobalChangesModelessDialog:
         ;  {adjust menus here as needed}
      kMyMovableModalDialog:
         ;  {adjust menus here as follows: }
            { disable all menus except Apple, then }
            { call MyAdjustEditMenuForModalDialogs for editable text items}
   END; {of CASE}
END;
```

To adjust the items in the Edit menu, `MyAdjustMenusForDialogs` calls another application-defined routine, `MyAdjustEditMenuForModalDialogs`, which is shown in Listing 6-23 on the next page. The `MyAdjustEditMenuForModalDialogs` routine uses application-defined code to implement the Undo command; uses the Menu Manager procedure `EnableItem` to enable the Cut, Copy, Paste, and Clear commands when appropriate; and disables the commands that support Edition Manager capabilities. Remember that your application should use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items.

Listing 6-23 Adjusting the Edit menu for a modal dialog box

```
PROCEDURE MyAdjustEditMenuForModalDialogs;
VAR
   window:           WindowPtr;
   menu:             MenuHandle;
   selection, undo:  Boolean;
   offset:           LongInt;
   undoText:         Str255;
BEGIN
   window := FrontWindow;
   menu := GetMenuHandle(mEdit); {get a handle to the Edit menu}
   IF menu = NIL THEN            {add your own error handling}
      EXIT (MyAdjustEditMenuForModalDialogs);
   undo := MyIsLastActionUndoable(undoText);
   IF undo THEN   {if action can be undone}
   BEGIN
      EnableItem(menu, iUndo);
      SetMenuItemText(menu, iUndo, undoText);
   END
   ELSE           {if action can't be undone}
   BEGIN
      SetMenuItemText(menu, iUndo, gCantUndo);
      DisableItem(menu, iUndo);
   END;
   selection := MySelection(window);
   IF selection THEN
   BEGIN    {enable editing items if there's a selection}
      EnableItem(menu, iCut);
      EnableItem(menu, iCopy);
   END
   ELSE
   BEGIN    {disable editing items if there isn't a selection}
      DisableItem(menu, iCut);
      DisableItem(menu, iCopy);
   END;
   IF MyGetScrap(NIL, 'TEXT', offset) > 0 THEN
      EnableItem(menu, iPaste) {enable if something to paste}
   ELSE
      DisableItem(menu, iPaste);{disable if nothing to paste}
   DisableItem(menu, iSelectAll);
   DisableItem(menu, iCreatePublisher);
   DisableItem(menu, iSubscribeTo);
   DisableItem(menu, iPubSubOptions);
   END;
END;
```

See the chapter “Menu Manager” in this book for more information on menus and the menu bar.

When the user dismisses the alert box or modal dialog box, the Menu Manager restores all menus to their state prior to the appearance of the alert or modal dialog box—unless your application handles its own menu bar access, in which case you must restore the menus to their previous states. You can use a routine similar to `MyAdjustMenus`, shown in Listing 6-21 on page 6-70, to adjust the menus appropriately according to the type of window that becomes the frontmost window.

Adjusting Menus for Movable Modal and Modeless Dialog Boxes

Although it always leaves the Help, Keyboard, and Application menus and their commands enabled, system software does nothing else to manage the menu bar when you display movable modal and modeless dialog boxes. Instead, your application should allow or deny access to the rest of your menus as appropriate to the context. For example, if your application displays a modeless dialog box for a search-and-replace command, you should allow access to the Edit menu to assist the user with the editable text items, and you should allow use of the File menu so that the user can open another file to be searched. However, you should disable other menus if their commands cannot be used inside the active modeless dialog box.

When creating a modeless dialog box, your application should perform the following tasks:

- disable only those menus whose commands are invalid in the current context
- if the modeless dialog box includes editable text items, use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items

When your application creates a movable modal dialog box, it should perform the following tasks:

- leave the Apple menu enabled so that the user can open other applications with it
- if your movable modal dialog box contains editable text items, leave the Edit menu enabled but use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands
- disable all of your other menus

Listing 6-21 on page 6-70 shows an application-defined routine, `MyAdjustMenus`, that SurfWriter uses to adjust its menus after it displays a window or dialog box. You can use a similar routine to adjust your menus as appropriate given the nature of the active window, movable modal dialog box, or modeless dialog box.

Displaying Multiple Alert and Dialog Boxes

You should generally present the user with only one modal dialog box or alert box at a time. Sometimes, you may need to present a modal dialog box and an alert box on the screen at one time. For example, when the user saves a file with the same name as another file, the Standard File Package displays an alert box on top of the standard file dialog box. The alert box asks the user whether to replace the existing file.

Avoid closing a modal dialog box and immediately displaying another modal dialog box or an alert box in response to a user action. This situation creates a "tunneling modal dialog box" effect that might confuse the user. Missing the content of the previous modal dialog box and unable to return to it, the user has difficulty predicting what will happen next.

However, the user should never see more than one modal dialog and one alert box on the screen simultaneously. You can present multiple simultaneous modeless dialog boxes, just as you can present multiple document windows.

When you remove an alert box or a modal dialog box that overlies the default button of a previous alert box, the Dialog Manager doesn't redraw that button's bold outline. Therefore, you should not use an alert box if you need to display another overlapping alert box or dialog box. Instead, you should create a modal dialog box, and you must provide it with an application-defined item that draws the bold outline around the default button. The `ModalDialog` procedure then causes the item to be redrawn after an update event.

In System 7, the Window Manager automatically dims the window frame of a dialog box when you deactivate it to display an alert box, another modal dialog box, or a window. When you deactivate a dialog box, you should use the Control Manager procedure `HiliteControl` to make the controls of a dialog box inactive. You should also draw the outline of the default button of a deactivated dialog box in gray instead of black. Listing 6-16 on page 6-58 shows an application-defined procedure that draws a gray outline when the default button is inactive; Listing 6-34 on page 6-98 shows how to use `HiliteControl` to make buttons inactive and active in response to activate events for a dialog box.

Displaying Alert and Dialog Boxes From the Background

If you ever need to display an alert box or a modal dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. For example, if your application performs lengthy background tasks such as printing many documents or transferring large amounts of data to other computers, you might wish to inform the user that the operation is completed. In these cases, you should post a notification request to notify the user when the operation is completed. Then the Notification Manager automatically displays an alert box containing whatever message you specify; you do not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to ask the user to bring your application to the foreground. The user can then respond to your alert box or modal dialog box. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for information about the Notification Manager.

Including Color in Your Alert and Dialog Boxes

On color monitors, the Dialog Manager automatically adds color to your alert and dialog boxes so that they match the colors of the windows, alert boxes, and dialog boxes used by system software. These colors provide aesthetic consistency across all monitors, from black-and-white displays to 8-bit color displays. On a color monitor, for example, the racing stripes in the title bar of a modeless dialog box are gray, the close box and window frame are in color, and the buttons and text are black.

When you create alert and dialog resources, your application's alert and dialog boxes use the system's default colors. With the following exceptions, creating alert and dialog resources is typically all you need to do to provide color for your alert and dialog boxes:

- When you need to include a color version of an icon in an alert box or a dialog box, you must create a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource. Plate 2 at the front of this book shows an alert box that includes a color icon.
- If you use `GetNewDialog` or `NewDialog` to create a dialog box and you need to produce a blended gray color for outlining the inactive (that is, dimmed) default button, you must create a **dialog color table** (`'dctb'`) **resource** with the same resource ID as the dialog resource.

"Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 explains how to create a draw routine that outlines the default button of a dialog box. If you deactivate a dialog box, you should dim its buttons and use gray to draw the outline for the default button. Because `GetNewDialog` and `NewDialog` supply black-and-white graphics ports for dialog boxes, you can create a dialog color table resource for the dialog box to force the Dialog Manager to supply a color graphics port. Then you can use a blended gray color for the outline for the default button. (`NewColorDialog` supplies a color graphics port.)

Even when you create a dialog color table resource for drawing a gray outline, you should not change the system's default colors. Listing 6-24 shows a dialog color table resource that leaves the default colors intact but forces the Dialog Manager to supply a color graphics port.

Listing 6-24 Rez input for a dialog color table resource using the system's default colors

```
data 'dctb' (kGlobalChangesDialog, purgeable) {
   $"0000 0000 0000 FFFF"  /*use default colors*/
};
```

By using the system's default colors, you ensure that your application's interface is consistent with that of the Finder and other applications. However, if you feel absolutely compelled to break from this consistency, the Dialog Manager offers you the ability to specify colors other than the default colors. Be aware, however, that nonstandard colors in your alert and dialog boxes may initially confuse your users.

Also be aware that despite any changes you make, users can alter the colors of alert and dialog boxes anyway by changing the settings in the Color control panel.

Your application can specify its own colors in an **alert color table** (`'actb'`) **resource** with the same resource ID as the alert resource or in a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource. Both of these resources have exactly the same format as a window color table (`'wctb'`) resource, described in the chapter "Window Manager" in this book.

s **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not specify colors for these elements if you wish to maintain backward compatibility. s

You don't have to call any new routines to change the colors used in alert or dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load a dialog color table resource with the same resource ID as the dialog resource.

Likewise, you can change the system default colors for controls and the color, style, typeface, and size of text used in an alert box or a dialog box by creating an **item color table** (`'ictb'`) **resource** with the same resource ID as the item list resource. You don't have to call any routines to create color items. When you use the `GetNewDialog` function, the Dialog Manager looks first for an item color table resource with the same resource ID as that of the item list resource.

Note
If you want to provide an item color table resource for an alert box or a dialog box, you must create an alert color table resource or a dialog color table resource, even if the item color table resource has no actual color information and describes only static text and editable text style changes. You cannot use an item color table resource to set the font on computers that do not support Color QuickDraw. Also, be aware that changing the default system font makes your application more difficult to localize. u

Even if you provide your own `'dctb'`, `'actb'`, or `'ictb'` resources, you do not need to test whether your application is running on a computer that supports Color QuickDraw in order to use these resources.

Handling Events in Alert and Dialog Boxes

The next two sections explain how the Dialog Manager uses the Control Manager to handle events in controls automatically and how it uses TextEdit to handle events in editable text items automatically. The information in these two sections, "Responding to Events in Controls" and "Responding to Events in Editable Text Items," applies to all alert boxes and all types of dialog boxes: modal, modeless, and movable modal.

To display and handle events in alert boxes, you can use the Dialog Manager functions `Alert`, `NoteAlert`, `CautionAlert`, and `StopAlert`. The Dialog Manager handles all of the events generated by the user until the user clicks a button (typically the OK or Cancel button). When the user clicks a button, the alert box functions invert the button that was clicked, close the alert box, and report the user's selection to your application. Your application is responsible for performing the appropriate action associated with that button. This is described in detail in "Responding to Events in Alert Boxes" beginning on page 6-81.

For modal dialog boxes, you use the `ModalDialog` procedure. The Dialog Manager handles most of the user interaction until the user selects an item. The `ModalDialog` procedure then reports that the user selected an enabled item, and your application is responsible for performing the action associated with that item. Your application typically calls `ModalDialog` repeatedly, responding to clicks on enabled items as reported by `ModalDialog`, until the user clicks OK or Cancel. This is described in detail in "Responding to Events in Modal Dialog Boxes" beginning on page 6-82.

For alert boxes and modal dialog boxes, you should also supply an event filter function as one of the parameters to the alert box functions or the `ModalDialog` procedure. As the user interacts with the alert or modal dialog box, these routines pass events to your event filter function before handling each event. Your event filter function can handle any events not handled by the Dialog Manager or, if necessary, can choose to handle events normally handled by the Dialog Manager. This is described in detail in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86.

To handle events in modeless or movable modal dialog boxes, you can use the `IsDialogEvent` function to determine whether the event occurred while a dialog box was the frontmost window. For every type of event that occurs when the dialog box is active (including null events), `IsDialogEvent` returns `TRUE`; otherwise, it returns `FALSE`. When `IsDialogEvent` returns `TRUE`, you can use the `DialogSelect` function to handle key-down events in editable text items automatically, to handle update and activate events automatically, and to report the enabled items that the user clicks. You then respond appropriately to clicks in your active items.

Alternatively, you can handle events in modeless and movable modal dialog boxes much as you handle events in other windows. That is, when you receive an event you can first determine the type of event that occurred and then take the appropriate action according to which window is in front. If a modeless or movable modal dialog box is in front, you can provide code that takes any actions specific to that dialog box and call the `DialogSelect` function to handle any events that your code doesn't handle. The sections "Responding to Mouse Events in Modeless and Movable Modal Dialog Boxes" beginning on page 6-89, “Responding to Keyboard Events in Modeless and Movable Modal Dialog Boxes” beginning on page 6-94, and “Responding to Activate and Update Events in Modeless and Movable Modal Dialog Boxes” beginning on page 6-97 all take this alternate approach.

Responding to Events in Controls

The Dialog Manager greatly simplifies the work necessary for you to implement buttons, checkboxes, pop-up menus, and radio buttons. For alert boxes and all types of dialog boxes—modal, modeless, and movable modal—the Dialog Manager uses Control Manager routines to display controls automatically, highlight controls appropriately, and report to your application when mouse-down events occur within controls. For example, when the user moves the cursor to an enabled button and holds down the mouse button, the Dialog Manager uses the Control Manager function `TrackControl` to invert the button. When the user releases the mouse button with the enabled button still inverted, the Dialog Manager uses `TrackControl` to report which item was clicked. Your application then responds appropriately—for example, by performing the operation associated with the OK button, by deselecting any other radio button when a radio button is clicked, or by canceling the current operation when the Cancel button is clicked.

For clicks in checkboxes, pop-up menus, and radio buttons, your application usually uses the Control Manager routines `GetControlValue` and `SetControlValue` to get and appropriately set the items’ values. The chapter “Control Manager” in this book explains these routines in detail, but this chapter also offers examples of how to use these routines in your alert and dialog boxes. Because the Control Manager does not know how radio buttons are grouped, it doesn’t automatically turn one off when the user clicks another one. Instead, it’s up to your application to handle this by using the `GetControlValue` and `SetControlValue` routines.

When the user clicks the OK button, your application performs whatever action is necessary according to the values returned by `GetControlValue` for each of the various checkboxes and radio buttons displayed in your alert or dialog box.

When `ModalDialog` and `DialogSelect` call `TrackControl`, they do not allow you to specify any special action procedures necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control that, for example, measures how long the user holds down the mouse button or how far the user has moved an indicator, you can create your own control (or picture or application-defined item that draws a control-like object) in your dialog box. If you use the `ModalDialog` procedure, you must then provide an event filter function that appropriately handles events within that item, and if you use the `DialogSelect` function, you must test for and respond to those events yourself. Alternatively, you can use Window Manager routines to display an appropriate window and then use the Control Manager to create and manage such complex controls yourself. See the chapters “Window Manager” and “Control Manager” in this book for more information.

Responding to Events in Editable Text Items

When the user enters or edits text in an editable text item in your dialog boxes, the Dialog Manager calls TextEdit to handle the events automatically. (You generally shouldn't include editable text items in alert boxes.) You typically disable editable text items because you generally don't need to be informed every time the user types a character or clicks one of them. Instead you need to determine the text only when the OK button is clicked. As illustrated in Listing 6-12 on page 6-49, use `GetDialogItemText` to determine the final value of the editable text item after the user clicks the OK button.

When you use the `ModalDialog` procedure to handle events in modal dialog boxes and when you use the `DialogSelect` function for modeless or movable modal dialog boxes, the Dialog Manager calls TextEdit to handle keystrokes and mouse actions within editable text items, so that

- when the user clicks the item, a blinking vertical bar appears that indicates an insertion point where text may be entered
- when the user drags over text in the item, the text is highlighted; when the user double-clicks a word, the word is highlighted; the highlighted selection is then replaced by what the user types
- when the user holds down the Shift key while clicking or dragging, the highlighted selection is extended or shortened appropriately
- when the user presses the Backspace key, the highlighted selection or the character preceding the insertion point is deleted
- when the user presses the Tab key, the cursor automatically advances to the next editable text item in the item list resource, wrapping around to the first if there are no more items

If your modeless or movable modal dialog box contains any editable text items, call `DialogSelect` even when `WaitNextEvent` returns `FALSE`. This is necessary because the `DialogSelect` function calls the `TEIdle` procedure to make the text cursor blink within your editable text items during null events; otherwise, the text cursor will not blink. Listing 6-25 illustrates an application-defined routine, `DoIdle`, that calls `DialogSelect` whenever the application receives null events while its modeless dialog box is the frontmost window.

Listing 6-25 Using `DialogSelect` during null events

```
PROCEDURE DoIdle (event: EventRecord);
VAR
   window:     WindowPtr;
   windowType: Integer;
   itemHit:    Integer;
   result:     Boolean;
BEGIN
   window := FrontWindow;
   {determine which type of window--document, }
   { modeless dialog box, etc.--is in front}
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:  {document window is frontmost}
      ;  {see examples in "Event Manager" chapter}
   kMyGlobalChangesModelessDialog:  {modeless dialog is frontmost}
      result := DialogSelect(event, window, itemHit);
   END; {of CASE}
END;
```

Generally, your application should handle menu bar access when you display dialog boxes containing editable text items. Leave your Edit menu enabled, and use the `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` procedures to support the Cut, Copy, Paste, and Clear commands and their keyboard equivalents. You should also provide your own code to support the Undo command. "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 describe how to allow users to access your Edit menu when you display dialog boxes.

If you don't supply your own event filter function and the user presses the Return or Enter key while a modal dialog box is onscreen, the Dialog Manager treats the event as a click on the default button (that is, the first item in the list) regardless of whether the dialog box contains an editable text item. If your event filter function responds to the user pressing Return and Enter by moving the cursor in editable text items, don't display a bold outline around any buttons. If your event filter function responds to the user pressing Return and Enter as if the user clicks the default button, then you should display a bold outline around the default button. See "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86 for an example of how to map the Return and Enter keys to the default button in your dialog boxes.

Initially, an editable text item may contain default text or no text. You can provide default text either by specifying a text string as the last element for that item in the item list resource or by using the `SetDialogItemText` procedure, which is described on page 6-131.

When a dialog box that contains editable text items is first displayed, the insertion point usually appears in the first editable text item in the item list resource. You may instead want to use the `SelectDialogItemText` procedure so that the dialog box appears with text selected, or so that an insertion point or a text selection reappears if the user makes an error while entering text. For example, the user who accidentally types nonnumeric input when a number is required can be given the opportunity to type the entry again. The `SelectDialogItemText` procedure is described in detail on page 6-131.

By default, the Dialog Manager displays editable text items in the system font. To maintain visual consistency across applications for your users and to make it easier to localize your application, you should not change the font or font size.

Responding to Events in Alert Boxes

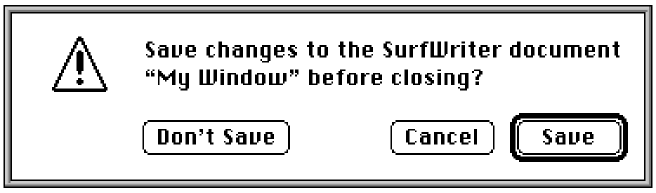
After displaying an alert box or playing an alert sound, the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions call the `ModalDialog` procedure to handle events automatically for you.

The `ModalDialog` procedure, in turn, gets each event by calling the Event Manager function `GetNextEvent`. If the event is a mouse-down event outside the content region of the alert box, `ModalDialog` emits the system alert sound and gets the next event.

The `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions continue calling `ModalDialog` until the user selects an enabled control (typically a button). At this time these functions remove the alert box from the screen and return the item number of the selected control. Your application then responds as appropriate for a click on this item.

For example, the code that supports the alert box displayed in Figure 6-39 must respond to three different events—one for each button that the user may click.

Figure 6-39 Three buttons for which `CautionAlert` reports events


Save changes to the SurfWriter document “My Window” before closing?

Listing 6-9 on page 6-47 shows an application-defined routine, named `MyCloseDocument`, for the Close command. If the document has been modified since the last save, `MyCloseDocument` displays the alert box illustrated in Figure 6-39 before closing the window. After `MyCloseDocument` displays the caution alert, it tests for the item number that `CautionAlert` returns after it removes the alert box. If the user clicks the Save button, `CautionAlert` returns its item number, and `MyCloseDocument` calls other application-defined routines to save the file, close the file, and close the window. If the user clicks the Don’t Save button, `MyCloseDocument` closes the window without saving the file. The only other possible response is for the user to click the Cancel button, in which case `MyCloseDocument` does nothing—the Dialog Manager removes the alert box, and `MyCloseDocument` simply leaves the document window as it is.

The standard event filter function allows users to press the Return or Enter key in lieu of clicking the default button. When one of these keys is pressed, the standard event filter function returns `TRUE` to `ModalDialog`, which in turn causes `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` to return the item number of the default button. When you write your own event filter function, it should emulate the standard filter function by responding in this way to keyboard events involving the Return and Enter keys.

For events inside the alert box, `ModalDialog` passes the event to an event filter function before handling the event. The event filter function provides a secondary event-handling loop for handling events that `ModalDialog` doesn't handle and for overriding events that `ModalDialog` would otherwise handle. You should provide a simple event filter function for every alert box and modal dialog box in your application.

You specify a pointer to your event filter function in the second parameter to the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. In the `MyCloseDocument` routine shown on page 6-47, a pointer to the `MyEventFilter` function is specified. In most cases, you can use the same event filter function in every one of your alert and modal dialog boxes. An example of a simple event filter function that allows background applications to receive update events and performs the other necessary event handling is provided in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86.

Unless your event filter function handles the event in its own way and returns `TRUE`, `ModalDialog` handles the event inside the alert box as follows:

- In response to an activate or update event for the alert box, `ModalDialog` activates or updates its window.
- If the user presses the mouse button while the cursor is in a control, the Control Manager function `TrackControl` tracks the mouse. If the user releases the mouse button while the cursor is in an enabled control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the control's item number. (Generally, buttons should be the only controls you use in alert boxes.)
- If the user presses the mouse button while the cursor is in any enabled item other than a control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the item number. (Generally, button controls should be the only enabled items in alert boxes.)
- If the user presses the mouse button while the cursor is in a disabled item, or if it is in no item, or if any other event occurs, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` do nothing.

Responding to Events in Modal Dialog Boxes

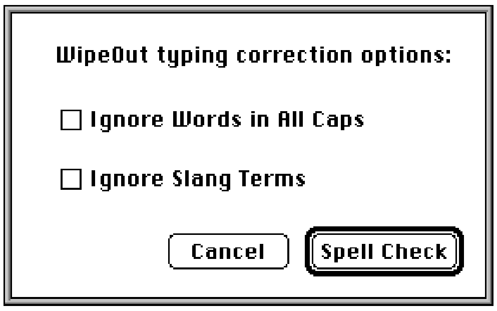
Call the `ModalDialog` procedure immediately after displaying a modal dialog box. This procedure repeatedly handles events inside the modal dialog box until an event involving an enabled item—such as a click in a radio button—occurs. If the event is a mouse-down event outside the content region of the dialog box, `ModalDialog` emits the system alert sound and gets the next event. After receiving an event involving an enabled item, `ModalDialog` returns the item number. Normally you then do whatever is appropriate in response to an event in that item. Your application should continue calling `ModalDialog` until the user selects the OK or Cancel button, at which point your application should close the dialog box.

For example, if the user clicks a radio button, your application should get the value of that button, turn off any other selected radio button within its group, and call `ModalDialog` again to get the next event. If the user clicks the Cancel button, your application should restore the user's work to its state just before the user invoked the dialog box, and then your application should remove the dialog box from the screen.

Note
Do not use `ModalDialog` for modeless or movable modal dialog boxes. u

The code that supports the modal dialog box shown in Figure 6-40 must respond to events in four controls: two checkboxes and two buttons.

Figure 6-40 Four items for which `ModalDialog` reports events



Listing 6-26 illustrates an application-defined routine, `MySpellCheckDialog`, that responds to events in these four controls.

Listing 6-26 Responding to events in a modal dialog box

```
FUNCTION MySpellCheckDialog: OSErr;
VAR
   docWindow:           WindowPtr;
   ignoreCapsCheck:     Boolean;
   ignoreSlangCheck:    Boolean;
   spellDialog:         DialogPtr;
   itemHit, itemType:   Integer;
   itemHandle:          Handle;
   itemRect:            Rect;
   capsVal:             Integer;
   slangVal:            Integer;
   event:               EventRecord;
BEGIN
   capsVal := 0;
   slangVal := 0;
   ignoreCapsCheck := FALSE;
   ignoreSlangCheck := FALSE;
   MySpellCheckDialog := kSuccess;{assume success}
   docWindow := FrontWindow;     {get front window}
   IF docWindow <> NIL THEN
      DoActivate(docWindow, FALSE, event);   {deactivate document window}
   spellDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
   IF spellDialog = NIL THEN
   BEGIN
      MySpellCheckDialog := kFailed;
      Exit(MySpellCheckDialog);
   END;
   MyAdjustMenus;             {adjust menus as needed}
   GetDialogItem(spellDialog, kUserItem, itemType, itemHandle, itemRect);
   SetDialogItem(spellDialog, kUserItem, itemType,
                 Handle(@MyDrawDefaultButtonOutline), itemRect);
   ShowWindow(spellDialog);   {show dialog box with default button outlined}
   REPEAT
      ModalDialog(@MyEventFilter, itemHit);  {get events}
      IF itemHit = kAllCaps THEN    {user clicked Ignore Words in All Caps}
      BEGIN
         {get the control handle to the checkbox}
         GetDialogItem(spellDialog, kAllCaps, itemType, itemHandle,
                       itemRect);
         {get the last value of the checkbox}
         capsVal := GetControlValue(ControlHandle(itemHandle));
         {toggle the value of the checkbox}
         capsVal := 1 - capsVal;
         {set the checkbox to the new value}
         SetControlValue(ControlHandle(itemHandle), capsVal);
      END;
      IF itemHit = kSlang THEN   {user clicked Ignore Slang Terms}
      BEGIN
         {get checkbox's handle, get its value, toggle it, then reset it}
         GetDialogItem(spellDialog, kSlang, itemType, itemHandle, itemRect);
         slangVal := GetControlValue(ControlHandle(itemHandle));
         slangVal := 1 - slangVal;
         SetControlValue(ControlHandle(itemHandle), slangVal);
      END;
   UNTIL ((itemHit = kSpellCheck) OR (itemHit = kCancel));
   DisposeDialog(spellDialog);      {close the dialog box}
   IF itemHit = kSpellCheck THEN    {user clicked Spell Check button}
   BEGIN
      IF capsVal = 1 THEN           {user wants to ignore all caps}
         ignoreCapsCheck := TRUE;
      IF slangVal = 1 THEN          {user wants to ignore slang}
         ignoreSlangCheck := TRUE;
      {now start the spell check}
      SpellCheckMyDoc(ignoreCapsCheck, ignoreSlangcheck);
   END;
END;
```

The `MySpellCheckDialog` routine calls `ModalDialog` immediately after using `GetNewDialog` to create and display the dialog box. The `MySpellCheckDialog` routine repeatedly responds to events in the two checkboxes until the user clicks either the Spell Check or the Cancel button. When the user clicks either of the checkboxes (which are the third and fourth items in the item list resource), `MySpellCheckDialog` uses the `GetDialogItem` procedure to get a handle to the checkbox. The `MySpellCheckDialog` routine coerces this handle to a control handle and passes it to the Control Manager function `GetControlValue` to get the last value of the control (1 if the checkbox was selected or 0 if it was unselected). Subtracting this value from 1, `MySpellCheckDialog` derives a new value for the control. Then `MySpellCheckDialog` passes this value to the Control Manager procedure `SetControlValue` to set the new value. The Control Manager responds by drawing an X in the box if the value of the control is 1 or removing the X if the value of the control is 0.

As soon as the user clicks the Spell Check or Cancel button (which are the first and second items in the item list resource), `MySpellCheckDialog` stops responding to events in the checkboxes. This routine uses the `DisposeDialog` procedure (which is explained in “Closing Dialog Boxes” beginning on page 6-100) to remove the dialog box. If the user clicks the Cancel button, `MySpellCheckDialog` does no further processing of the information in the dialog box. If, however, the user clicks the Spell Check button, `MySpellCheckDialog` calls another application-defined routine, `SpellCheckMyDoc`, to check the document for spelling errors according to the preferences that the user communicated in the checkboxes.

For events inside the dialog box, `ModalDialog` passes the event to an event filter function before handling the event. In this example, the application specifies a pointer to its own event filter function, `MyEventFilter`. As described in the next section, your application should provide an event filter function. You can use the same event filter function in most or all of your alert and modal dialog boxes.

Unless your event filter function handles the event and returns `TRUE`, `ModalDialog` handles the event as follows:

- In response to an activate or update event for the dialog box, `ModalDialog` activates or updates its window.
- If the user presses the mouse button while the cursor is in an editable text item, `ModalDialog` responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If a key-down event occurs and there’s an editable text item, text entry and editing are handled as described in “Responding to Events in Editable Text Items” beginning on page 6-79. If the editable text item is enabled, `ModalDialog` returns its item number after it receives either the mouse-down or key-down event. Normally, editable text items are disabled, and you use the `GetDialogItemText` procedure to read the information in the items only after the user clicks the OK button. Listing 6-12 on page 6-49 illustrates this technique.

- If the user presses the mouse button while the cursor is in a control, `ModalDialog` calls the Control Manager function `TrackControl`. If the user releases the mouse button while the cursor is in an enabled control, `ModalDialog` returns the control's item number. Your application should respond appropriately; for example, Listing 6-26 uses an application-defined routine that checks the spelling of a document when the user clicks the Spell Check button.
- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `ModalDialog` returns the item's number, and your application should respond appropriately. Generally, only controls should be enabled. If your application creates a complex control—such as one that measures how far a dial is moved—your application must provide an event filter function to handle mouse events in that item.
- If the user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `ModalDialog` does nothing.

Writing an Event Filter Function for Alert and Modal Dialog Boxes

For alert and modal dialog boxes, the Dialog Manager provides a standard event filter function that checks whether the user has pressed the Enter or Return key and, if so, returns the item number of the default button. In early versions of Macintosh system software, when a single application controlled the computer, the standard event filter function for alert boxes and most modal dialog boxes was usually sufficient. However, because the standard event filter function does not permit background applications to receive or respond to update events, it is no longer sufficient.

Thus, your application should provide a simple event filter function that performs these functions and also allows inactive windows to receive update events. You can use the same event filter function in most or all of your alert and modal dialog boxes.

You can also use your event filter function to handle other events that `ModalDialog` doesn't handle—such as the Command-period key-down event, disk-inserted events, keyboard equivalents, and mouse-down events (if necessary) for application-defined items that you provide.

For example, the standard event filter function ignores key-down events for the Command key. When your application allows the user to access your menus after you display a dialog box, your event filter function should handle keyboard equivalents for menu commands and return TRUE.

At a minimum, your event filter function should perform the following tasks:

- return TRUE and the item number for the default button if the user presses the Return or Enter key
- return TRUE and the item number for the Cancel button if the user presses the Esc key or the Command-period key combination
- update your windows in response to update events (this also allows background applications to receive update events) and return FALSE
- return FALSE for all events that your event filter function doesn't handle

You can also use the event filter function to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor in an application-defined item. For example, if you provide an application-defined item that requires you to measure how long the user holds down the mouse button or how far the user drags the cursor, use the event filter function to handle events inside that item.

If it seems that you will spend time replicating much of your primary event loop in this event filter function, you might consider handling all the events in your main event loop instead of using the Dialog Manager's `Alert`, `NoteAlert`, `StopAlert`, and `CautionAlert` functions or `ModalDialog` procedure.

Your own event filter function should have three parameters and return a Boolean value. For example, this is how to declare an event filter function named `MyEventFilter`:

```
FUNCTION MyEventFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

After receiving an event that it does not handle, your function should return `FALSE`. When your function returns `FALSE`, `ModalDialog` handles the event, which you pass in the parameter `theEvent`. (Your function can also change the event to simulate a different event and return `FALSE`, which passes the altered event to the Dialog Manager for handling.) If your function does handle the event, your function should return `TRUE` as a function result and, in the `itemHit` parameter, the number of the item that it handled. The `ModalDialog` procedure and, in turn, the `Alert`, `NoteAlert`, `StopAlert`, and `CautionAlert` functions then return this item number in their own `itemHit` parameter.

Because `ModalDialog` calls the `GetNextEvent` function with a mask that excludes disk-inserted events, your event filter function can call the Event Manager procedure `SetSystemEventMask` to accept disk-inserted events. See the chapter "Event Manager" in this book for a discussion about handling disk-inserted events.

For alert and modal dialog boxes, the Dialog Manager provides a standard event filter function that checks whether the user has pressed the Enter or Return key and, if so, returns the item number of the default button. Your event filter function should always check whether the Return key or Enter key was pressed and, if so, return the item number of the default button in the `itemHit` parameter and a function result of `TRUE`. Your event filter function should also check whether the Esc key was pressed and, if so, return the item number for the Cancel button in the `itemHit` parameter and a function result of `TRUE`. Your event filter function should also respond to the Command-period key-down event as if the user had clicked the Cancel button.

To give visual feedback indicating which item has been selected, you should invert buttons that are activated by keyboard equivalents for all alert and dialog boxes. A good rule of thumb is to invert a button for 8 ticks, long enough to be noticeable but not so long as to be annoying. The Control Manager performs this action whenever a user clicks a button, and your application should do this whenever a user presses the keyboard equivalent of a button click.

For modal dialog boxes that contain editable text items, your application should handle menu bar access to allow use of your Edit menu and its Cut, Copy, Paste, Clear, and Undo commands, as explained in "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68. Your event filter function should then test for and handle mouse-down events in the menu bar and key-down events for keyboard equivalents of Edit menu commands. Your application should respond to users' choices from the Edit menu by using the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands.

Listing 6-27 shows `MyEventFilter`, which begins by handling update events in windows other than the alert or dialog box. (By responding to update events for your application's own inactive windows in this way, you allow `ModalDialog` to perform a minor switch when necessary so that background applications can update their windows, too.)

Next, `MyEventFilter` handles activate events**.** This event filter function then handles key-down events for the Return and Enter keys as if the user had clicked the default button, and it handles key-down events for the Esc key as if the user had clicked the Cancel button. (See *Inside Macintosh: Text* for information about character codes for the Return, Enter, and Esc keys.) Your event filter function can then include tests for other events, such as disk-inserted events and keyboard equivalents.

Listing 6-27 A typical event filter function for alert and modal dialog boxes

```
FUNCTION MyEventFilter(theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
VAR
   key:        Char;
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
   finalTicks: LongInt;
BEGIN
   MyEventFilter := FALSE; {assume Dialog Mgr will handle it}
   IF (theEvent.what = updateEvt) AND
      (WindowPtr(theEvent.message) <> theDialog) THEN
      DoUpdate(WindowPtr(theEvent.message))  {update the window behind}
   ELSE IF (theEvent.what = activateEvt) AND (WindowPtr(theEvent.message)
            <> theDialog) THEN
      DoActivate(WindowPtr(theEvent.message),
                  (BAnd(theEvent.modifiers, activeFlag) <> 0), theEvent)
   ELSE
      CASE theEvent.what OF
         keyDown, autoKey:    {user pressed a key}
         BEGIN
            key := Char(BAnd(theEvent.message, charCodeMask));
            IF (key = Char(kReturnKey)) OR (key = Char(kEnterKey)) THEN
            BEGIN    {respond as if user clicked Spell Check}
               GetDialogItem(theDialog, kSpellCheck, itemType, itemHandle,
                             itemRect);
                     {invert the Spell Check button for user feedback}
               HiliteControl(ControlHandle(itemHandle), inButton);
               Delay(kVisualDelay, finalTicks); {invert button for 8 ticks}
               HiliteControl(ControlHandle(itemHandle), 0);
               myEventFilter := TRUE;  {event's being handled}
               itemHit := kSpellCheck; {return the default button}
            END;
            IF (key = Char(kEscapeKey)) OR   {user pressed Esc key}
               (Boolean(BAnd(theEvent.modifiers, cmdKey)) AND
               (key = Char(kPeriodKey))) THEN   {user pressed Cmd-pd}
            BEGIN    {handle as if user clicked Cancel}
               GetDialogItem(theDialog, kCancel, itemType, itemHandle,
                             itemRect);
                     {invert the Cancel button for user feedback}
               HiliteControl(ControlHandle(itemHandle), inButton);
               Delay(kVisualDelay, finalTicks); {invert button for 8 ticks}
               HiliteControl(ControlHandle(itemHandle), 0);
               MyEventFilter := TRUE;  {event's being handled}
               itemHit := kCancel;  {return the Cancel button}
            END;  {of Cancel}
            {handle any other keyboard equivalents here}
         END;  {of keydown, autokey}
         {handle disk-inserted and other events here, as needed}
         OTHERWISE
      END;  {of CASE}
END;
```

To use this event filter function for an alert box, the application specifies a pointer to `MyEventFilter` when it calls one of the `Alert` functions, as shown in Listing 6-19 on page 6-66. To use this event filter function for a modal dialog box, the application specifies a pointer to `MyEventFilter` when it calls `ModalDialog`, as shown in Listing 6-26 on page 6-83.

Responding to Mouse Events in Modeless and Movable Modal Dialog Boxes

To handle events in modeless and movable modal dialog boxes, you can use the `IsDialogEvent` function to determine when events occur while a dialog box is the frontmost window. For such events, you can then use the `DialogSelect` function to handle key-down events in editable text items automatically, to handle update and activate events automatically, and to report the enabled items that the user clicks. You must also use additional Toolbox routines to handle other types of keyboard events and other events in the dialog box.

S **WARNING**
The `IsDialogEvent` and `DialogSelect` functions are unreliable when running in versions of system software previous to System 7. s

Alternatively, and probably most efficiently, your application can respond to events in modeless and movable modal dialog boxes by first determining the type of event that occurred and then taking the appropriate action according to which type of window is in front. If a modeless or movable modal dialog box is in front, you can provide code that takes any actions specific to that dialog box. You can then use the `DialogSelect` function instead of the Control Manager functions `FindControl` and `TrackControl` to handle mouse events in your dialog boxes. The `DialogSelect` function also handles update events, activate events, and events in editable text items. (If your modeless or movable modal dialog box contains editable text items, you should call `DialogSelect` during null events to cause the text cursor to blink.)

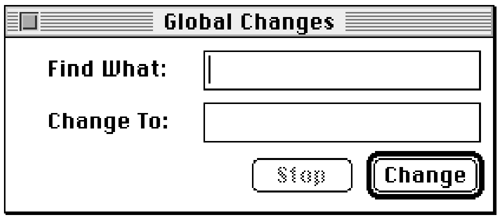
If you choose to determine whether events involve movable modal or modeless dialog boxes without the aid of the `IsDialogEvent` function, your application should be prepared to handle the following mouse events:

- clicks in the menu bar, which your application has adjusted as appropriate for the dialog box. Be sure to use the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items in your dialog boxes.
- clicks in the content region of an active movable modal or modeless dialog box. You can use the `DialogSelect` function to aid you in handling the event.
- clicks in the content region of an inactive modeless dialog box. In this case, your application should make the modeless dialog box active by making it the front-most window.
- clicks in the content region of an inactive window whenever a movable modal or modeless dialog box is active. For movable modal dialog boxes, your application should emit the system alert sound, whereas for modeless dialog boxes, your application should bring the inactive window to the front.
- mouse-down events in the drag region (that is, the title bar) of an active movable modal or modeless dialog box. Your application should use the Window Manager procedure `DragWindow` to move the dialog box in response to the user's actions.
- mouse-down events in the drag region of an inactive window when a movable modal dialog box is active. Your application should *not* move the inactive window in response to the user's actions. Instead, your application should play the system alert sound.
- clicks in the close box of a modeless dialog box. Your application should dispose of or hide the modeless dialog box, whichever action is more appropriate.

Figure 6-41 shows a simple modeless dialog box with editable text items.

Listing 6-28 illustrates an application-defined procedure that handles mouse-down events for all windows, including the modeless dialog box shown in Figure 6-41.

Figure 6-41 A modeless dialog box for which `DialogSelect` reports events



Listing 6-28 Handling mouse-down events for all windows

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:             Integer;
   thisWindow:       WindowPtr;
BEGIN
   {find general location of the cursor at the time of mouse-down event}
   part := FindWindow(event.where, thisWindow);
   CASE part OF   {take action based on the cursor location}
   inMenuBar: ;   {cursor in menu bar; respond with Menu Manager routines}
   inSysWindow: ; {cursor in a DA; use SystemClick here}
   inContent:     {cursor in the content area of one of this app's windows}
      IF thisWindow <> FrontWindow THEN
      BEGIN {mouse-down in a window other than the front }
            { window--make the clicked window the front window, }
            { unless the front window is a movable modal dialog box}
         IF MyIsMovableModal(FrontWindow) THEN
            SysBeep(30)    {emit system alert sound}
         ELSE
            SelectWindow(thisWindow);
      END
      ELSE  {mouse-down in the content area of front window}
         DoContentClick(thisWindow, event);
   inDrag:                 {handle mouse-down in drag area}
      IF (thisWindow <> FrontWindow) AND (MyIsMovableModal(FrontWindow))
      THEN
         SysBeep(30)    {emit system alert sound}
      ELSE
        DragWindow(thisWindow, event.where, GetGrayRgn^^.rgnBBox);
   inGrow: ;            {handle mouse-down in zoom box here}
   inGoAway:            {handle mouse-down in close box here}
      IF TrackGoAway(thisWindow, event.where) THEN
         DoCloseCmd;
   inZoomIn, inZoomOut: ;  {handle zoom box region for standard windows}
   END;     {end of CASE}
END;  {of DoMouseDown}
```

The DoMouseDown routine first uses the Window Manager function FindWindow to determine approximately where the cursor is when the mouse button is pressed. When the user presses the mouse button while the cursor is in the content area of a window, DoMouseDown first checks whether the mouse-down event occurs in the currently active window by comparing the window pointer returned by FindWindow with that returned by the Window Manager function FrontWindow.

When the mouse-down event occurs in an inactive window, DoMouseDown uses another application-defined routine, MyIsMovableModal, to check whether the active window is a movable modal dialog box. If so, DoMouseDown plays the system alert sound. Otherwise, DoMouseDown uses the Window Manager procedure SelectWindow to make the selected window active. (Although not illustrated in this book, the MyIsMovableModal routine uses the Window Manager function GetWVariant to determine whether the variation code for the front window is movableDBoxProc. If so, MyIsMovableModal returns TRUE.) See the chapter “Window Manager” in this book for more information about the SelectWindow and GetWVariant routines.

As in this example, you must ensure that the movable dialog box is modal within your application. That is, the user should not be able to switch to another of your application’s windows while the movable modal dialog box is active. Instead, your application should emit the system alert sound. Notice as well that when the mouse-down event occurs in the drag region of any window, DoMouseDown checks whether the drag region belongs to an inactive window while a movable modal dialog box is active. If it does, DoMouseDown again plays the system alert sound. (However, by clicking other applications’ windows or by selecting other applications from the Application and Apple menus, users should be able to switch your application to the background when you display a movable modal dialog box—an action users cannot perform with fixed-position modal dialog boxes.)

If a user presses the mouse button while the cursor is in the content region of the active window, DoMouseDown calls another application-defined routine, DoContentClick, to further handle mouse events. Listing 6-29 shows how this routine in turn uses the DialogSelect function to handle the mouse-down event after the application determines that it occurs in the modeless dialog box shown in Figure 6-41 on page 6-91.

Listing 6-29 Using the DialogSelect function for responding to mouse-down events

```
PROCEDURE DoContentClick (thisWindow: windowPtr; event: EventRecord);
VAR
   itemHit:    Integer;
   refCon:     Integer;
BEGIN
   windowType := MyGetWindowType(thisWindow);
   CASE windowType OF
      kMyDocWindow: ;
         {handle clicks in document window here; see the chapter "Control }
         { Manager" for sample code for this case}
      kGlobalChangesID:    {user clicked Global Changes dialog box}
         BEGIN
            IF DialogSelect(event, DialogPtr(thisWindow), itemHit) THEN
            BEGIN
               IF itemHit = kChange THEN     {user clicked Change}
                  ;  {use GetDialogItem and GetDialogItemText to get }
                     { the text strings and replace one string with the }
                     { other here}
               IF itemHit = kStop THEN    {user clicked Stop}
                  ;  {stop making changes here}
            END;
         END; {of CASE for kGlobalChangesID}
      {handle other window types here}
   END;  {of CASE}
END;
```

In this example, when the user clicks the Change button, `DialogSelect` returns its item number. Within the user's document, the application then performs a global search and replace. (Listing 6-12 on page 6-49 illustrates how an application can use the `GetDialogItem` and `GetDialogItemText` procedures for this purpose.) Generally, only controls should be enabled in a dialog box; therefore, your application normally responds only when `DialogSelect` returns `TRUE` after the user clicks an enabled control. For example, if the event is an activate or update event for a dialog box, `DialogSelect` activates or updates it and returns `FALSE`, so your application does not need to respond to the event.

At this point, you may also want to check for and respond to any special events that you do not wish to pass to `DialogSelect` or that require special processing before you pass them to `DialogSelect`. You would need to do this, for example, if the dialog box needs to respond to disk-inserted events.

IMPORTANT

When `DialogSelect` calls `TrackControl`, it does not allow you to specify any action procedures necessary for a more complex control—for example, a control that measures how long the user holds down the mouse button or one that measures how far the user has moved an indicator. For instances like this, you can create a picture or an application-defined item that draws a control-like object; you must then test for and respond to those events yourself before passing events to `DialogSelect`. Or, you can use the Control Manager functions `FindControl` and `TrackControl` to process the mouse events inside the controls of your dialog box. s

Listing 6-28 on page 6-91 calls one of its application-defined routines, `DoCloseCmd`, whenever the user clicks the close box of the active window. If the active window is a modeless dialog box, you might find it more efficient to hide the window rather than remove its data structures. Listing 6-30 shows how you can use the Window Manager routine `HideWindow` to hide the Global Changes modeless dialog box when the user clicks its close box. The next time the user chooses the Global Changes command, the dialog box is already available, in the same location and with the same text selected as when it was last used. (Listing 6-20 on page 6-67 illustrates how first to create and later redisplay this modeless dialog box.)

Listing 6-30 Hiding a modeless dialog box in response to a Close command

```
PROCEDURE DoCloseCmd;
VAR
   myWindow: WindowPtr;
   myData: MyDocRecHnd;
   windowType: Integer;
BEGIN
   myWindow := FrontWindow;
   windowType := MyGetWindowType(myWindow);
   CASE windowType OF
      kMyGlobalChangesModelessDialog:
         HideWindow(myWindow);
      kMySpellModelessDialog:
         HideWindow(myWindow);
      kMyDocWindow:
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(myWindow));
            MyCloseDocument(myData);
         END;  {of kMyDocWindow case}
      kDAWindow:
         CloseDeskAcc(WindowPeek(myWindow)^.windowKind);
   END; {of CASE}
END;
```

Responding to Keyboard Events in Modeless and Movable Modal Dialog Boxes

If you adopt the previously described strategy of determining—without the aid of the `IsDialogEvent` function—whether events involve movable modal or modeless dialog boxes, your application should be prepared to handle the following keyboard events:

- keyboard equivalents, such as Command-C to copy, to which your application should respond appropriately
- key-down events for the Return and Enter keys, to which your application should respond as if the user had clicked the default button
- key-down events for the Esc or Command-period keystrokes, to which your application should respond as if the user had clicked the Cancel button
- key-down and auto-key events in editable text items, for which your application can use the `DialogSelect` function, which in turn calls TextEdit to handle keystrokes within editable text items automatically

Listing 6-31 illustrates how an application can check for keyboard equivalents whenever it receives key-down events. If the user holds down the Command key while pressing another key, the application calls another of its application-defined procedures, `DoMenuCommand`, which handles keyboard equivalents for menu commands. See the chapter "Menu Manager" in this book for an example of a `DoMenuCommand` procedure. Remember that when a movable modal dialog box or a modeless dialog box is active, your application should adjust the menus appropriately, and use the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items.

Listing 6-31 Checking for key-down events involving the Command key

```
PROCEDURE DoKeyDown (event: EventRecord);
VAR
   key:  Char;
BEGIN
   key := CHR(BAnd(event.message, charCodeMask));
   IF BAnd(event.modifiers, cmdKey) <> 0 THEN
   BEGIN                      {Command key down}
      IF event.what = keyDown THEN
      BEGIN
         MyAdjustMenus;             {adjust the menus as needed}
         DoMenuCommand(MenuKey(key));  {handle the menu command}
      END;
   END
   ELSE
      MyHandleKeyDown(event);
END;
```

After determining that a key-down event does not involve a keyboard equivalent, Listing 6-31 calls another of its own routines, `MyHandleKeyDown`, which is shown in Listing 6-32.

Listing 6-32 Checking for key-down events in a modeless dialog box

```
PROCEDURE MyHandleKeyDown (event: EventRecord);
VAR
   window:        WindowPtr;
   windowType:    Integer;
BEGIN
   window := FrontWindow;
   {determine the type of window--document, modeless, etc.}
```

```
      windowType := MyGetWindowType(window);
      IF windowType = kMyDocWindow THEN   {key-down in doc window}
      BEGIN    {handle keystrokes in document window here}
      END
      ELSE     {key-down in modeless dialog box}
         MyHandleKeyDownInModeless(event, windowType);
   END;
```

The `MyHandleKeyDown` routine determines what type of window is active when the user presses a key. If a modeless dialog box is the frontmost window, `MyHandleKeyDown` automatically calls another application-defined routine, `MyHandleKeyDownInModeless`, to respond to key-down events in modeless dialog boxes. The `MyHandleKeyDownInModeless` routine is shown in Listing 6-33.

Listing 6-33 Responding to key-down events in a modeless dialog box

```
PROCEDURE MyHandleKeyDownInModeless(event: EventRecord; windowType: Integer);
VAR
   key:        Char;
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
   finalTicks: LongInt;
   handled:    Boolean;
   item:       Integer;
   theDialog:  DialogPtr;
BEGIN
   handled := FALSE;
   theDialog := FrontWindow;
   CASE windowType OF
      kGlobalChangesID:    {key-down in Global Changes dialog box}
      BEGIN
         key := Char(BAnd(event.message, charCodeMask));
         IF (key = Char(kReturnKey)) OR (key = Char(kEnterKey)) THEN
         BEGIN    {respond as if user clicked Change}
            GetDialogItem(theDialog, kChange, itemType, itemHandle,
                          itemRect);
                  {invert the Change button for 8 ticks for user feedback}
            HiliteControl(ControlHandle(itemHandle), inButton);
            Delay(kVisualDelay, finalTicks);
            HiliteControl(ControlHandle(itemHandle), 0);
               {use GetDialogItem and GetDialogItemText to get the text }
               { strings and replace one string with the other here}
            handled := TRUE;  {event's been handled}
         END;
         IF (key = Char(kEscapeKey)) OR   {user pressed Esc key}
            (Boolean(BAnd(event.modifiers, cmdKey)) AND
               (key = Char(kPeriodKey))) THEN   {user typed Cmd-pd}
         BEGIN    {handle as if user clicked Stop}
            GetDialogItem(theDialog, kStop, itemType, itemHandle,
                          itemRect);
                  {invert the Stop button for 8 ticks for user feedback}
            HiliteControl(ControlHandle(itemHandle), inButton);
            Delay(kVisualDelay, finalTicks);
            HiliteControl(ControlHandle(itemHandle), 0);
               {cancel the current operation here}
            handled := TRUE;  {event's been handled}
         END;
         IF NOT handled THEN  {let DialogSelect handle keydown events in }
                              { editable text items}
            handled := DialogSelect(event, theDialog, item);
      END;  {of case kGlobalChangesID}
   {handle other modeless and movable modal dialog boxes here}
   END;  {of CASE}
END;
```

When `MyHandleKeyDownInModeless` determines that the front window is the Global Changes modeless dialog box, it checks whether the user pressed Return or Enter. If so, `MyHandleKeyDownInModeless` responds as if the user had clicked the default button: Change. The `MyHandleKeyDownInModeless` routine uses the Control Manager procedure `HiliteControl` to highlight the Change button for 8 ticks. (Listing 6-27 on page 6-88 illustrates how to use `HiliteControl` to highlight the button from within a modal dialog box's event filter function.)

When the user presses Esc or Command-period, `MyHandleKeyDownInModeless` responds as if the user had clicked the Cancel button.

Finally, `MyHandleKeyDownInModeless` uses the `DialogSelect` function, which in turn calls TextEdit to handle keystrokes within editable text items.

Responding to Activate and Update Events in Modeless and Movable Modal Dialog Boxes

If you adopt the previously described strategy of determining—without the aid of the `IsDialogEvent` function—whether events involve movable modal or modeless dialog boxes, your application should be prepared to handle activate and update events for both movable modal and modeless dialog boxes. You can use `DialogSelect` to assist you in handling activate and update events. For faster performance, you may instead want to use the `UpdateDialog` function when handling update events. Both `DialogSelect` and `UpdateDialog` use the QuickDraw procedure `SetPort` to make the dialog box the current graphics port before redrawing or updating it.

You should use the Control Manager procedure `HiliteControl` to make the buttons and other controls inactive in a modeless or movable modal dialog box when you deactivate it. The `HiliteControl` procedure dims inactive buttons, radio buttons, checkboxes, and pop-up menus to indicate to the user that clicking these items has no effect while the dialog box is in the background. When you activate a modeless or movable modal dialog box again, you should use `HiliteControl` to make the controls active again.

The application-defined `DoActivateGlobalChangesDialog` routine shown in Listing 6-34 illustrates how to use `HiliteControl` to make the Change button active when activating a modeless dialog box and how to make the Change and Stop buttons inactive when deactivating the dialog box.

Listing 6-34 Activating a modeless dialog box

```
PROCEDURE DoActivateGlobalChangesDialog (window: WindowPtr;
                                         event: EventRecord);
VAR
   activate:   Boolean;
   handled:    Boolean;
   item:       Integer;
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
BEGIN
   MyCheckEvent(event); {get a valid event record to pass to DialogSelect}
   activate := (BAnd(event.modifiers, activeFlag) <> 0);
   IF activate THEN     {activate the modeless dialog box}
   BEGIN
      {highlight editable text}
      SelectDialogItemText(window, kFindText, 0, 32767);
      {make the Change button active (make the Stop button active }
      { only during a change operation)}
      GetDialogItem(DialogPtr(window), kChange, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 0); {make Change active}
      {draw a bold outline around the newly activated Change button}
      MyDrawDefaultButtonOutline(DialogPtr(window), kChange);
   END
   ELSE     {dim the Change and Stop buttons for a deactivate dialog box}
   BEGIN
      GetDialogItem(DialogPtr(window), kChange, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);  {dim Change button}
      {draw a gray outline around the newly dimmed Change button}
      MyDrawDefaultButtonOutline(DialogPtr(window), kChange);
      GetDialogItem(DialogPtr(window), kStop, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);  {dim Stop button}
   END;
   {let Dialog Manager handle activate events}
   handled := DialogSelect(event, window, item);
   MyAdjustMenus;    {adjust the menus appropriately}
END;
```

The `DoActivateGlobalChangesDialog` routine uses `DialogSelect` to handle activate events in the modeless dialog box. In response to an activate event, `DialogSelect` handles the event and returns `FALSE`. The `DialogSelect` function sets the current graphics port to the modeless dialog box whenever the user makes it active.

Because `DialogSelect` expects three parameters, one of which must be an event record, `DoActivateGlobalChangesDialog` uses the application-defined routine `MyCheckEvent` to verify that the event is a valid event. If it's not, `MyCheckEvent` creates and returns a valid event record for an activate event.

Because `DialogSelect` doesn't call any draw procedures for items in response to activate events, `DoActivateGlobalChangesDialog` calls the application-defined draw routine `MyDrawDefaultButtonOutline` to draw either a black outline around the default button when activating the dialog box or a gray outline when deactivating it. The `MyDrawDefaultButtonOutline` routine is shown in Listing 6-17 on page 6-59.

Because users can switch out of your application when you display a movable modal dialog box, your application must handle activate events for it, too.

You can also use `DialogSelect` to handle update events. In response to an update event, `DialogSelect` calls the Window Manager procedure `BeginUpdate`, the Dialog Manager procedure `DrawDialog` to redraw the entire dialog box, and then the Window Manager procedure `EndUpdate`. However, a faster way to update the dialog box is to use the `UpdateDialog` procedure, which redraws only the update region of a dialog box. As shown in Listing 6-35, you should call `BeginUpdate` before using `UpdateDialog`, and then call `EndUpdate`.

Listing 6-35 Updating a modeless dialog box

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
      windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         ;  {update document windows here}
      kMyGlobalChangesModelessDialog:
         BEGIN
            BeginUpdate(window);
            UpdateDialog(window, window^.visRgn);
            EndUpdate(window);
         END;
      {handle cases for other window types here}
   END; {of CASE}
END;
```

Closing Dialog Boxes

When you no longer need a dialog box, you can dispose of it by using either the `CloseDialog` procedure if you allocated the memory for the dialog box or the `DisposeDialog` procedure if you did not. Or, you can merely make it invisible by using the Window Manager procedure `HideWindow`.

Generally, your application should not allocate memory for modal dialog boxes or movable modal dialog boxes, but it should allocate memory for modeless dialog boxes. Under these circumstances, your application should use `DisposeDialog` to dispose of either a fixed or movable modal dialog box when the user clicks the OK or Cancel button, and it should use `CloseDialog` to dispose of a modeless dialog box when the user clicks the close box or chooses Close from the File menu.

You do not close alert boxes; the Dialog Manager does that for you automatically by calling the `DisposeDialog` procedure after the user responds to the alert box by clicking any enabled button.

The `CloseDialog` procedure removes a dialog box from the screen and deletes it from the window list. It also releases the memory occupied by

- the data structures associated with the dialog box (such as its structure, content, and update regions)
- all the items in the dialog box (except for pictures and icons, which might be shared by other resources) and any data structures associated with them—for example, the region occupied by the scroll box of a scroll bar

The `CloseDialog` procedure does not dispose of the dialog record or the item list resource. Unlike `GetNewDialog`, `NewDialog` does not use a copy of the item list resource. So, if you create a dialog box with `NewDialog`, you may want to use `CloseDialog` to keep the item list resource in memory even if you didn't supply a pointer to the memory.

The `DisposeDialog` procedure calls `CloseDialog` and, in addition, releases the memory occupied by the dialog's item list resource and the dialog record. If you passed `NIL` as a parameter to `GetNewDialog` or `NewDialog` to let the Dialog Manager allocate memory in the heap, call `DisposeDialog` when you're done with a dialog box.

For modeless and movable modal dialog boxes, you might find it more efficient to hide the dialog box rather than remove its data structures. Listing 6-30 on page 6-94 uses the Window Manager routine `HideWindow` to hide the Global Changes modeless dialog box when the user clicks its close box. The next time the user invokes the Global Changes command, the dialog box is already available, in the same location and with the same text selected as when it was last used.

If you adjust the menus when you display a dialog box, be sure to return them to an appropriate state when you close the dialog box, as described in "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73.

Dialog Manager Reference

This section describes the data structure, routines, and resources that are specific to the Dialog Manager.

The "Data Structure" section shows the Pascal data structure for the dialog record, which the Dialog Manager creates and maintains. The "Dialog Manager Routines" section describes Dialog Manager routines for invoking alerts, creating and disposing of dialog boxes, manipulating items in alert and dialog boxes, and handling events in dialog boxes.

The "Application-Defined Routines" section describes routines that your application must supply when you need to create application-defined items in dialog boxes, to filter events that the Dialog Manager doesn't handle, and to define its own alert sounds.

The "Resources" section describes the dialog resource, the alert resource, the item list resource, the dialog color table resource, the alert color table resource, and the item color table resource. The summary sections that conclude this chapter include listings of the constants that define values for the item types in alert and dialog boxes, the OK and Cancel buttons in alert boxes, and the icons in note alert boxes, caution alert boxes, and stop alert boxes, along with the constants used by the `Gestalt` function for the Dialog Manager.

Data Structure

This section describes the dialog record. Your application doesn't need to create or use this record; rather, your application simply uses the appropriate Dialog Manager routines, creates any necessary resources, and then allows the Dialog Manager to create and use records of this data type as necessary. The dialog record is described here for completeness only.

The Dialog Record

To create an alert or a dialog box, you use a Dialog Manager routine—such as `Alert` or `GetNewDialog`—that incorporates information from your item list resource and from your alert resource or dialog resource into a data structure, called a *dialog record,* in memory. The Dialog Manager creates a dialog record, which is a data structure of type `DialogRecord`, whenever your application creates an alert or a dialog box. Your application generally should not create a dialog record or directly access its fields.

```
TYPE  DialogPtr      = WindowPtr;
      DialogPeek     = ^DialogRecord

      DialogRecord   =
      RECORD
         window:     WindowRecord;  {dialog window}
         items:      Handle;        {item list resource}
         textH:      TEHandle;      {current editable text item}
         editField:  Integer;       {editable text item number }
                                    { minus 1}
         editOpen:   Integer;       {used internally; reserved}
         aDefItem:   Integer;       {default button item number}
      END;
```

Field descriptions

`window`	The window record for the alert box or dialog box.
`items`	A handle to the item list resource for the alert or the dialog box.
`textH`	A handle to the current editable text item.
`editField`	The current editable text item.
`editOpen`	Used internally; reserved.
`aDefItem`	The item number of the default button.

Dialog Manager Routines

This section describes the routines for initializing the Dialog Manager, invoking alerts, creating and disposing of dialog boxes, manipulating items in alert and dialog boxes, and handling events in alert and dialog boxes.

Some Dialog Manager routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, `GetDialogItem` is also available as `GetDItem`. Table 6-1 provides a mapping between the previous name of a routine and its new equivalent name.

Table 6-1 Mapping between new and previous names of Dialog Manager routines

New name	Previous name
`DialogCopy`	`DlgCopy`
`DialogCut`	`DlgCut`
`DialogDelete`	`DlgDelete`
`DialogPaste`	`DlgPaste`
`DisposeDialog`	`DisposDialog`
`FindDialogItem`	`FindDItem`

Table 6-1 Mapping between new and previous names of Dialog Manager routines (continued)

New name	Previous name
`GetAlertStage`	`GetAlrtStage`
`GetDialogItem`	`GetDItem`
`GetDialogItemText`	`GetIText`
`HideDialogItem`	`HideDItem`
`NewColorDialog`	`NewCDialog`
`ResetAlertStage`	`ResetAlrtStage`
`SelectDialogItemText`	`SelIText`
`SetDialogFont`	`SetDAFont`
`SetDialogItem`	`SetDItem`
`SetDialogItemText`	`SetIText`
`ShowDialogItem`	`ShowDItem`
`UpdateDialog`	`UpdtDialog`

Initializing the Dialog Manager

Before using the Dialog Manager, you must initialize—in order—QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit. The first Dialog Manager routine to call is the `InitDialogs` procedure, which initializes the Dialog Manager.

At your application's request, the Dialog Manager uses the system alert sound for signaling the user during various alert stages. For alerts, if you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure (described on page 6-144) and call the `ErrorSound` procedure to make it the current sound procedure.

By default, the Dialog Manager displays static text and editable text items in the system font. To make it easier to localize your application for use with worldwide versions of system software, you should not change the font. However, if you determine that it is imperative for your application to display static text or editable text in a font other than the system font, you can use the `SetDialogFont` procedure.

InitDialogs

Use the `InitDialogs` procedure to initialize the Dialog Manager.

```
PROCEDURE InitDialogs (resumeProc:  ResumeProcPtr);
```

`resumeProc`
A pointer to a procedure used by the System Error Handler in case a fatal system error occurs on a system that predates MultiFinder. For System 7, your application should set this parameter to `NIL`.

DESCRIPTION

Before using the Dialog Manager, you must initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit, in that order. Then, to initialize the Dialog Manager, call `InitDialogs` once before all other Dialog Manager routines. The `InitDialogs` procedure does the following initialization:

- It saves the pointer passed in the `resumeProc` parameter. For System 7, your application should set the `resumeProc` parameter to `NIL`.
- It installs the system alert sound. To change the system alert sound, use the `ErrorSound` procedure.
- It passes empty strings to the `ParamText` procedure.

ErrorSound

To use your own alert sound instead of the system alert sound for signaling the user, use the `ErrorSound` procedure.

```
PROCEDURE ErrorSound (soundProc: SoundProcPtr);
```

`soundProc` A pointer to a procedure that generates the desired alert sounds.

DESCRIPTION

The Dialog Manager uses the system alert sound for signaling the user during various alert stages. The system alert sound, which is a sound resource stored in the System file, is played whenever system software or your application uses the Sound Manager procedure `SysBeep`. By changing the setting in the Sound control panel, the user can determine which sound is played. If you want to use sounds other than the system alert sound at various alert stages, write your own sound procedure and call the `ErrorSound` procedure to make it the current sound procedure.

SPECIAL CONSIDERATIONS

If you pass `NIL` in the `soundProc` parameter, the Dialog Manager neither plays sounds nor causes the menu bar to blink, and thus the user receives no signal.

SEE ALSO

See the description of `MyAlertSound` on page 6-144 for a discussion of how to write the sound procedure pointed to by the `soundProc` parameter. For examples of how to incorporate sound alerts into alert stages, see Listing 6-2 on page 6-21 and Listing 6-3 on page 6-22.

SetDialogFont

Although you generally should not change the font used in static and editable text items, you can do so with the `SetDialogFont` procedure. The `SetDialogFont` procedure is also available as the `SetDAFont` procedure.

```
PROCEDURE SetDialogFont (fontNum: Integer);
```

`fontNum`	A font ID number. Do not rely on font number constants. Instead, use the Font Manager function `GetFNum` to find the font number to pass in this parameter.

DESCRIPTION

For subsequently created dialog and alert boxes, `SetDialogFont` sets the font of the dialog or alert box's graphics port to the specified font. If you don't call this procedure, the system font is used. The `SetDialogFont` procedure does not affect titles of controls, which are always displayed in the system font.

SPECIAL CONSIDERATIONS

There are a number of caveats regarding the `SetDialogFont` procedure.

First, the Standard File Package does not always properly calculate the position of the standard file dialog box once this procedure has been called; for example, the standard file dialog box may be partially obscured by a menu bar. Second, be aware that this procedure affects all static text and editable text items in all of the alert and dialog boxes you subsequently display. Third, `SetDialogFont` does not change the font for control titles. Fourth, you can't use `SetDialogFont` to change the font size or font style. Finally, and most importantly, your application will be much easier to localize if you always use the system font in your alert and dialog boxes and never use `SetDialogFont`.

SEE ALSO

See the chapter "Font Manager" in *Inside Macintosh: Text* for information about the `GetFNum` function.

Creating Alerts

To create an alert—consisting of an alert sound, an alert box, or both—use one of these functions: `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert`. The first three functions display, respectively, the note, caution, and stop alert icons (see Figure 6-3, Figure 6-4, and Figure 6-5) in the upper-left corner of the alert box. The `Alert` function allows you to display your own icon or to have no icon at all in the upper-left corner of your alert box.

These functions take descriptive information about the alert from an alert resource that you provide. When you call one of these functions, you pass it the resource ID of the alert resource and a pointer to an event filter function. These functions create a dialog record, play an alert sound, and display an alert box according to the alert stages that you specify in the alert resource.

You should specify a pointer to an event filter function when you call the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. You can use the same event filter function in most or all of your alert and modal dialog boxes.

If you need to find out the current alert stage—for example, to ensure that your application deactivates the frontmost window only if an alert box is to be displayed at that stage—use the `GetAlertStage` function. To change the current alert stage, use the `ResetAlertStage` procedure.

Your application does not dispose of alert boxes; the Dialog Manager does that for you automatically.

Alert

To display an alert box (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), you can use the `Alert` function. This function does not display a default icon in the upper-left corner of the alert box; you can leave this area blank, or you can specify your own icon in the alert's item list resource, which in turn is specified in the alert resource.

```
FUNCTION Alert (alertID: Integer;
                filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure.

DESCRIPTION

The `Alert` function creates the alert defined in the specified alert resource. The function calls the current alert sound procedure and passes it the sound number specified in the alert resource for the current alert stage. If no alert box is to be drawn at this stage, `Alert` returns –1; otherwise, it uses the `NewDialog` function to create and display the alert box. The default system window colors are used unless your application provides an alert color table resource with the same resource ID as the alert resource.

The `Alert` function uses the `ModalDialog` procedure, which repeatedly gets and handles most events for you. The `ModalDialog` procedure, in turn, gets each event by calling the Event Manager function `GetNextEvent`. If the event is a mouse-down event outside the content region of the alert box, `ModalDialog` emits an error sound and gets the next event.

The `Alert` function continues calling `ModalDialog` until the user selects an enabled control (typically a button), at which time the `Alert` function removes the alert box from the screen and returns the item number of the selected control. Your application then responds as appropriate when the user clicks this item.

For events inside the alert box, `ModalDialog` passes the event to an event filter function before handling the event. The event filter function provides a secondary event-handling loop for events that `ModalDialog` doesn't handle. You specify a pointer to your event filter function in the `filterProc` parameter of the `Alert` function.

If you set the `filterProc` parameter to `NIL`, the Dialog Manager uses the standard event filter function, which behaves as follows:

- If the user presses the Return or Enter key, the event filter function returns `TRUE` and returns the item number for the default button.

However, your application should provide a simple event filter function that not only replicates this behavior but also

- returns `TRUE` and the item number for the Cancel button if the user presses Esc or Command-period
- updates your windows in response to update events (this also allows background windows to receive update events) and returns `FALSE`
- returns `FALSE` for all events that your event filter function doesn't handle

You can also use the event filter function to test for and respond to keyboard equivalents.

Unless the event filter function handles the event in its own way and returns `TRUE`, `ModalDialog` handles the event inside the alert box as follows:

- If the user presses the mouse button while the cursor is in a control, the Control Manager function `TrackControl` tracks the cursor. If the user releases the mouse button while the cursor is in an enabled control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the control's item number. (Generally, buttons should be the only controls you use in alert boxes.)
- If the user presses the mouse button while the cursor is in any enabled item other than a control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the item number. (Generally, button controls should be the only enabled items in alert boxes.)
- If user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` do nothing.

The `Alert` function uses the QuickDraw routine `SetPort` to make the alert box the current graphics port. It's not necessary for your application to call `SetPort` again before displaying alert boxes, because you can't draw into any other windows between the time you create an alert box and the time the Dialog Manager displays it.

SPECIAL CONSIDERATIONS

If you need to display an alert box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you will not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to an alert box that your application presents.

SEE ALSO

The `ModalDialog` procedure is described on page 6-135. See "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86 for a discussion of how to write an event filter function. See "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 for a discussion of alerts and alert stages. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about button titles and messages in alert boxes. Alert resources are described on page 6-150. Alert color table resources are described on page 6-157. The Dialog Manager uses the system alert sound as the error sound unless you change it by calling the `ErrorSound` procedure, described on page 6-104. See "Responding to Events in Alert Boxes" beginning on page 6-81 for a discussion of how to respond to events returned by the `Alert` function. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for information about the Notification Manager.

The `NoteAlert`, `CautionAlert`, and `StopAlert` functions are identical to the `Alert` function, except that `NoteAlert` (described on page 6-110), `CautionAlert` (described on page 6-111), and `StopAlert` (described next) display icons in the upper-left corners of alert boxes.

StopAlert

To display an alert box with a stop icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `StopAlert` function.

```
FUNCTION StopAlert (alertID: Integer;
                    filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `StopAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `StopAlert` draws the stop icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The stop icon has the following resource ID:

```
CONST stopIcon = 0;  {stop icon}
```

By default, the Dialog Manager uses the standard stop icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a stop alert to inform the user that a problem or situation is so serious that the action cannot be completed. Stop alerts typically have only a single button (OK), because all the user can do is acknowledge that the action cannot be completed.

SEE ALSO

Figure 6-5 on page 6-9 illustrates the stop icon in a typical stop alert. Except that it includes a stop icon in the alert box, `StopAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

NoteAlert

To display an alert box with a note icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `NoteAlert` function.

```
FUNCTION NoteAlert (alertID: Integer;
                    filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `NoteAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `NoteAlert` draws the note icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The note icon has the following resource ID:

```
CONST noteIcon = 1;  {note icon}
```

By default, the Dialog Manager uses the standard note icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a note alert to inform users of a minor mistake that won't have any disastrous consequences if left as is. Usually this type of alert simply offers information, and the user responds by clicking an OK button. Occasionally, a note alert may ask a simple question and provide a choice of responses.

SEE ALSO

Figure 6-3 on page 6-8 illustrates the note icon in a typical note alert. Except that it includes a note icon in the alert box, `NoteAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

CautionAlert

To display an alert box with a caution icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `CautionAlert` function.

```
FUNCTION CautionAlert (alertID: Integer;
                       filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `CautionAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `CautionAlert` draws the caution icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The caution icon has the following resource ID:

```
CONST cautionIcon = 2;  {caution icon}
```

By default, the Dialog Manager uses the standard caution icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a caution alert to alert the user of an operation that may have undesirable results if it's allowed to continue. Give the user the choice of continuing the action (by clicking an OK button) or stopping it (by clicking a Cancel button).

SEE ALSO

Figure 6-4 on page 6-9 illustrates the caution icon in a typical caution alert. Except that it includes a caution icon in the alert box, `CautionAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

GetAlertStage

To determine the stage of the last occurrence of an alert, use the `GetAlertStage` function. The `GetAlertStage` function is also available as the `GetAlrtStage` function.

```
FUNCTION GetAlertStage: Integer;
```

DESCRIPTION

The `GetAlertStage` function returns a number from 0 to 3 as the stage of the last occurrence of an alert. For example, you can use the `GetAlertStage` function to ensure that your application deactivates the active window only if an alert box is to be displayed at that stage.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ACount` contains this number. In addition, the global variable `ANumber` contains the resource ID of the alert resource of the last alert that occurred.

SEE ALSO

Listing 6-19 on page 6-66 illustrates how to use `GetAlertStage` to determine whether to deactivate a window for the current alert stage. Listing 6-2 on page 6-21 illustrates how to use an alert resource to specify different alert responses according to different alert stages.

ResetAlertStage

To reset the current alert stage to the first alert stage, use the `ResetAlertStage` procedure. The `ResetAlertStage` procedure is also available as the `ResetAlrtStage` procedure.

```
PROCEDURE ResetAlertStage;
```

DESCRIPTION

The `ResetAlertStage` procedure resets every alert to a first-stage alert.

SEE ALSO

Listing 6-2 on page 6-21 illustrates how to use an alert resource to specify different alert responses according to different alert stages.

Creating and Disposing of Dialog Boxes

To create a dialog box, you should generally use the `GetNewDialog` function, which takes information about the dialog from a dialog resource in a resource file. Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages. However, you can also use the `NewDialog` and `NewColorDialog` functions—for which you pass descriptive information in parameters—to create dialog boxes.

The `NewColorDialog` function is identical to the `NewDialog` function, except that `NewColorDialog` returns a pointer to a color graphics port.

When you no longer need a dialog box, use the `CloseDialog` procedure if you allocated the memory for the dialog record of the dialog box and use the `DisposeDialog` procedure if you did not. (To merely make the dialog box invisible to the user, you can use the Window Manager procedure `HideWindow`.)

GetNewDialog

To create a dialog box from a description in a dialog resource, use the `GetNewDialog` function.

```
FUNCTION GetNewDialog (dialogID: Integer; dStorage: Ptr;
                       behind: WindowPtr): DialogPtr;
```

`dialogID`	The resource ID of a dialog resource. If the dialog resource is missing, the Dialog Manager returns to your application without creating the dialog box.
`dStorage`	A pointer to the memory for the dialog record. If you set this parameter to `NIL` for modal dialog boxes and movable modal dialog boxes, the Dialog Manager automatically allocates memory for them in your application heap. For a modeless dialog box, however, you should allocate your own memory as you would for a window—otherwise, your heap could become fragmented.
`behind`	A pointer to the window behind which the dialog box is to be placed on the desktop. Always set this parameter to the window pointer `Pointer(-1)` to bring the dialog box in front of all other windows.

DESCRIPTION

The `GetNewDialog` function creates a dialog record from the information in the dialog resource and returns a pointer to it. You can use this pointer with Window Manager or QuickDraw routines to manipulate the dialog box. If the dialog resource specifies that the dialog box should be visible, the dialog box is displayed. If the dialog resource specifies that the dialog box should initially be invisible, use the Window Manager procedure `ShowWindow` to display the dialog box.

If you supply a dialog color table resource with the same resource ID as the dialog resource, GetNewDialog uses the NewColorDialog function and returns a pointer to a color graphics port. If no dialog color table resource is present, GetNewDialog uses NewDialog to return a pointer to a black-and-white graphics port, although system software draws the window frame using the system's default colors.

The dStorage and behind parameters of GetNewDialog have the same meaning as they do in the Window Manager function GetNewWindow. Always set the behind parameter to Pointer(-1) to bring the dialog box to the front.

The dialog resource contains the resource ID of the dialog box's item list resource. After calling the Resource Manager to read the item list resource into memory (if it's not already in memory), GetNewDialog makes a copy of the item list resource and uses that copy; thus you may have several dialog boxes with identical items.

If you provide a dialog color table resource, GetNewDialog copies it before passing it to the Window Manager routine SetWinColor unless the number-of-entries element of the dialog color table resource is set to –1, in which case the default window colors are used instead. The GetNewDialog function makes the copy so that the dialog color table resource can be purged without affecting the dialog box.

SPECIAL CONSIDERATIONS

The GetNewDialog function doesn't release the memory occupied by the resources. Therefore, your application should mark all resources used for a dialog box as purgeable.

If either the dialog resource or the item list resource can't be read, the function result is NIL; your application should test to ensure that NIL is not returned before performing any more operations with the dialog box or its items.

For modal dialog boxes, the Dialog Manager function ModalDialog traps all events. This prevents your event loop from receiving activate events for your windows. Thus, if one of your application's windows is active when you use GetNewDialog to create a modal dialog box, you must explicitly deactivate that window before displaying the modal dialog box.

If you ever need to display a dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you do not use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to the dialog box that your application presents.

The GetNewDialog function uses either NewDialog or NewColorDialog, each of which generates an update event for the entire window contents. Thus, with the exception of controls, items aren't drawn immediately. The Dialog Manager calls the Control Manager to draw controls, and the Control Manager draws them immediately. So the controls won't be drawn twice, the Dialog Manager calls the Window Manager procedure `ValidRect` for the enclosing rectangle of each control. If you find that there is too great a lag between the drawing of controls and the drawing of other items, try making the dialog box initially invisible and then calling the Window Manager procedure `ShowWindow` to show it.

SEE ALSO

See "Creating Dialog Boxes" beginning on page 6-23 and "Displaying Alert and Dialog Boxes" beginning on page 6-61 for discussions and examples of how to use `GetNewDialog`.

The `GetNewWindow` and `ShowWindow` procedures are described in the chapter "Window Manager" of this book. The Notification Manager is described in the chapter "Notification Manager" in *Inside Macintosh: Processes.*

"Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 discuss menu adjustment when your application displays dialog boxes. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about messages and control titles in dialog boxes.

NewColorDialog

To create a dialog box, you can use the `NewColorDialog` function, which returns a pointer to a color graphics port. Generally, you should instead use `GetNewDialog` to create a dialog box, because `GetNewDialog` takes information about the dialog box from a dialog resource in a resource file. (Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages.) The `NewColorDialog` function is also available as the `NewCDialog` function.

```
FUNCTION NewColorDialog (dStorage: Ptr; boundsRect: Rect;
                         title: Str255; visible: Boolean;
                         procID: Integer; behind: WindowPtr;
                         goAwayFlag: Boolean; refCon: LongInt;
                         items: Handle): CDialogPtr;
```

`dStorage` A pointer to the memory for the dialog record. If you set this parameter to `NIL` for modal dialog boxes and movable modal dialog boxes, the Dialog Manager allocates memory for them on your application heap. For a modeless dialog box, however, you should allocate your own memory as you would for a window—otherwise, your heap could become fragmented.

`boundsRect`	A rectangle, given in global coordinates, that determines the size and position of the dialog box; these coordinates specify the upper-left and lower-right corners of the dialog box.
`title`	A text string used for the title of a modeless or movable modal dialog box. You can specify an empty string (not `NIL`) for a title bar that contains no text.
`visible`	A flag that specifies whether the dialog box should be drawn on the screen immediately. If you set this parameter to `FALSE`, the dialog box is not drawn until your application uses the Window Manager procedure `ShowWindow` to display it.
`procID`	The window definition ID for the type of dialog box. Use the `dBoxProc` constant to specify modal dialog boxes, the `noGrowDocProc` constant to specify modeless dialog boxes, and the `movableDBoxProc` constant to specify movable modal dialog boxes.
`behind`	A pointer to the window behind which the dialog box is to be placed on the desktop. Always set this parameter to the window pointer `Pointer(-1)` to bring the dialog box in front of all other windows.
`goAwayFlag`	A flag to specify whether a modeless dialog box should have a close box in its title bar when the dialog box is active. If you set this parameter to `TRUE`, the dialog window has a close box in its title bar when the window is active; only modeless dialog boxes should have close boxes.
`refCon`	A value that the Dialog Manager uses to set the `refCon` field of the dialog box's window record. Your application may store any value here for any purpose. For example, your application can store a number that represents a dialog box type, or it can store a handle to a record that maintains state information about the dialog box. You can use the Window Manager procedure `SetWRefCon` at any time to change this value in the dialog record for a dialog box, and you can use the `GetWRefCon` function to determine its current value.
`items`	A handle to an item list resource for the dialog box. You can get the handle by calling the Resource Manager function `GetResource` to read the item list resource into memory. Use the Memory Manager procedure `HNoPurge` to make the handle unpurgeable while you use it or use the Operating System utility function `HandToHand` to make a copy of the handle and use the copy.

DESCRIPTION

The `NewColorDialog` function creates a dialog box as specified by its parameters and returns a pointer to a color graphics port for the new dialog box. The first eight parameters (`dStorage` through `refCon`) are passed to the Window Manager function `NewCWindow`, which creates the dialog box. You can use this pointer with Window Manager or QuickDraw routines to manipulate the dialog box.

The Dialog Manager uses the default window colors for the dialog box. By using the system's default colors, you ensure that your application's interface is consistent with that of the Finder and other applications. However, if you absolutely feel compelled to break from this consistency, you can use the Window Manager procedure `SetWinColor` to use your own dialog color table resource that specifies colors other than the default colors. Be aware, however, that nonstandard colors in your alert and dialog boxes may initially confuse your users.

The Window Manager creates an auxiliary window record for the color dialog box. You can access this record with the Window Manager function `GetAuxWin`. (The `dialogCItemhandle` field of the auxiliary window record points to the dialog box's item color table resource.) If the dialog box's content color isn't white, it's a good idea to call `NewColorDialog` with the `visible` flag set to `FALSE`. After the color table and color item list resource are installed, use the Window Manager procedure `ShowWindow` to display the dialog box if it's the frontmost window. If the dialog box is a modeless dialog box that is not in front, use the Window Manager procedure `ShowHide` to display it.

When specifying the size and position of the dialog box in the `boundsRect` parameter, you should generally try to center dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is more appropriate. Also ensure that the tops of dialog boxes (including the title bars of modeless and movable modal dialog boxes) lie below the menu bar when you position them on the main screen. You can use the Menu Manager function `GetMBarHeight` to determine the height of the menu bar.

SPECIAL CONSIDERATIONS

For modal dialog boxes, the Dialog Manager function `ModalDialog` traps all events. This prevents your event loop from receiving activate events for your windows. Thus, if one of your application's windows is active when you use `NewColorDialog` to create a modal dialog box, you must explicitly deactivate that window before displaying the modal dialog box.

If you ever need to display a dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you do not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to the dialog box that your application presents.

The `NewColorDialog` function generates an update event for the entire window contents. Thus, with the exception of controls, items aren't drawn immediately. The Dialog Manager calls the Control Manager to draw controls, and the Control Manager draws them immediately. So that the controls won't be drawn twice, the Dialog Manager calls the Window Manager procedure `ValidRect` for the enclosing rectangle of each control. If you find that there is too great a lag between the drawing of controls and the drawing of other items, try making the dialog box initially invisible and then calling the Window Manager procedure `ShowWindow` to show it.

SEE ALSO

Window Manager routines are described in the chapter "Window Manager" in this book. The Notification Manager is described in the chapter "Notification Manager" in *Inside Macintosh: Processes.* See *Inside Macintosh: Memory* for a description of `HNoPurge`. See *Inside Macintosh: Operating System Utilities* for a description of `HandToHand`.

"Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 discuss menu bar adjustment when your application displays dialog boxes. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about messages and control titles in dialog boxes. The `GetResource` function is described in the chapter "Resource Manager" of *Inside Macintosh: More Macintosh Toolbox.*

NewDialog

To create a dialog box, you can use the `NewDialog` function, which returns a pointer to a black-and-white graphics port (although system software draws the window frame of the dialog box using the system's default window colors). Generally, you should instead use `GetNewDialog` to create a dialog box; `GetNewDialog` takes information about the dialog from a dialog resource in a resource file. (Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages.)

The `NewDialog` function is identical to the `NewColorDialog` function, except that `NewDialog` returns a pointer to a black-and-white graphics port. See the discussion of `NewColorDialog` on page 6-115 for descriptions of the parameters that you also pass to `NewDialog`.

```
FUNCTION NewDialog (dStorage: Ptr; boundsRect: Rect;
                    title: Str255; visible: Boolean;
                    procID: Integer; behind: WindowPtr;
                    goAwayFlag: Boolean; refCon: LongInt;
                    items: Handle):  DialogPtr;
```

DESCRIPTION

The `NewDialog` function creates a dialog box as specified by its parameters and returns a pointer to a black-and-white graphics port for the new dialog box. The first eight parameters (`dStorage` through `refCon`) are passed to the Window Manager function `NewWindow`, which creates the dialog box.

When specifying the size and position of the dialog box in the `boundsRect` parameter, you should generally try to center dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is more appropriate. Also ensure that the tops of dialog boxes (including the title bars of modeless and movable modal dialog boxes) lie below the menu bar when you position them on the main screen. You can use the Menu Manager function `GetMBarHeight` to determine the height of the menu bar.

SEE ALSO

If you use a dialog color table resource to change the default window colors, use the `NewColorDialog` function, which returns a pointer to a color graphics port. See the description of `NewColorDialog` on page 6-115 for additional information common to both the `NewDialog` and `NewColorDialog` functions.

CloseDialog

To dismiss a dialog box for whose dialog record you allocated memory, use the `CloseDialog` procedure.

```
PROCEDURE CloseDialog (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `CloseDialog` procedure removes a dialog box from the screen and deletes it from the window list. The `CloseDialog` procedure releases the memory occupied by

- the data structures associated with the dialog box (such as its structure, content, and update regions)
- all the items in the dialog box (except for pictures and icons, which might be shared by other resources) and any data structures associated with them

Generally, you should provide memory for the dialog record of modeless dialog boxes when you create them. (You can let the Dialog Manager provide memory for modal and movable modal dialog boxes.) You should then use `CloseDialog` to close a modeless dialog box when the user clicks the close box or chooses Close from the File menu.

Because `CloseDialog` does not dispose of the dialog resource or the item list resource, it is important to make these resources purgeable. Unlike `GetNewDialog`, `NewColorDialog` does not use a copy of the item list resource. Thus, if you use `NewColorDialog` to create a dialog box, you may want to use `CloseDialog` to keep the item list resource in memory even if you didn't supply a pointer to the memory.

SEE ALSO

If you let the Dialog Manager allocate memory for the dialog box (by passing `NIL` in the `dStorage` parameter to the `GetNewDialog`, `NewColorDialog`, or `NewDialog` function), use the `DisposeDialog` procedure, described next, instead of `CloseDialog`.

DisposeDialog

To dismiss a dialog box for which the Dialog Manager supplies memory, use the `DisposeDialog` procedure. The `DisposeDialog` procedure is also available as the `DisposDialog` procedure.

```
PROCEDURE DisposeDialog (theDialog: DialogPtr);
```

`theDialog`   A pointer to a dialog record.

DESCRIPTION

The `DisposeDialog` procedure calls the `CloseDialog` procedure and, in addition, releases the memory occupied by the dialog box's item list resource and the dialog record. Call `DisposeDialog` when you're done with a dialog box if you pass `NIL` in the `dStorage` parameter to `GetNewDialog`, `NewColorDialog`, or `NewDialog`.

Generally, your application should not allocate memory for the dialog records of modal dialog boxes or movable modal dialog boxes. In these cases your application should use `DisposeDialog` when the user clicks the OK or Cancel button.

SEE ALSO

If you allocate memory for the dialog box (for example, by passing a pointer in the `dStorage` parameter to the `GetNewDialog`, `NewColorDialog`, or `NewDialog` function), use `CloseDialog`, described on page 6-119, instead of `DisposeDialog`.

Manipulating Items in Alert and Dialog Boxes

In many cases, you won't have to make any changes to alert or dialog boxes after you define them in the resource file. If you do need to make changes, use the Dialog Manager routines described in this section.

For most item manipulation, first call the `GetDialogItem` procedure to get the information about the item. You can then use other routines to manipulate that item. Use the `SetDialogItem` procedure if you use any of these other routines to change the item. You must also use `SetDialogItem` to install any of your own application-defined draw procedures. If you use `SetDialogItem`, make the dialog box initially invisible, change the item as appropriate, then make the dialog box visible by using the Window Manager procedure `ShowWindow`. (For information about manipulating text in an alert box or a dialog box, see "Handling Text in Alert and Dialog Boxes" beginning on page 6-129.)

You can dynamically add items to and remove items from a dialog box by using the `AppendDITL` and `ShortenDITL` procedures. These procedures are especially useful if you share a single item list resource among multiple dialog boxes, because you can then use `AppendDITL` or `ShortenDITL` to add or remove items as appropriate for individual dialog boxes. You typically make such dialog boxes invisible, use the `AppendDITL` and `ShortenDITL` procedures as appropriate, then make the dialog boxes visible by using the Window Manager procedure `ShowWindow`.

GetDialogItem

To get a handle to an item so that you can manipulate it (for example, to determine its current value, to change it, or to install a pointer to a draw procedure for an application-defined item), use the `GetDialogItem` procedure. The `GetDialogItem` procedure is also available as the `GetDItem` procedure.

```
PROCEDURE GetDialogItem (theDialog: DialogPtr; itemNo: Integer;
                         VAR itemType: Integer; VAR item: Handle;
                         VAR box: Rect);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an item in the dialog box's item list resource.
`itemType`	A value that represents the type of item requested in the `itemNo` parameter. You can use any of these constants to determine the value returned in this parameter:

```
CONST
 ctrlItem    = 4;     {add this constant to the next }
                      { four constants}
 btnCtrl     = 0;     {standard button control}
 chkCtrl     = 1;     {standard checkbox control}
 radCtrl     = 2;     {standard radio button}
 resCtrl     = 3;     {control defined in a 'CNTL'}
 helpItem    = 1;     {help balloons}
 statText    = 8;     {static text}
 editText    = 16;    {editable text}
 iconItem    = 32;    {icon}
 picItem     = 64;    {QuickDraw picture}
 userItem    = 0;     {application-defined item}
 itemDisable = 128;   {add to any of the above to }
                      { disable it}
```

`item`	For an application-defined draw procedure, a pointer to the draw procedure (coerced to a handle), returned for the item specified in the `itemNo` parameter; for all other item types, a handle to the item.
`box`	The display rectangle (described in coordinates local to the dialog box), returned for the item specified in the `itemNo` parameter.

DESCRIPTION

The `GetDialogItem` procedure returns in its parameters the following information about the item numbered `itemNo` in the item list resource of the specified dialog box: in the `itemType` parameter, the item type; in the `item` parameter, a handle to the item (or, for application-defined draw procedures, the procedure pointer); and in the `box` parameter, the display rectangle for the item.

For most item manipulation, first use the `GetDialogItem` procedure to get the information about the item. You can then use other routines, such as `GetDialogItemText` and `SetDialogItem`, to determine and change the value of that item.

SEE ALSO

Listing 6-12 on page 6-49 illustrates the use of `GetDialogItem` in conjunction with `GetDialogItemText` to retrieve the text entered by a user in an editable text item. Listing 6-16 on page 6-58 illustrates the use of `GetDialogItem` in conjunction with `SetDialogItem` to install the draw procedure for an application-defined item into a dialog box. Listing 6-26 on page 6-83 illustrates the use of `GetDialogItem` to determine the current value of a checkbox in a dialog box.

SetDialogItem

After using the `GetDialogItem` procedure to get a handle to an item from a dialog box, use the `SetDialogItem` procedure to set or change the item. The `SetDialogItem` procedure is also available as the `SetDItem` procedure.

```
PROCEDURE SetDialogItem (theDialog: DialogPtr; itemNo: Integer;
                         itemType: Integer; item: Handle;
                         box: Rect);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an item in the dialog box's item list resource.
`itemType`	A value that represents the type of item in the `itemNo` parameter. To specify the value for this parameter, you can use any of the constants listed on page 6-121 for the `itemType` parameter of the `GetDialogItem` procedure.

`item`	For an application-defined item, a pointer to the draw procedure (coerced to a handle) for the item specified in the `itemNo` parameter; for all other item types, a handle to the item.
`box`	The display rectangle (described in coordinates local to the dialog box) for the item specified in the `itemNo` parameter.

DESCRIPTION

The `SetDialogItem` procedure sets the item specified by the `itemNo` parameter for the specified dialog box. This procedure installs the item without drawing it; typically you create an invisible dialog box, use `SetDialogItem`, then use the Window Manager procedure `ShowWindow` to draw the dialog box and its items.

SEE ALSO

Listing 6-16 on page 6-58 illustrates how to use `SetDialogItem` to install an application-defined draw procedure. The `ShowWindow` procedure is described in the chapter “Window Manager” of this book.

HideDialogItem

Although you should rarely need to do so, you can make an item in a dialog box invisible by using the `HideDialogItem` procedure. The `HideDialogItem` procedure is also available as the `HideDItem` procedure.

```
PROCEDURE HideDialogItem (theDialog: DialogPtr; itemNo: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an item in the dialog box’s item list resource.

DESCRIPTION

The `HideDialogItem` procedure hides the item specified by `itemNo` by giving it a display rectangle that’s off the screen. Specifically, if the left coordinate of the item’s display rectangle is less than 8192 (hexadecimal $2000), `HideDialogItem` adds 16,384 (hexadecimal $4000) to both the left and right coordinates of the rectangle. If the item is already hidden (that is, if the left coordinate is greater than 8192), `HideDialogItem` does nothing. To redisplay an item that’s been hidden by `HideDialogItem`, you can use the `ShowDialogItem` procedure.

SPECIAL CONSIDERATIONS

If your application needs to display a number of dialog boxes that are similar except for one or two items, it's generally easier to modify the common elements using the `AppendDITL` and `ShortenDITL` procedures than to use the `HideDialogItem` and `ShowDialogItem` procedures.

The rectangle for a static text item must always be at least as wide as the first character of the text.

You generally shouldn't use `HideDialogItem` to make an editable text item invisible, because as the user presses the Tab key, the Dialog Manager attempts to move the cursor to the hidden editable text item, where the user's subsequent keystrokes will be placed.

ShowDialogItem

To redisplay an item that has been hidden by the `HideDialogItem` procedure, use the `ShowDialogItem` procedure. The `ShowDialogItem` procedure is also available as the `ShowDItem` procedure.

```
PROCEDURE ShowDialogItem (theDialog: DialogPtr; itemNo: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an item in the dialog box's item list resource.

DESCRIPTION

The `ShowDialogItem` procedure redisplays the item specified in `itemNo` by restoring the display rectangle the item had prior to the `HideDialogItem` call. Specifically, if the left coordinate of the item's display rectangle is greater than 8192, `ShowDialogItem` subtracts 16,384 from both the left and right coordinates of the rectangle. If the item is already visible (that is, if the left coordinate is less than 8192), `ShowDialogItem` does nothing.

The `ShowDialogItem` procedure adds the rectangle that contained the item to the update region so that it will be drawn. Note that if the item is a control you define in a control (`'CNTL'`) resource, the rectangle added to the update region is the rectangle defined in the control resource, not the display rectangle defined in the item list resource. If the item is an editable text item, `ShowDialogItem` activates it by calling the TextEdit procedure `TEActivate`.

FindDialogItem

To determine the item number of an item at a particular location in a dialog box, use the `FindDialogItem` function. The `FindDialogItem` function is also available as the `FindDItem` function.

```
FUNCTION FindDialogItem (theDialog: DialogPtr; thePt: Point)
                         : Integer;
```

`theDialog`	A pointer to a dialog record.
`thePt`	A point, specified in coordinates local to the dialog box.

DESCRIPTION

If the point specified in the parameter `thePt` lies within an item, `FindDialogItem` returns a number corresponding to the position of that item in the dialog box's item list resource. If the point doesn't lie within the item's rectangle, `FindDialogItem` returns –1. If items overlap, `FindDialogItem` returns the item number of the first item, in the item list resource, containing the point.

This function is useful for changing the cursor when it's over a particular item.

The `FindDialogItem` function returns 0 for the first item in the item list resource, 1 for the second, and so on. To get the proper item number before calling the `GetDialogItem` or `SetDialogItem` procedure, add 1 to `FindDialogItem`'s function result, as shown here:

```
theItem := FindDialogItem(theDialog, thePoint) + 1;
```

Note that `FindDialogItem` returns the item number of disabled items as well as enabled items.

AppendDITL

To add items to an existing dialog box while your application is running, use the `AppendDITL` procedure.

```
PROCEDURE AppendDITL (theDialog: DialogPtr; theDITL: Handle;
                      theMethod: DITLMethod);
```

`theDialog`	A pointer to a dialog record. This is the dialog record to which you will add the item list resource specified in the parameter `theDITL`.
`theDITL`	A handle to the item list resource whose items you want to append to the dialog box.

`theMethod`	The manner in which you want the new items to be displayed in the existing dialog box. You can pass a negative value to offset the appended items from a particular item in the existing dialog box. You can also pass any of these constants:

```
CONST
overlayDITL =        0;    {overlay existing items}
appendDITLRight =    1;    {append at right}
appendDITLBottom =   2;    {append at bottom}
```

DESCRIPTION

The `AppendDITL` procedure adds the items in the item list resource specified in the parameter `theDITL` to the items of a dialog box. This procedure is especially useful if several dialog boxes share a single item list resource, because you can use `AppendDITL` to add items that are appropriate for individual dialog boxes. Your application can use the Resource Manager function `GetResource` to get a handle to the item list resource whose items you wish to add.

In the parameter `theMethod`, you specify how to append the new items, as follows:

- If you use the `overlayDITL` constant, `AppendDITL` superimposes the appended items over the dialog box. That is, `AppendDITL` interprets the coordinates of the display rectangles for the appended items (as specified in their item list resource) as local coordinates within the dialog box.
- If you use the `appendDITLRight` constant, `AppendDITL` appends the items to the right of the dialog box by positioning the display rectangles of the appended items relative to the upper-right coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.
- If you use the `appendDITLBottom` constant, `AppendDITL` appends the items to the bottom of the dialog box by positioning the display rectangles of the appended items relative to the lower-left coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.
- You can also append a list of items relative to an existing item by passing a negative number in the parameter `theMethod`. The absolute value of this number is interpreted as the item in the dialog box relative to which the new items are to be positioned. For example, if you pass –2, the display rectangles of the appended items are offset relative to the upper-left corner of item number 2 in the dialog box.

You typically create an invisible dialog box, call the `AppendDITL` procedure, then make the dialog box visible by using the Window Manager procedure `ShowWindow`.

SPECIAL CONSIDERATIONS

The `AppendDITL` procedure modifies the contents of the dialog box (for instance, by enlarging it). To use an unmodified version of the dialog box at a later time, your application should use the Resource Manager procedure `ReleaseResource` to release the memory occupied by the appended item list resource. Otherwise, if your application calls `AppendDITL` to add items to that dialog box again, the dialog box remains modified by your previous call—for example, it will still be longer at the bottom if you previously used the `appendDITLBottom` constant.

The `AppendDITL` procedure is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `AppendDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `AppendDITL` is available.

SEE ALSO

Listing 6-13 on page 6-54 and Listing 6-14 on page 6-55 illustrate a typical use of `AppendDITL`. Figure 6-29 on page 6-52 shows the result of using the `overlayDITL` constant, Figure 6-30 on page 6-52 shows the result of using the `appendDITLRight` constant, Figure 6-31 on page 6-53 shows the result of using the `appendDITLBottom` constant, and Figure 6-32 on page 6-53 shows the result of using a negative number in the parameter `theMethod`.

The chapter “Resource Manager” in *Inside Macintosh: More Macintosh Toolbox* describes the `GetResource` and `ReleaseResource` routines. The `Gestalt` function is described in the chapter “Gestalt Manager” of *Inside Macintosh: Operating System Utilities*. See the chapter “Window Manager” in this book for information about `ShowWindow`.

ShortenDITL

To remove items from an existing dialog box while your application is running, use the `ShortenDITL` procedure.

```
PROCEDURE ShortenDITL (theDialog: DialogPtr;
                       numberItems: Integer);
```

`theDialog` A pointer to a dialog record.

`numberItems`
The number of items to remove (starting from the last item in the item list resource).

DESCRIPTION

The `ShortenDITL` procedure removes the specified number of items from the dialog box. This procedure is especially useful if several dialog boxes share a single item list resource, because you can use `ShortenDITL` to remove items as necessary for individual dialog boxes.

You typically create an invisible dialog box, call the `ShortenDITL` procedure, then make the dialog box visible by using the Window Manager procedure `ShowWindow`. Note that `ShortenDITL` does not automatically resize the dialog box; you can use the Window Manager procedure `SizeWindow` if you need to resize the dialog box.

SPECIAL CONSIDERATIONS

The `ShortenDITL` procedure is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `ShortenDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `ShortenDITL` is available.

SEE ALSO

You can use the `CountDITL` function, described next, to determine the number of items in the dialog box's item list resource. See the chapter "Window Manager" in this book for information on the `ShowWindow` and `SizeWindow` procedures. The `Gestalt` function is described in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities.*

CountDITL

You can determine the number of items in a dialog box by using the `CountDITL` function.

```
FUNCTION CountDITL (theDialog: DialogPtr): Integer;
```

`theDialog`	A pointer to a dialog record.

DESCRIPTION

The `CountDITL` function returns the number of current items in a dialog box. You typically use `CountDITL` in conjunction with `ShortenDITL` to remove items from a dialog box.

SPECIAL CONSIDERATIONS

The `CountDITL` function is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `CountDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `CountDITL` is available.

SEE ALSO

The `Gestalt` function is described in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities.*

Handling Text in Alert and Dialog Boxes

The Dialog Manager provides several routines for manipulating text. You can use the `ParamText` procedure to supply text strings, such as document titles, dynamically in the static text items of alert and dialog boxes. The `GetDialogItemText` and `SetDialogItemText` procedures are useful for determining and changing text in both static text and editable text items. You can use the `SelectDialogItemText` procedure to select and highlight text in an editable text item.

When a dialog box containing an editable text item is active, use the `DialogCut` procedure to handle the Cut editing command, the `DialogCopy` procedure to handle the Copy command, the `DialogPaste` procedure to handle the Paste command, and the `DialogDelete` procedure to handle the Clear command.

Once you determine that an event occurs in a modeless or movable modal dialog box, you can use the `DialogSelect` function, which is described on page 6-139, to handle key-down events in editable text items automatically. The `ModalDialog` procedure uses `DialogSelect` to handle key-down events in the editable text items of modal dialog boxes.

ParamText

To substitute text strings in the static text items of your alert or dialog boxes while your application is running, use the `ParamText` procedure.

```
PROCEDURE ParamText (param0: Str255; param1: Str255;
                     param2: Str255; param3: Str255);
```

`param0`	A text string to substitute for the special string `^0` in the static text items of all subsequently created alert and dialog boxes.
`param1`	A text string to substitute for the special string `^1` in the static text items of all subsequently created alert and dialog boxes.
`param2`	A text string to substitute for the special string `^2` in the static text items of all subsequently created alert and dialog boxes.
`param3`	A text string to substitute for the special string `^3` in the static text items of all subsequently created alert and dialog boxes.

DESCRIPTION

The `ParamText` procedure replaces the special strings `^0` through `^3` in the static text items of all subsequently created alert and dialog boxes with the text strings you pass as parameters. Pass empty strings (not `NIL`) for parameters not used.

SPECIAL CONSIDERATIONS

The strings used in `ParamText` are stored in the low-memory global variable `DAStrings`, which specifies a set of string handles used by the Dialog Manager.

If the user launches a desk accessory in your application's partition and the desk accessory calls `ParamText`, it may change the text in your application's dialog box.

You should be very careful about using `ParamText` in modeless dialog boxes. If a modeless dialog box using `ParamText` is onscreen and you display another dialog box or alert box that also uses `ParamText`, both boxes will be affected by the latest call to `ParamText`.

The strings you pass in the parameters to `ParamText` cannot contain the special strings `^0` through `^3`, or else the procedure will enter an endless loop of substitutions in versions of system software earlier than 7.1.

Note that you should try to store text strings in resource files to facilitate translation into other languages; therefore, `ParamText` is best used for supplying text strings, such as document names, that the user specifies. To avoid problems with grammar and sentence structure when you localize your application, you should use `ParamText` to supply only one text string per screen message.

SEE ALSO

Listing 6-9 on page 6-47 and Listing 6-10 on page 6-48 show an example of how you can use `ParamText` to supply the title of the user's current document to your alert and dialog boxes. If you need to supply a default text string to an editable text item while your application is running, use `SetDialogItemText`. The `SetDialogItemText` procedure also allows you to set or change the entire text string for a static text item.

GetDialogItemText

After using the `GetDialogItem` procedure to get a handle to an editable text item or a static text item in a dialog box, you can use the `GetDialogItemText` procedure to get the text string contained in that item. The `GetDialogItemText` procedure is also available as the `GetIText` procedure.

```
PROCEDURE GetDialogItemText (item: Handle; VAR text: Str255);
```

`item`	A handle to an editable text item or a static text item in a dialog box.
`text`	The text contained within the item.

DESCRIPTION

The `GetDialogItemText` procedure returns, in the `text` parameter, the text of the given editable text or static text item.

SPECIAL CONSIDERATIONS

If the user types more than 255 characters in an editable text item, `GetDialogItemText` returns only the first 255.

SEE ALSO

Listing 6-12 on page 6-49 illustrates how to use `GetDialogItemText` to retrieve the text that a user types into an editable text item.

SetDialogItemText

After using the `GetDialogItem` procedure to get a handle to an editable text item or a static text item in a dialog box, you can use the `SetDialogItemText` procedure to display a particular text string in that item. The `SetDialogItemText` procedure is also available as the `SetIText` procedure.

```
PROCEDURE SetDialogItemText (item: Handle; text: Str255);
```

`item`	A handle to an editable text item or a static text item in a dialog box.
`text`	The text to display in the item.

DESCRIPTION

The `SetDialogItemText` procedure places the specified text in the specified item and draws the item. This procedure is useful for supplying a default text string—such as a document name—for an editable text item while your application is running.

SPECIAL CONSIDERATIONS

All strings should be stored in resource files to ease translation into other languages.

SEE ALSO

For static text items, the `ParamText` procedure, described on page 6-129, is useful when you need to determine and provide only a portion of a text string while your application is running.

SelectDialogItemText

To select and highlight text contained in an editable text item, use the `SelectDialogItemText` procedure. The `SelectDialogItemText` procedure is also available as the `SelIText` procedure.

```
PROCEDURE SelectDialogItemText (theDialog: DialogPtr;
                                itemNo: Integer;
                                strtSel: Integer;
                                endSel: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an editable text item in the dialog box's item list resource.
`strtSel`	A number representing the position of the first character to begin selecting.
`endSel`	A number representing one position past the last character to be selected.

DESCRIPTION

If the item in the `itemNo` parameter is an editable text item that contains text, the `SelectDialogItemText` procedure sets the text selection range to extend from the character position specified in the `strtSel` parameter up to but not including the character position specified in the `endSel` parameter. The selection range is highlighted unless `strtSel` equals `endSel`, in which case a blinking vertical bar is displayed to indicate an insertion point at that position. If the editable text item doesn't contain text, `SelectDialogItemText` displays the insertion point.

You can select the entire text by specifying the number 0 in the `strtSel` parameter and the number 32767 in the `endSel` parameter.

For example, if the user makes an unacceptable entry in the editable text item, your application can display an alert box reporting the problem and then use `SelectDialogItemText` to select the entire text so it can be replaced by a new entry. Without this procedure, the user would have to select the item before making the new entry.

SEE ALSO

For details about text selection range and character position, see the chapter "TextEdit" in *Inside Macintosh: Text.*

DialogCut

When a dialog box containing an editable text item is active, use the `DialogCut` procedure to handle the Cut editing command. The `DialogCut` procedure is also available as the `DlgCut` procedure.

```
PROCEDURE DialogCut (theDialog: DialogPtr);
```

`theDialog`	A pointer to a dialog record.

DESCRIPTION

The `DialogCut` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TECut` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TECut` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogCopy

When a dialog box containing an editable text item is active, use the `DialogCopy` procedure to handle the Copy editing command. The `DialogCopy` procedure is also available as the `DlgCopy` procedure.

```
PROCEDURE DialogCopy (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogCopy` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TECopy` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TECopy` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogPaste

When a dialog box containing an editable text item is active, use the `DialogPaste` procedure to handle the Paste editing command. The `DialogPaste` procedure is also available as the `DlgPaste` procedure.

```
PROCEDURE DialogPaste (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogPaste` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TEPaste` to the selected editable text item. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see “Adjusting Menus for Modal Dialog Boxes” beginning on page 6-68 and “Adjusting Menus for Movable Modal and Modeless Dialog Boxes” on page 6-73. The `TEPaste` procedure is described in the chapter “TextEdit” in *Inside Macintosh: Text.*

DialogDelete

When a dialog box containing an editable text item is active, use the `DialogDelete` procedure to handle the Clear editing command. The `DialogDelete` procedure is also available as the `DlgDelete` procedure.

```
PROCEDURE DialogDelete (theDialog: DialogPtr);
```

`theDialog`	A pointer to a dialog record.

DESCRIPTION

The `DialogDelete` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TEDelete` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see “Adjusting Menus for Modal Dialog Boxes” beginning on page 6-68 and “Adjusting Menus for Movable Modal and Modeless Dialog Boxes” on page 6-73. The `TEDelete` procedure is described in the chapter “TextEdit” in *Inside Macintosh: Text*.

Handling Events in Dialog Boxes

Handling events in an alert box is very simple: after you invoke an alert box, the Dialog Manager handles most events for you by automatically calling the `ModalDialog` procedure. To handle events in a modal dialog box, your application must explicitly call the `ModalDialog` procedure after displaying the dialog box. In either case, when an enabled item is clicked, the Dialog Manager returns the item number. You'll then do whatever is appropriate in response to that click. For both alert and modal dialog boxes, you should also provide a simple event filter function that allows other windows to respond to update events and that allows your alert or dialog box to respond to a few key-down events for keys such as Return, Enter, and Esc.

You can use your normal event-handling code to determine whether an event occurs in a modeless or movable modal dialog box, or you can use the `IsDialogEvent` function to learn whether they need to be handled as part of a dialog box. Once you determine that an event occurs in a modeless or movable modal dialog box, you can use the `DialogSelect` function to handle key-down events in editable text items automatically, to handle update and activate events automatically, and to report the enabled items clicked by the user. You then respond as appropriate to clicks in your active items. Or you can use Control Manager, TextEdit, and Window Manager routines (such as `FindWindow`, `BeginUpdate`, `EndUpdate`, `FindControl`, `TrackControl`, and `TEClick`) to handle these events without the aid of the Dialog Manager.

ModalDialog

To handle events when you display a modal dialog box, use the `ModalDialog` procedure.

```
PROCEDURE ModalDialog (filterProc: ModalFilterProcPtr;
                       VAR itemHit: Integer);
```

`filterProc`	A pointer to an event filter function.
`itemHit`	A number representing the position of the selected item in the item list resource for the active modal dialog box.

DESCRIPTION

Call the `ModalDialog` procedure immediately after displaying a modal dialog box. The `ModalDialog` procedure assumes that a modal dialog box is displayed as the current port, and `ModalDialog` repeatedly handles events inside that port until an event involving an enabled dialog box item—such as a click in a radio button, for example—occurs. If the event is a mouse-down event outside the content region of the dialog box, `ModalDialog` emits the system alert sound and gets the next event. After receiving an event involving an enabled item, `ModalDialog` returns its item number in the `itemHit` parameter. Your application should then do whatever is appropriate in response to an event in that item. Your application should continue calling `ModalDialog` until the user selects the OK or Cancel button.

For events inside the dialog box, `ModalDialog` passes the event to the event filter function pointed to in the `filterProc` parameter before handling the event. When the event filter returns `FALSE`, `ModalDialog` handles the event. If the event filter function handles the event, the event filter function returns `TRUE`, and `ModalDialog` performs no more event handling.

If you set the `filterProc` parameter to `NIL`, the standard event filter function is executed. The standard event filter function returns `TRUE` and causes `ModalDialog` to return item number 1, which is the number of the default button, when the user presses the Return key or the Enter key. However, your application should provide a simple event filter function that

- returns `TRUE` and the item number for the default button if the user presses the Return or Enter key
- returns `TRUE` and the item number for the Cancel button if the user presses the Esc key or the Command-period key combination
- updates your windows in response to update events (this allows background applications to receive update events) and return `FALSE`
- returns `FALSE` for all events that your event filter function doesn't handle

You can use the same event filter function in most or all of your alert and modal dialog boxes.

You can also use the event filter function specified in the `filterProc` parameter to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor within an application-defined item.

To handle events, `ModalDialog` calls the `IsDialogEvent` function. If the result of `IsDialogEvent` is `TRUE`, then `ModalDialog` calls the `DialogSelect` function to handle the event. Unless the event filter function returns `TRUE`, `ModalDialog` handles the event as follows:

- In response to an activate or update event for the dialog box, `ModalDialog` activates or updates its window.
- If the user presses the mouse button while the cursor is in an editable text item, `ModalDialog` responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If a key-down event occurs and there's an editable text item, `ModalDialog` uses TextEdit to handle text entry and editing automatically. If the editable text item is enabled, `ModalDialog` returns its item number after it receives either the mouse-down or key-down event. Normally, editable text items are disabled, and you use the `GetDialogItemText` procedure to read the information in the items only after the user clicks the OK button.
- If the user presses the mouse button while the cursor is in a control, `ModalDialog` calls the Control Manager function `TrackControl`. If the user releases the mouse button while the cursor is in an enabled control, `ModalDialog` returns the control's item number. Your application should respond appropriately—for example, by performing a command after the user clicks the OK button.

- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `ModalDialog` returns the item's number, and your application should respond appropriately. Generally, only controls should be enabled. If your application creates a control more complex than a button, radio button, or checkbox, your application must handle events inside that item with your event filter function.
- If the user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `ModalDialog` does nothing.

SPECIAL CONSIDERATIONS

Do not use `ModalDialog` for movable modal dialog boxes (that is, those created with the `movableDBoxProc` window definition ID) or for modeless dialog boxes (that is, those created with the `noGrowDocProc` window definition ID). If you want the Dialog Manager to assist you in handling events for movable modal and modeless dialog boxes, use the `IsDialogEvent` and `DialogSelect` functions instead.

The `ModalDialog` procedure calls the Event Manager function `GetNextEvent` with a mask that excludes disk-inserted events. To receive disk-inserted events, your event filter function can call the Event Manager procedure `SetSystemEventMask`.

When `ModalDialog` calls `TrackControl`, it does not allow you to specify the action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control, a picture, or an application-defined item that draws a control- like object in your dialog box. You must then provide an event filter function that appropriately handles events in that item.

SEE ALSO

Listing 6-26 on page 6-83 illustrates the use of `ModalDialog`. "Responding to Events in Editable Text Items" beginning on page 6-79 describes how `ModalDialog` uses TextEdit to handle text entry and editing in editable text items. The `IsDialogEvent` and `DialogSelect` functions (which your application may use instead of `ModalDialog` for modeless and movable modal dialog boxes) are described on page 6-138 and page 6-139, respectively. See the description of `MyEventFilter` on page 6-145 for information about the event filter function your application should specify in the `filterProc` parameter.

The `GetNextEvent` and `SetSystemEventMask` routines are described in the chapter "Event Manager" in this book. See that chapter as well for a discussion of disk-inserted events. See "Responding to Events in Controls" on page 6-78 for a description of how your application should respond to events inside of controls; the `TrackControl` function is fully described in the chapter "Control Manager" in this book. Also see that chapter for information about creating your own nonstandard controls. TextEdit is described in the chapter "TextEdit" of *Inside Macintosh: Text.*

IsDialogEvent

To determine whether a modeless dialog box or a movable modal dialog box is active when an event occurs, you can use the `IsDialogEvent` function.

```
FUNCTION IsDialogEvent (theEvent: EventRecord): Boolean;
```

`theEvent`	An event record returned by an Event Manager function such as `WaitNextEvent`.

DESCRIPTION

If any event, including a null event, occurs when your dialog box is active, `IsDialogEvent` returns `TRUE`; otherwise, it returns `FALSE`. When `IsDialogEvent` returns `FALSE`, pass the event to the rest of your event-handling code. When `IsDialogEvent` returns `TRUE`, pass the event to `DialogSelect` after testing for the events that `DialogSelect` does not handle.

A dialog record includes a window record. When you use the `GetNewDialog`, `NewDialog`, or `NewColorDialog` function to create a dialog box, the Dialog Manager sets the `windowKind` field in the window record to `dialogKind`. To determine whether the active window is a dialog box, `IsDialogEvent` checks the `windowKind` field.

Before passing the event to `DialogSelect`, you should perform the following tests whenever `IsDialogEvent` returns `TRUE`:

- Check whether the event is a key-down event for the Return, Enter, Esc, or Command-period keystrokes. When the user presses the Return or Enter key, your application should respond as if the user had clicked the default button; when the user presses Esc or Command-period, your application should respond as if the user had clicked the Cancel button. Use the Control Manager procedure `HiliteControl` to highlight the applicable button for 8 ticks.
- At this point, you may also want to check for and respond to any special events that you do not wish to pass to `DialogSelect` or that require special processing before you pass them to `DialogSelect`. You would need to do this, for example, if the dialog box needs to respond to disk-inserted events.
- Check whether the event is an update event for a window other than the dialog box and, if it is, update your window.
- For complex items that you create, such as pictures or application-defined items that emulate complex controls, test for and respond to mouse events inside those items as appropriate. When `DialogSelect` calls `TrackControl`, it does not allow you to specify the action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control or a picture or an application-defined item that draws a control-like object in your dialog box. You must then test for and respond to those events yourself.

If your application uses `IsDialogEvent` to help handle events when you display a movable modal dialog box, perform the following additional tests before passing events to `DialogSelect`:

- Test for mouse-down events in the title bar of the movable modal dialog box and respond by dragging the dialog box accordingly.
- Test for and respond to mouse-down events in the Apple menu and, if the movable modal dialog box includes editable text items, in the Edit menu. (You should disable all other menus when you display a movable modal dialog box.)
- Play the system alert sound for every other mouse-down event outside the movable modal dialog box.

SPECIAL CONSIDERATIONS

Both `IsDialogEvent` and `DialogSelect` are unreliable when running in versions of system software earlier than System 7. You shouldn't use these routines if you expect your application to run in earlier versions of system software.

SEE ALSO

The `WaitNextEvent` function is described in the chapter "Event Manager" in this book. See *Inside Macintosh: Sound* for a description of the `SysBeep` procedure. The `FrontWindow` function is described in the chapter "Window Manager" in this book.

DialogSelect

After determining that an event related to an active modeless dialog box or an active movable modal dialog box has occurred, you can use the `DialogSelect` function to handle most of the events inside the dialog box.

```
FUNCTION DialogSelect (theEvent: EventRecord;
                       VAR theDialog: DialogPtr;
                       VAR itemHit: Integer): Boolean;
```

`theEvent`	An event record returned by an Event Manager function such as `WaitNextEvent`.
`theDialog`	A pointer to a dialog record for the dialog box where the event occurred.
`itemHit`	A number corresponding to the position of an item within the item list resource of the active dialog box.

DESCRIPTION

The `DialogSelect` function handles most of the events relating to a dialog box. If the event is an activate or update event for a dialog box, `DialogSelect` activates or updates it and returns `FALSE`. If the event involves an enabled item, `DialogSelect` returns a function result of TRUE. In its itemHit parameter, it returns the item number of the item selected by the user. In the parameter theDialog, it returns a pointer to the dialog record for the dialog box where the event occurred. In all other cases, the DialogSelect function returns FALSE. When DialogSelect returns TRUE, do whatever is appropriate as a response to the event involving that item in that particular dialog box; when it returns FALSE, do nothing.

Generally, only controls should be enabled in a dialog box; therefore your application should normally respond only when DialogSelect returns TRUE after the user clicks an enabled control, such as the OK button.

The DialogSelect function first obtains a pointer to the window containing the event. For update and activate events, the event record contains the window pointer. For other types of events, DialogSelect calls the Window Manager function FrontWindow. The Dialog Manager then makes this window the current graphics port by calling the QuickDraw procedure SetPort. Then DialogSelect prepares to handle the event by setting up text information if there are any editable text items in the active dialog box.

If the event is an update event for a dialog box, DialogSelect calls the Window Manager procedure BeginUpdate, the Dialog Manager procedure DrawDialog, and then the Window Manager procedure EndUpdate. When an item is a control defined in a control ('CNTL') resource, the rectangle added to the update region is the rectangle defined in the control resource, not the display rectangle defined in the item list resource.

The DialogSelect function handles the event as follows:

- In response to an activate or update event for the dialog box, DialogSelect activates or updates its window and returns FALSE.
- If a key-down event or an auto-key event occurs and there's an editable text item in the dialog box, DialogSelect uses TextEdit to handle text entry and editing, and DialogSelect returns TRUE for a function result. In its itemHit parameter, DialogSelect returns the item number.
- If a key-down event or an auto-key event occurs and there's no editable text item in the dialog box, DialogSelect returns FALSE.
- If the user presses the mouse button while the cursor is in an editable text item, DialogSelect responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If the editable text item is disabled, DialogSelect returns FALSE. If the editable text item is enabled, DialogSelect returns TRUE and in its itemHit parameter returns the item number. Normally, editable text items are disabled, and you use the GetDialogItemText function to read the information in the items only after the OK button is clicked.
- If the user presses the mouse button while the cursor is in a control, DialogSelect calls the Control Manager function TrackControl. If the user releases the mouse button while the cursor is in an enabled control, DialogSelect returns TRUE for a function result and in its itemHit parameter returns the control's item number. Your application should respond appropriately—for example, by performing a command after the user clicks the OK button.

- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `DialogSelect` returns `TRUE` for a function result and in its `itemHit` parameter returns the item's number. Generally, only controls should be enabled. If your application creates a complex control—such as one that measures how far a dial is moved—your application must handle mouse events in that item before passing the event to `DialogSelect`.
- If the user presses the mouse button while the cursor is in a disabled item, or if it is in no item, or if any other event occurs, `DialogSelect` does nothing.
- If the event isn't one that `DialogSelect` specifically checks for (if it's a null event, for example), and if there's an editable text item in the dialog box, `DialogSelect` calls the TextEdit procedure `TEIdle` to make the insertion point blink.

SPECIAL CONSIDERATIONS

Because `DialogSelect` handles only mouse-down events in a dialog box and key-down events in a dialog box's editable text items, you should handle other events as appropriate before passing them to `DialogSelect`. Likewise, when `DialogSelect` calls `TrackControl`, it does not allow you to specify any action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control or a picture or an application-defined item that draws a control-like object in your dialog box. You must then test for and respond to those events yourself.

Within dialog boxes, use the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support Cut, Copy, Paste, and Clear commands in editable text boxes.

The `DialogSelect` function is unreliable when running in versions of system software earlier than System 7. You shouldn't use this routine if you expect your application to run under earlier versions of system software.

SEE ALSO

Listing 6-25 on page 6-79 illustrates the use of `DialogSelect` to make the cursor blink in editable text items during null events; Listing 6-29 on page 6-92 illustrates the use of `DialogSelect` to handle mouse events in a modeless dialog box; Listing 6-33 on page 6-96 illustrates the use of `DialogSelect` to handle key-down events in editable text items; Listing 6-34 on page 6-98 illustrates the use of `DialogSelect` to handle activate events in a modeless dialog box.

DrawDialog

If you don't use any other Dialog Manager routines for handling events in a dialog box, you can use the `DrawDialog` procedure to draw its entire contents.

```
PROCEDURE DrawDialog (theDialog: DialogPtr);
```

`theDialog`	A pointer to a dialog record.

DESCRIPTION

The `DrawDialog` procedure draws the entire contents of the specified dialog box. The `DrawDialog` procedure draws all dialog items, calls the Control Manager procedure `DrawControls` to draw all controls, and calls the TextEdit procedure `TEUpdate` to update all static and editable text items and to draw their display rectangles. The `DrawDialog` procedure also calls the application-defined items' draw procedures if the items' rectangles are within the update region.

The `DialogSelect`, `ModalDialog`, `Alert`, `StopAlert`, `NoteAlert`, and `CautionAlert` routines use `DrawDialog` automatically. If you use `GetNewDialog` to create a dialog box but don't use any of these other Dialog Manager routines when handling events in the dialog box, you can use `DrawDialog` to redraw the contents of the dialog box when it's visible. If the dialog box is invisible, first use the Window Manager procedure `ShowWindow` and then use `DrawDialog`.

SEE ALSO

See the chapters "Window Manager" and "Event Manager" in this book for more information on update and activate events for windows. The `DrawControls` procedure is described in the chapter "Control Manager" in this book. The `TEUpdate` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text.*

UpdateDialog

You can use the `UpdateDialog` procedure to redraw the update region of a specified dialog box. The `UpdateDialog` procedure is also available as the `UpdtDialog` procedure.

```
PROCEDURE UpdateDialog (theDialog: DialogPtr;
                        updateRgn: RgnHandle);
```

`theDialog`	A pointer to a dialog record.
`updateRgn`	A handle to the window region that needs to be updated.

DESCRIPTION

The `UpdateDialog` procedure redraws only the region in a dialog box specified in the `updateRgn` parameter. Because the `DialogSelect`, `ModalDialog`, `Alert`, `StopAlert`, `NoteAlert`, and `CautionAlert` routines automatically call `DrawDialog` to handle update events in your alert and dialog boxes, your application might never need to use `UpdateDialog`.

Instead of drawing the entire contents of the specified dialog box, `UpdateDialog` draws only the items in the specified update region. You can use `UpdateDialog` in response to an update event, and you should usually bracket it by calls to the Window Manager procedures `BeginUpdate` and `EndUpdate`. The `UpdateDialog` procedure uses the QuickDraw procedure `SetPort` to make the dialog box the current graphics port. For drawing controls, `UpdateDialog` uses the Control Manager procedure `UpdateControls`, which is faster than the `DrawControls` procedure.

SEE ALSO

Listing 6-35 on page 6-99 illustrates the use of `UpdateDialog` to respond to update events in a modeless dialog box. See the chapter “Window Manager” in this book for more information on update and activate events for windows. The `UpdateControls` procedure is described in the chapter “Control Manager” in this book.

Application-Defined Routines

If you supply an application-defined item in a dialog box, you must provide a draw procedure for the Dialog Manager to use when displaying the item; that procedure is referred to in this section as `MyItem`. If you want the Dialog Manager to play sounds other than the system alert sound, you must provide your own sound procedure, referred to in this section as `MyAlertSound`. To supplement the Dialog Manager’s ability to handle events in the Macintosh multitasking environment, you should provide an event filter function that the Dialog Manager calls whenever it displays alert boxes and modal dialog boxes. This function is referred to as `MyEventFilter`.

MyItem

To draw your own application-defined item in a dialog box, provide a draw procedure that takes two parameters: a window pointer to the dialog box and an item number from the dialog box’s item list resource. For example, this is how you should declare the procedure if you were to name it `MyItem`:

```
PROCEDURE MyItem (theWindow: WindowPtr; itemNo: Integer);
```

`theWindow`	A pointer to the dialog record for the dialog box containing an application-defined item. If your procedure can draw in more than one dialog box, this parameter tells your procedure which one to draw in.

`itemNo`	A number corresponding to the position of an item in the item list resource for the specified dialog box. If your procedure draws more than one item, this parameter tells your procedure which one to draw.

DESCRIPTION

The Dialog Manager calls your procedure to draw an application-defined item at the time you display the specified dialog box. When calling your draw procedure, the Dialog Manager sets the current port to the dialog box's graphics port. Normally, you create an invisible dialog box and then use the Window Manager procedure `ShowWindow` to display the dialog box.

Before you display the dialog box, use the `SetDialogItem` procedure to install this procedure in the dialog record. Before using `SetDialogItem`, you must first use the `GetDialogItem` procedure to obtain a handle to an item of type `userItem`.

If you enable the application-defined item that you draw with this procedure, the `ModalDialog` procedure and the `DialogSelect` function return the item's number when the user clicks that item. If your application needs to respond to a user action more complex than this (for example, if your application needs to measure how long the user holds down the mouse or how far the user drags the cursor), your application must track the cursor itself. If you use `ModalDialog`, your event filter function must handle events inside the item; if you use `DialogSelect`, your application must handle events inside the item before handing events to `DialogSelect`.

SEE ALSO

Listing 6-17 on page 6-59 illustrates a procedure that draws a bold outline around a button of any size and shape; Listing 6-16 on page 6-58 shows the use of `GetDialogItem` and `SetDialogItem` to install this draw procedure in a dialog record. The `ShowWindow` procedure is described in the chapter "Window Manager" in this book.

MyAlertSound

If you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure and call the `ErrorSound` procedure to make it the current sound procedure. For example, you can declare a sound procedure named `MyAlertSound`, as shown here:

```
PROCEDURE MyAlertSound (soundNo: Integer);
```

`soundNo`	An integer from 0 to 3, representing the four possible alert stages.

DESCRIPTION

For each of the four alert stages that can be reported in the `soundNo` parameter, your procedure can emit any sound that you define. When the Dialog Manager calls your procedure, it passes 0 as the sound number for alert sounds specified by the `silent` constant in the alert resource. The Dialog Manager passes 1 for sounds specified by the `sound1` constant, 2 for sounds specified by the `sound2` constant, and 3 for sounds specified by the `sound3` constant.

SPECIAL CONSIDERATIONS

When the Dialog Manager detects a click outside an alert box or a modal dialog box, it uses the Sound Manager procedure `SysBeep` to play the system alert sound. By changing settings in the Sound control panel, the user can select which sound to play as the system alert sound. For consistency with system software and other Macintosh applications, your sound procedure should call `SysBeep` whenever your sound procedure receives sound number 1 (which you can represent with the `sound1` constant).

SEE ALSO

Listing 6-3 on page 6-22 illustrates how to use `MyAlertSound`. The `SysBeep` procedure is described in *Inside Macintosh: Sound.*

MyEventFilter

To supplement the Dialog Manager’s ability to handle events, your application should provide an event filter function that the Dialog Manager calls when it displays alert boxes and modal dialog boxes. Your event filter function should have three parameters and return a Boolean value. For example, this is how you would declare it if you were to name it `MyEventFilter`:

```
FUNCTION MyEventFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

`theDialog`	A pointer to a dialog record for an alert box or a modal dialog box.
`theEvent`	An event record returned by an Event Manager function such as `WaitNextEvent`.
`itemHit`	A number corresponding to the position of an item in the item list resource for the alert or modal dialog box.

DESCRIPTION

After receiving an event that it does not handle, your function should return `FALSE`. When your function returns `FALSE`, `ModalDialog` handles the event, which you pass in the parameter `theEvent`. (Your function can also change the event to simulate a different event and return `FALSE`, which passes the event to the Dialog Manager for handling.) If your function *does* handle the event, your function should return `TRUE` as a function result, and in the `itemHit` parameter return the number of the item that it handled. The `ModalDialog` procedure and, in turn, the `Alert`, `NoteAlert`, `StopAlert`, and `CautionAlert` functions then return this item number in their own `itemHit` parameters.

Your event filter function should perform the following tasks:

- return `TRUE` and the item number for the default button if the user presses Return or Enter
- return `TRUE` and the item number for the Cancel button if the user presses Esc or Command-period
- update your windows in response to update events (this allows background applications to receive update events) and return `FALSE`
- return `FALSE` for all events that your event filter function doesn't handle

You can also use the event filter function to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor in an application-defined item. For example, if you provide an application-defined item that requires you to measure how long the user holds down the mouse button or how far the user drags the cursor, use the event filter function to handle events inside that item.

The `ModalDialog` procedure calls the Event Manager function `GetNextEvent` with a mask that excludes disk-inserted events; to receive disk-inserted events, your event filter function can call the Event Manager procedure `SetSystemEventMask`.

You can use the same event filter function in most or all of your alert and modal dialog boxes.

For alert and modal dialog boxes, the Dialog Manager provides a standard event filter function that checks whether the user has pressed the Enter or Return key and, if so, returns the item number of the default button. Your event filter function should always check whether the Return key or Enter key was pressed and, if so, return the number of the default button in the `itemHit` parameter and a function result of `TRUE`.

In all alert and dialog boxes, any buttons that are activated by key sequences should invert to indicate which item has been selected. Use the Control Manager procedure `HiliteControl` to invert a button for 8 ticks, long enough to be noticeable but not so long as to be annoying. The Control Manager performs this action whenever users click a button, and your application should do this whenever the user presses the keyboard equivalent of a button click.

For modal dialog boxes that contain editable text items, your application should handle menu bar access to allow use of your Edit menu and its Cut, Copy, Paste, Clear, and Undo commands. Your event filter function should then test for and handle clicks in your Edit menu and keyboard equivalents for the appropriate commands in your Edit menu. Your application should respond by using the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands.

For an alert box, you specify a pointer to your event filter function in a parameter that you pass to the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. For a modal dialog box, specify a pointer to your event filter function in a parameter that you pass to the `ModalDialog` procedure.

SEE ALSO

Listing 6-27 on page 6-88 illustrates an event filter function. The functions `GetNextEvent` and `SetSystemEventMask` are described in the chapter "Event Manager" in this book.

Resources

This section describes resources used by the Dialog Manager for displaying alerts and dialog boxes. These resources are

- the dialog (`'DLOG'`) resource, which specifies the window type, display rectangle, and item list resource for a dialog box
- the alert (`'ALRT'`) resource, which specifies alert sounds, a display rectangle, and an item list resource for an alert box
- the item list (`'DITL'`) resource, which specifies the items—such as buttons and static text—to display in an alert box or a dialog box
- the dialog color table (`'dctb'`) resource, which lets you supply a color graphics port for a dialog box and also use colors other than the default colors in a dialog box
- the alert color table (`'actb'`) resource, which lets you use colors other than the default colors in an alert box
- the item color table (`'ictb'`) resource, which lets you change the default colors, typeface, font style, and font size of items in an alert box or a dialog box

This section describes the structures of these resources after they are compiled by the Rez resource compiler, available from APDA. If you are interested in creating the Rez input files for these resources, see "Using the Dialog Manager" beginning on page 6-17 for detailed information.

The Dialog Resource

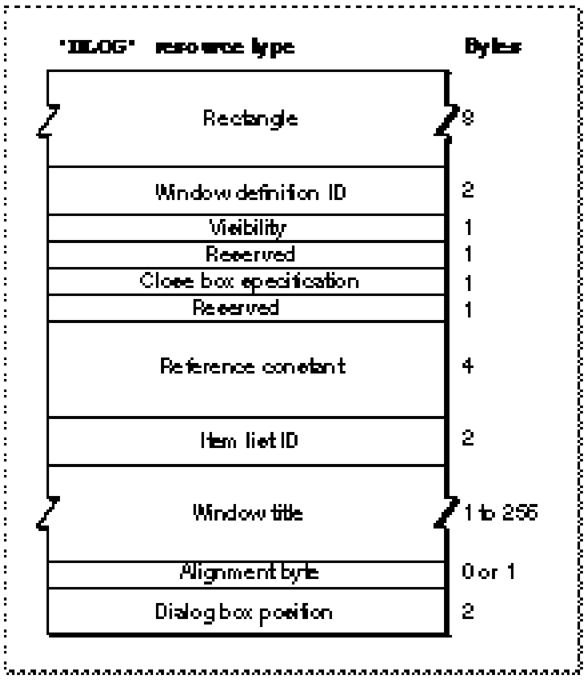
You can use a dialog resource to define a dialog box. A dialog resource is a resource of type `'DLOG'`. All dialog resources must be marked purgeable, and they must have resource ID numbers greater than 128.

To specify the items in a dialog box, you must also provide an item list resource, described beginning on page 6-151. Use the `GetNewDialog` function (described on page 6-113) to create the dialog box defined in the dialog resource.

The format of a Rez input file for a dialog resource differs from its compiled output format. This section describes the structure of a Rez-compiled dialog resource. If you are concerned only with creating a dialog resource, see “Creating Dialog Boxes” beginning on page 6-23.

Figure 6-42 shows the format of a compiled dialog resource.

Figure 6-42 Structure of a compiled dialog (`'DLOG'`) resource



The compiled version of a dialog resource contains the following elements:

- Rectangle. This determines the dialog box’s dimensions and, possibly, its position. (The last element in the dialog resource usually specifies a position for the dialog box.)
- Window definition ID.
  - If the integer 0 appears here (as specified in the Rez input file by the `dBoxProc` window definition ID), the Dialog Manager displays a modal dialog box.

- If the integer 4 appears here (as specified in the Rez input file by the `noGrowDocProc` window definition ID), the Dialog Manager displays a modeless dialog box.
- If the integer 5 appears here (as specified in the Rez input file by the `movableDBoxProc` window definition ID), the Dialog Manager displays a movable modal dialog box.

These types of dialog boxes are illustrated in Figure 6-6 on page 6-10, Figure 6-8 on page 6-12, and Figure 6-7 on page 6-11, respectively.

- Visibility. If this is set to a value of 1 (as specified by the `visible` constant in the Rez input file), the Dialog Manager displays this dialog box as soon as you call the `GetNewDialog` function. If this is set to a value of 0 (as specified by the `invisible` constant in the Rez input file), the Dialog Manager does not display this dialog box until you call the Window Manager procedure `ShowWindow`.
- Close box specification. This specifies whether to draw a close box. Normally, this is set to a value of 1 (as specified by the `goAway` constant in the Rez input file) only for a modeless dialog box to specify a close box in its title bar. Otherwise, this is set to a value of 0 (as specified by the `noGoAway` constant in the Rez input file).
- Reference constant. This contains any value that an application stores here. For example, an application can store a number that represents a dialog box type, or it can store a handle to a record that maintains state information about the dialog box or other window types. An application can use the Window Manager procedure `SetWRefCon` at any time to change this value in the dialog record for a dialog box, and you can use the `GetWRefCon` function to determine its current value.
- Item list resource ID. The ID of the item list resource that specifies the items—such as buttons and static text—to display in the dialog box.
- Window title. This is a Pascal string displayed in the dialog box's title bar only when the dialog box is modeless.
- Alignment byte. This is an extra byte added if necessary to make the previous Pascal string end on a word boundary.
- Dialog box position. This specifies the position of the dialog box on the screen. (If your application positions dialog boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager.)
  - If 0x0000 appears here (as specified by the `noAutoCenter` constant in the Rez input file), the Dialog Manager positions this dialog box according to the global coordinates specified in the rectangle element of this resource.
  - If 0xB00A appears here (as specified by the `alertPositionParentWindow` constant in the Rez input file), the Dialog Manager positions the dialog box over the frontmost window so that the window's title bar appears. This is illustrated in Figure 6-33 on page 6-63.
  - If 0x300A appears here (as specified by the `alertPositionMainScreen` constant in the Rez input file), the Dialog Manager centers the dialog box near the top of the main screen. This is illustrated in Figure 6-34 on page 6-63.
  - If 0x700A appears here (as specified in the Rez input file by the `alertPositionParentWindowScreen` constant), the Dialog Manager positions the dialog box on the screen where the user is currently working. This is illustrated in Figure 6-35 on page 6-64.

The Alert Resource

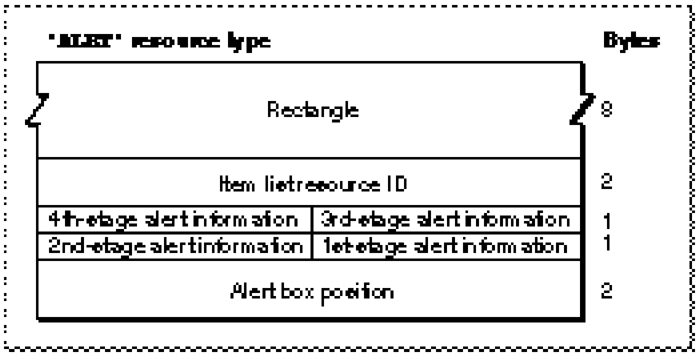
You can use an alert resource to define an alert. An alert resource is a resource of type `'ALRT'`. All alert resources must be marked purgeable, and they must have resource ID numbers greater than 128.

To specify the items in an alert box, you must also provide an item list resource, described beginning on page 6-151. To display the alert, you call either the `NoteAlert`, `CautionAlert`, `StopAlert`, or `Alert` function and pass it the resource ID of the alert resource. The `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert` functions are described in "Creating Alerts" beginning on page 6-105.

The format of a Rez input file for an alert resource differs from its compiled output format. This section describes the structure of a Rez-compiled alert resource. If you are concerned only with creating an alert resource, see "Creating Alert Sounds and Alert Boxes" beginning on page 6-18.

Figure 6-43 shows the structure of a compiled alert resource.

Figure 6-43 Structure of a compiled alert (`'ALRT'`) resource


'ALRT' resource type
Bytes
Rectangle
8
Item list resource ID
2
4th-stage alert information
3rd-stage alert information
1
2nd-stage alert information
1st-stage alert information
1
Alert box position
2

The compiled version of an alert resource contains the following elements:

- Rectangle. This determines the alert box's dimensions and, possibly, its position. (The last element in the alert resource usually specifies a position for the alert box.)
- Item list resource ID. The ID of the item list resource that specifies the items—such as buttons and static text—to display in the alert box.
- Fourth-stage alert information. This specifies the response when the user repeats the action that invokes this alert four or more consecutive times. The Dialog Manager responds in the manner specified in the 4 bits that make up this element.
  - If the first bit is set, the Dialog Manager draws a bold outline around the second item in the item list resource (typically, the Cancel button) and—if your application does not specify an event filter function—returns 2 when the user presses the Return or Enter key at the fourth consecutive occurrence of the alert. If the first bit is not set, the Dialog Manager draws a bold outline around the first item in the item list resource (typically, the OK button) and—if your application does not specify an event filter function—returns 1 when the user presses the Return or Enter key.

  - If the second bit is set, the Dialog Manager displays the alert box at this stage. If the second bit is not set, the Dialog Manager doesn't display the alert box at this stage.
  - If neither of the next 2 bits is set, the Dialog Manager plays no alert sound at this stage. If bit 3 is set and bit 4 is not set, the Dialog Manager plays the first alert sound—by default, the system alert sound. If bit 3 is not set and bit 4 is set, the Dialog Manager plays the second alert sound; by default, it plays the system alert sound twice. If both bit 3 and bit 4 are set, the Dialog Manager plays the third alert sound; by default, it plays the system alert sound three times. By defining your own alert sound (described on page 6-144) and calling the `ErrorSound` procedure (described on page 6-104) to make it the current sound procedure, you can specify your own alert sounds.

- Third-stage alert information. This specifies the response when the user repeats the action that invokes this alert three consecutive times. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- Second-stage alert information. This specifies the response when the user repeats the action that invokes this alert two consecutive times. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- First-stage alert information. This specifies the response for the first time that the user performs the action that invokes this alert. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- Alert box position. This specifies the position of the alert box on the screen. (If your application positions alert boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager.)
  - If 0x0000 appears here (as specified by the `noAutoCenter` constant in the Rez input file), the Dialog Manager positions this alert box according to the global coordinates specified in the rectangle element of this resource.
  - If 0xB00A appears here (as specified by the `alertPositionParentWindow` constant in the Rez input file), the Dialog Manager positions the alert box over the frontmost window so that the window's title bar appears. This is illustrated in Figure 6-33 on page 6-63.
  - If 0x300A appears here (as specified by the `alertPositionMainScreen` constant in the Rez input file), the Dialog Manager centers the alert box near the top of the main screen. This is illustrated in Figure 6-34 on page 6-63.
  - If 0x700A appears here (as specified in the Rez input file by the `alertPositionParentWindowScreen` constant), the Dialog Manager positions the alert box on the screen where the user is currently working. This is illustrated in Figure 6-35 on page 6-64.

The Item List Resource

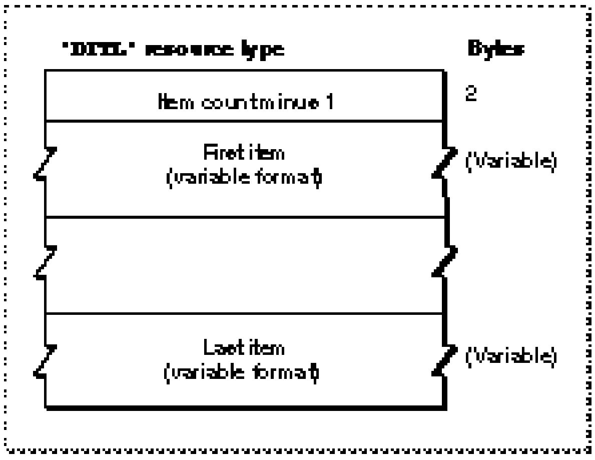
You use an item list resource to specify items—such as buttons and text—in alert boxes and dialog boxes. An item list resource is a resource with the resource type `'DITL'`. All item list resources must be marked purgeable, and they must have resource ID numbers greater than 128.

For an alert box, you specify the resource ID of the item list resource in an alert resource (described beginning on page 6-150). For a dialog box that you create with the `GetNewDialog` function, you specify the resource ID of the item list resource in a dialog resource (described beginning on page 6-148). For a dialog box that you create with either the `NewColorDialog` function (described on page 6-115) or the `NewDialog` function (described on page 6-118), you use the Resource Manager function `GetResource` to read the item list resource into memory and to provide a handle to the item list resource in memory.

The format of a Rez input file for an item list resource differs from its compiled output format. This section describes the structure of a Rez-compiled item list resource. If you are concerned only with creating an item list resource, see "Providing Items for Alert and Dialog Boxes" beginning on page 6-26.

Figure 6-44 shows the format of a compiled item list resource.

Figure 6-44 Structure of a compiled item list (`'DITL'`) resource



The compiled version of an item list resource contains the following elements:

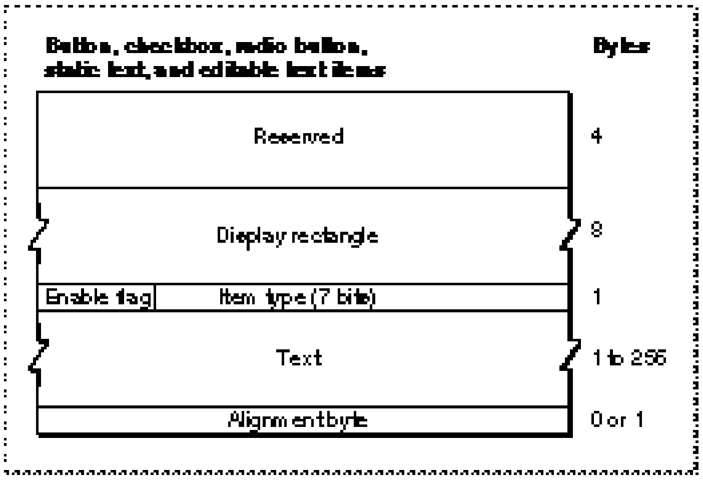
- Item count minus 1. This value is 1 less than the total number of items defined in this resource.
- A variable number of items.

The format of each item depends on its type. Figure 6-45 shows the format of an item defined to be a button, a checkbox, a radio button, a static text item, or an editable text item.

The compiled version of a button, checkbox, radio button, static text item, or editable text item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert box or dialog box; these coordinates specify the upper-left and lower-right corners of the item.

Figure 6-45 Structure of compiled button, checkbox, radio button, static text, and editable text items



- Enable flag. This specifies whether the item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type.
  - If this bit string is set to 4 (as specified in the Rez input file by the `Button` constant), then the item is a button.
  - If this bit string is set to 5 (as specified in the Rez input file by the `CheckBox` constant), then the item is a checkbox.
  - If this bit string is set to 6 (as specified in the Rez input file by the `RadioButton` constant), then the item is a radio button.
  - If this bit string is set to 8 (as specified in the Rez input file by the `StaticText` constant), then the item is static text.
  - If this bit string is set to 16 (as specified in the Rez input file by the `EditText` constant), then the item is editable text.
- Text. This specifies the text that appears in the item. This element consists of a length byte and as many as 255 additional bytes for the text. ("Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 contain recommendations about appropriate text in items.)
  - For a button, checkbox, or radio button, this is the title for that control.
  - For a static text item, this is the text of the item.
  - For an editable text item, this can be an empty string (in which case the editable text item contains no text), or it can be a string that appears as the default string in the editable text item.
- Alignment byte. This is added if necessary to make the previous text string end on a word boundary.

Figure 6-46 shows the format for an element defined to be a control, an icon, or a picture item.

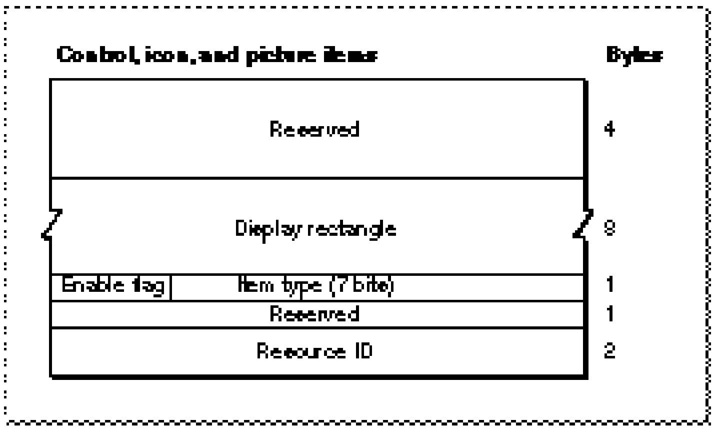


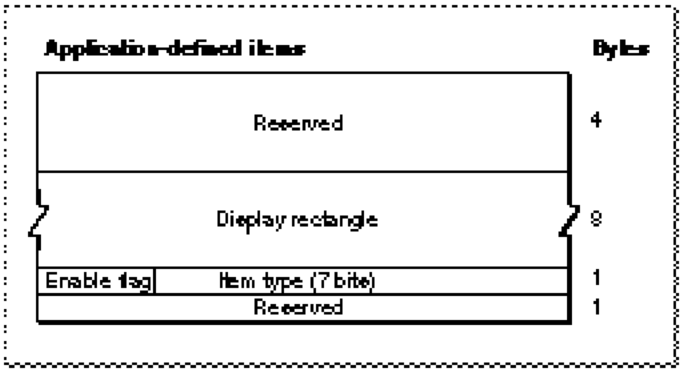
Figure 6-46 Structure of compiled control, icon, and picture items

The compiled version of a control, an icon, or a picture item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert or dialog box.
- Enable flag. This specifies whether the item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type.
  - If this 7-bit string is set to 7 (as specified in the Rez input file by the `Control` constant), then the item is a button.
  - If this is set to 32 (as specified in the Rez input file by the `Icon` constant), then the item is an icon.
  - If this is set to 64 (as specified in the Rez input file by the `Picture` constant), then the item is a QuickDraw picture.
- Resource ID.
  - For a control item, this is the resource ID of a `'CTRL'` resource.
  - For an icon item, this is the resource ID of an `'ICON'` resource and, optionally, a `'cicn'` resource
  - For a picture item, this is the resource ID of a `'PICT'` resource.

Figure 6-47 shows the format for an application-defined item.

Figure 6-47 Structure of a compiled application-defined item

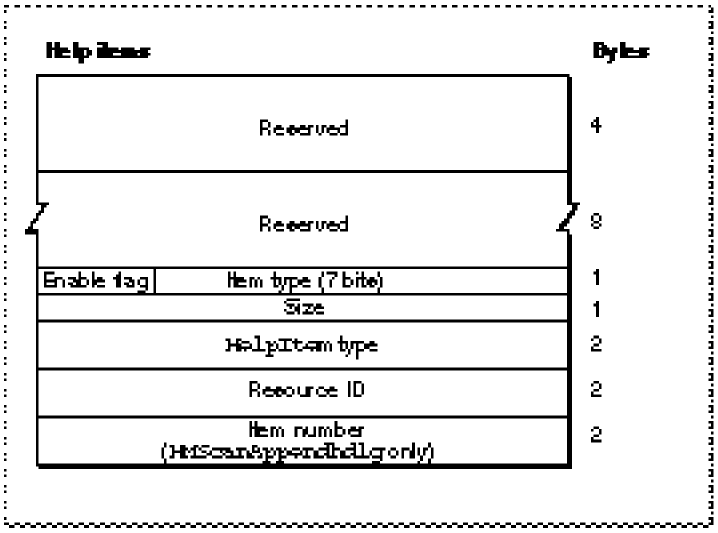


The compiled version of an application-defined item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the application-defined item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert box or dialog box.
- Enable flag. This specifies whether the application-defined item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type. This is set to a value of 0 (as specified in the Rez input file by the `UserItem` constant).

Figure 6-48 shows the format for a help item. (Help items are described in detail in the chapter "Help Manager" of *Inside Macintosh: More Macintosh Toolbox*.)

Figure 6-48 Structure of compiled help items



The compiled version of a help item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Reserved. This should be set to 0.
- Enable flag. This specifies whether the item is enabled or disabled. For help items, this bit should never be set, because the Dialog Manager cannot report to your application when mouse-down events occur inside the item.
- Item type. This is set to 1 (as specified in the Rez input file by the `HelpItem` constant).
- Size. This specifies the number of bytes contained in the rest of this element. This is set to 4 for an item identified by either the `HMScanhdlg` or `HMScanhrct` identifier, or it's set to 6 for an item identified by the `HMScanAppendhdlg` identifier.
- `HelpItem` type. This specifies the type of help item defined in the resource.
  - For an item identified by the `HMScanhdlg` identifier, this element contains the value 1.
  - For an item identified by the `HMScanhrct` identifier, this element contains the value 2.
  - For an item identified by the `HMScanAppendhdlg` identifier, this element contains the value 8.
- Resource ID. This is the resource ID of the resource containing the help messages for this alert box or dialog box.
  - For an item identified by either the `HMScanhdlg` or `HMScanAppendhdlg` identifier, this is the ID of an `'hdlg'` resource.
  - For an item identified by the `HMScanhrct` identifier, this is the ID of an `'hrct'` resource.
- Item number. This is available only for an item identified by the `HMScanAppendhdlg` identifier. This is the item number within the alert box or dialog box after which the help messages specified in the `'hdlg'` resource should be displayed. These help messages relate to the items that are appended to the alert box or dialog box. (The item list resource does not contain these 2 bytes for items identified by either the `HMScanhdlg` or `HMScanhrct` identifier.)

The Dialog Color Table Resource

On color monitors, the Dialog Manager automatically adds color to your alert and dialog boxes so that they match the colors of the windows, alert boxes, and dialog boxes used by system software. These colors provide aesthetic consistency across all monitors, from black-and-white displays to 8-bit color displays. On a color monitor, for example, the racing stripes in the title bar of a modeless dialog box are gray, the close box and window frame are in color, and the buttons and text are black.

When you create dialog resources, your application's dialog boxes use the system's default colors. Typically, this is all you need to do to provide color for your dialog boxes—with the following exceptions:

- When you need to include a color version of an icon in a dialog box, you must create a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource. Plate 2 at the front of this book shows an alert box that includes a color icon.

- n When you need to produce a blended gray color for outlining the inactive (that is, dimmed) default button, you must create a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource.

"Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 explains how to create a draw routine that outlines the default button of a dialog box. If you deactivate a dialog box, you should dim its buttons and use gray to draw the outline for the default button. Because `GetNewDialog` and `NewDialog` supply black-and-white graphics ports for dialog boxes, you can create a dialog color table resource for the dialog box to force the Dialog Manager to supply a color graphics port. Then you can use a blended gray color for the outline for the default button. (The `NewColorDialog` function supplies a color graphics port.)

Even when you create a dialog color table resource for drawing a gray outline, you should not change the system's default colors. If you feel absolutely compelled to use nonstandard colors, you can use the Dialog Manager to specify colors other than the default colors. Your application can specify its own colors for a dialog box by creating a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource (described beginning on page 6-148). You don't have to call any new routines to change the colors used in dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load a dialog color table resource with the same resource ID as the dialog resource.

Be aware, however, that nonstandard colors in your dialog boxes may initially confuse your users. Also be aware that despite any changes you may make, users can alter the colors of dialog boxes anyway by changing settings in the Color control panel.

s **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. s

A dialog color table resource has exactly the same format as a window color table (that is, a resource of type `'wctb'`), which is described in the chapter "Window Manager" of this book.

If the dialog box's content color isn't white, specify the `invisible` constant in the dialog resource. Use the Window Manager procedure `ShowWindow` to display the dialog box when it's the frontmost window. If the dialog box is a modeless dialog box that is not in front, use the Window Manager procedure `ShowHide` to display it.

The Alert Color Table Resource

On color monitors, the Dialog Manager automatically adds color to your alert boxes so that they match the colors of the windows and alerts used by system software. When you create alert resources, your application's alert boxes use the system's default colors. Typically, this is all you need to do to provide color for your alert boxes. (However, to include a color version of an icon in an alert box, you must add a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource.)

If you feel absolutely compelled to use nonstandard colors, you can use the Dialog Manager to specify colors other than the default colors. Your application can specify its own colors for an alert box by creating an alert color table (`'actb'`) resource with the same resource ID as the alert resource (described beginning on page 6-150). You don't have to call any new routines to change the colors used in alert or dialog boxes. When you call the `Alert` function, for example, the Dialog Manager automatically attempts to load an alert color table resource with the same resource ID as the alert resource.

Be aware, however, that nonstandard colors in your alert boxes may initially confuse your users. Also be aware that despite any changes you may make, users can alter the colors of dialog boxes anyway by changing settings in the Color control panel.

s **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. s

An alert color table resource has exactly the same format as a window color table (`'wctb'`) resource, which is described in the chapter "Window Manager" of this book.

The Item Color Table Resource

On color monitors, the Dialog Manager automatically draws the items in your dialog and alert boxes so that they match the colors of the items used by system software in its dialog and alert boxes. The Dialog Manager also uses the default system font when it draws the text in the static text and editable text items of your dialog and alert boxes.

If you feel absolutely compelled to use nonstandard fonts and colors, you can use the Dialog Manager to specify your own colors, typeface, font style, and font size.

Note
The Dialog Manager displays the typeface, font style, and font size you specify only on color monitors. u

Your application can specify these by creating an item color table (`'ictb'`) resource with the same resource ID as the dialog or alert box's item list resource , and then providing a dialog color table resource for a dialog box or an alert color table resource for an alert box. You don't have to call any new routines to change the colors, typefaces, font styles, or font sizes used in dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load an item color table resource with the same resource ID as the item list resource.

Note
To make it easier to localize your application for other script systems, you should not change the font. Do not use a smaller font, such as 9-point Geneva; some script systems, such as KanjiTalk, require 12-point fonts. u

Also, be aware that nonstandard colors for items in your dialog and alert boxes may initially confuse your users.

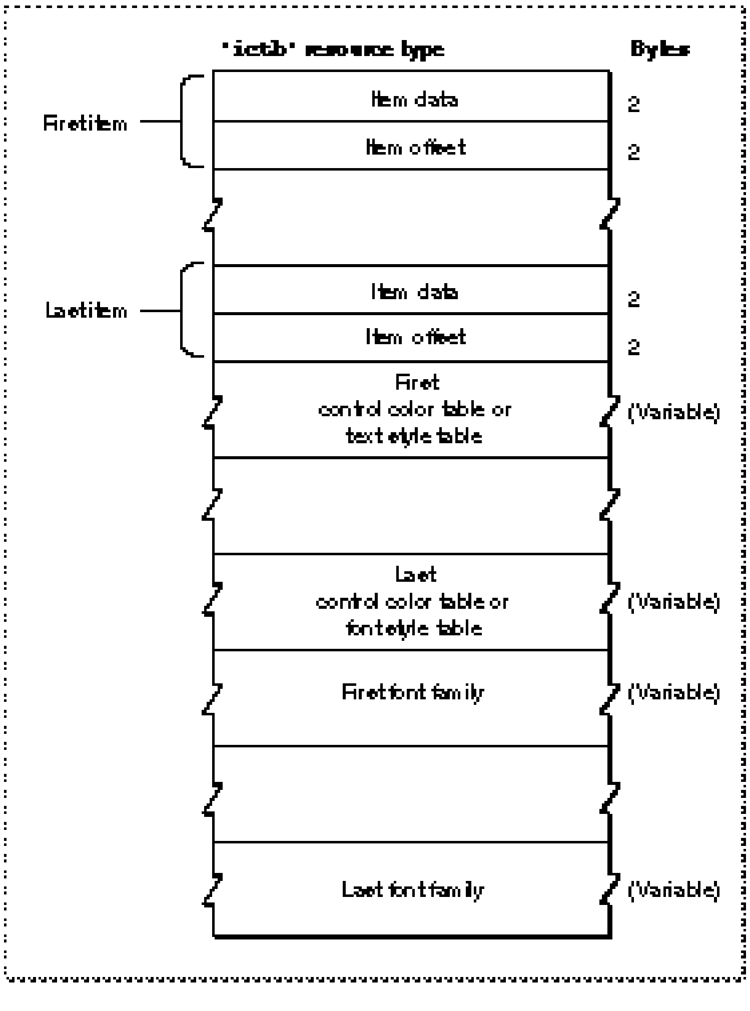
s **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. s

If you want to provide an item color table resource for an alert box or a dialog box, you must create an alert color table resource or a dialog color table resource, even if the item color table resource has no actual color information and describes only static text and editable text style changes.

An item color table resource is a resource of type `'ictb'`. All item color table resources must have resource ID numbers greater than 128.

There is no Rez template available for creating item color table resources. When you compile an item color table resource, it should follow the format illustrated in Figure 6-49.

Figure 6-49 Structure of a compiled item color table resource



You define an item color table resource for a dialog box or an alert box by specifying these elements in a resource with the `'ictb'` resource type:

- Items. These consist of a variable number of items, corresponding to those in an item list resource with the same resource ID as this item color table resource.
- Control color tables and text style tables.
  - A **control color table** defines the colors used in a control. Several controls can share the same control color table.
  - A **text style table** defines the font family, font style, font size, and color of text in an editable text item or a static text item. Several editable text and static text items can share the same text style table.
- Optionally, a list of font families. If you use any text style tables, you generally conclude the item color table resource with a list of text strings, each of which specifies a font family. Although you may specify font numbers instead of font names, it's much more reliable to specify names, because system software may renumber these fonts as they are installed and removed. For every editable text item and static text item listed at the top of the item color table resource, specify a font family at the bottom of the resource.

The information contained in an element depends on the type of item it describes:

- Item data. This contains information about how this item is described in the rest of this resource.
  - For a control, this is the length (in bytes) of its control color table.
  - For a static text item or an editable text item, the bits of this element determine which elements of the text style table to use and are interpreted as follows:

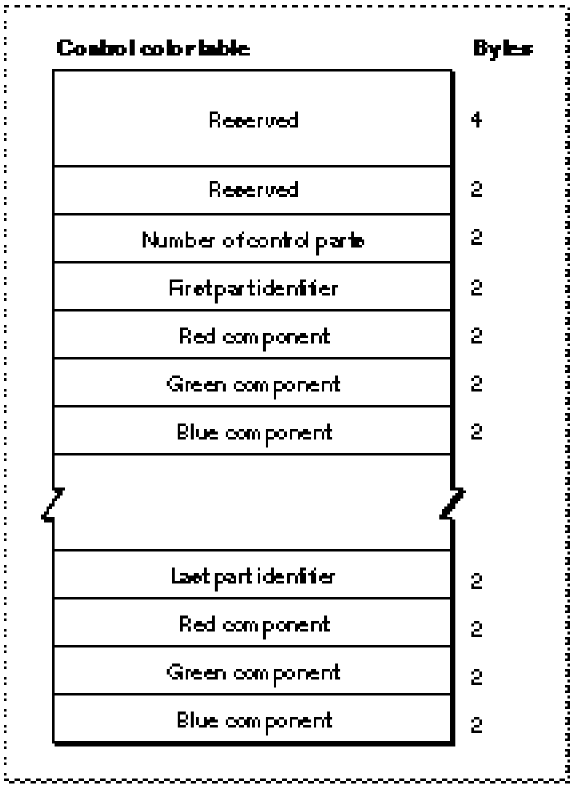
Bit	Meaning
0	Change the font family.
1	Change the typeface.
2	Change the font size.
3	Change the font foreground color.
4	Add the font size.
13	Change the font background color.
14	Change the font mode.
15	The font element is an offset to the name.

- Item offset. The number of bytes from the beginning of the resource to either the control color table or the text style table that describes this item.

When both the item data and item offset elements are set to 0, then the control or text item is drawn with the default colors, typeface, font size, and font style. Even if only the first few items of the dialog box have color style information, there must be room for all of the items actually in the box (with the item data and item offset elements of the unused entries set to 0).

For controls, the colors are described by a color table identical to a `'cctb'` resource used by the Control Manager. Multiple controls can use the same color table. If the resource sets both the item data and the item offset element to 0, then the system's default colors are used for the control. The format of a control color table is illustrated in Figure 6-50.

Figure 6-50 Structure of a compiled control color table



A control color table consists of the following elements:

- Reserved. This should always be set to a value of 0.
- Reserved. Again, should always be set to a value of 0.
- Number of control parts. For standard controls other than scroll bars, this should be set to 3, because a standard control uses only three parts: frame, control body, and text. For scroll bars, this should be set to 12; see the description of the control color table resource in the chapter "Control Manager" for information on specifying the colors for a scroll bar. To create a control that uses other parts, you must create a custom `'CDEF'` resource, as described in the chapter "Control Manager" in this book.

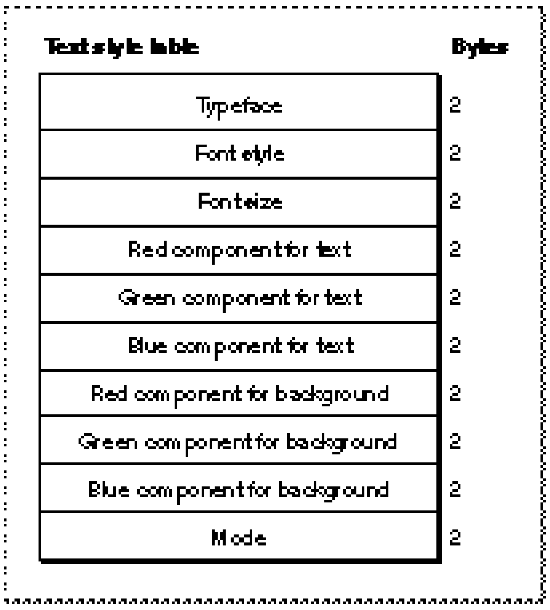
- Part identifier. This is a value that identifies a part of the first control. The following list shows the values and constants they represent for the standard controls other than scroll bars. For information on the part identifiers for a scroll bar, see the description of the control color table resource in the chapter “The Control Manager” in this book. They can be listed in any order in the control color table.

Constant	Value	Control part
`cFrameColor`	0	Frame
`cBodyColor`	1	Body
`cTextColor`	2	Text (such as titles)

- Red component. This is an integer that represents the intensity of the red component of the color to use when drawing this control part.
- Green component. This is an integer that represents the intensity of the green component of the color to use when drawing this control part.
- Blue component. This is an integer that represents the intensity of the blue component of the color to use when drawing this control part.
- Part identifier, and the red, green, and blue color components for the next control part. Specify color components for every part of this control whose color you want to change. If a part is not listed in the control color table, the Dialog Manager draws it in its default color.

Figure 6-51 shows the format of a text style table.

Figure 6-51 Structure of a compiled text style table



Text style table	Bytes
Typeface	2
Font style	2
Font size	2
Red component for text	2
Green component for text	2
Blue component for text	2
Red component for background	2
Green component for background	2
Blue component for background	2
Mode	2

The text style table must be 20 bytes long, as shown in Figure 6-51. Multiple editable text and static text items can use the same text style record. To display text in the standard typeface, color, font size, and font style, set the item data and item offset elements for the item to 0. Allocate space for all fields in the text style table, even if they are not used.

A text style table consists of the following elements (see *Inside Macintosh: Text* for a discussion of font families, font style, and point sizes):

- Typeface. This is the name of the font family to use. If bit 15 in the item data element is set to 1, then this element contains an offset (in bytes) to a font name element at the end of the resource. If bit 0 in the item data element is set to 1, then this element contains the number of a font family. If bit 0 in the item data element is set to 0, this element is set to 0, and the system default font is used.
- Font style. This is the font style to use. If bit 1 in the item data element is set to 1, then this element uses the bits of the low-order byte to describe which styles to apply to the text. If all bits in the low-order byte are set to 0, the plain font style is used. The bit numbers and the styles they represent are

Bit value	**Style**
0	Bold
1	Italic
2	Underline
3	Outline
4	Shadow
5	Condensed
6	Extended

- Font size. This is the point size of the font. If bit 2 in the item data element is set to 1, this element contains a value representing a point size. If bit 4 in the item data element is set to 1, this element contains a value to add to the current point size of the text. If bit 0 in the item data element is set to 0, this element is set to 0, and the system font size (12) is used.
- Text red color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the red component of the color to use when drawing the text.
- Text green color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the green component of the color to use when drawing the text.
- Text blue color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the blue component of the color to use when drawing the text.
- Background red color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the red component of the color to use when drawing the background behind the text.

- Background green color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the green component of the color to use when drawing the background behind the text.
- Background blue color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the blue component of the color to use when drawing the background behind the text.
- Mode. If bit 14 in the item data element is set to 1, this element contains an integer that represents how characters are placed in the bit image. The values that the Dialog Manager interprets and the constants that represent them are listed here. See *Inside Macintosh: Imaging* for a discussion of source transfer modes.

Constant	Value
`scrOr`	1
`srcXor`	2
`srcBic`	3

Summary of the Dialog Manager

Pascal Summary

Constants

```
CONST
{checking for AppendDITL, ShortenDITL, CountDITL using Gestalt function}
   gestaltDITLExtAttr = 'ditl';  {Gestalt selector for AppendDITL, etc.}
   gestaltDITLExtPresent = 0;    {if this bit's set, then AppendDITL, }
                                 { ShortenDITL, & CountDITL are available}

{item types for GetDialogItem, SetDialogItem}
   ctrlItem       = 4;     {add this constant to the next four constants}
   btnCtrl        = 0;     {standard button control}
   chkCtrl        = 1;     {standard checkbox control}
   radCtrl        = 2;     {standard radio button}
   resCtrl        = 3;     {control defined in a control resource}
   helpItem       = 1;     {help balloons}
   statText       = 8;     {static text}
   editText       = 16;    {editable text}
   iconItem       = 32;    {icon}
   picItem        = 64;    {QuickDraw picture}
   userItem       = 0;     {application-defined item}
   itemDisable    = 128;   {add to any of the above to disable it}

{item numbers of OK and Cancel buttons in alert boxes}
   ok       = 1;  {first button is OK button}
   cancel   = 2;  {second button is Cancel button}

{resource IDs of alert box icons}
   stopIcon    = 0;
   noteIcon    = 1;
   cautionIcon = 2;

{constants used for theMethod parameter in AppendDITL}
   overlayDITL       = 0;  {overlay existing items}
   appendDITLRight   = 1;  {append at right}
   appendDITLBottom  = 2;  {append at bottom}
```

```
{constants for procID parameter of NewDialog, NewColorDialog}
   dBoxProc          = 1;  {modal dialog box}
   noGrowDocProc     = 4;  {modeless dialog box}
   movableDBoxProc   = 5;  {movable modal dialog box}
```

Data Types

```
TYPE  DialogPtr      =  WindowPtr;
ResumeProcPtr        =  ProcPtr;
SoundProcPtr         =  ProcPtr;
ModalFilterProcPtr   =  ProcPtr;
DialogPeek           =  ^DialogRecord;
DialogRecord         =
RECORD
   window:     WindowRecord;  {dialog window}
   items:      Handle;        {item list resource}
   textH:      TEHandle;      {current editable text item}
   editField:  Integer;       {editable text item number minus 1}
   editOpen:   Integer;       {used internally}
   aDefItem:   Integer;       {default button item number}
END;

DITLMethod = Integer;
```

Dialog Manager Routines

Initializing the Dialog Manager

```
PROCEDURE InitDialogs       (resumeProc: ResumeProcPtr);
PROCEDURE ErrorSound        (soundProc: SoundProcPtr);
PROCEDURE SetDialogFont     (fontNum: Integer); {also spelled SetDAFont}
```

Creating Alerts

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
FUNCTION Alert              (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION StopAlert          (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION NoteAlert          (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION CautionAlert       (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION GetAlertStage      : Integer;
PROCEDURE ResetAlertStage;
```

Creating and Disposing of Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
FUNCTION GetNewDialog       (dialogID: Integer; dStorage: Ptr;
                             behind:  WindowPtr): DialogPtr;
FUNCTION NewColorDialog     (dStorage: Ptr; boundsRect: Rect; title:
                             Str255; visible: Boolean; procID: Integer;
                             behind: WindowPtr; goAwayFlag: Boolean;
                             refCon: LongInt; items: Handle): DialogPtr;
FUNCTION NewDialog          (dStorage:  Ptr; boundsRect: Rect; title:
                             Str255; visible: Boolean; procID: Integer;
                             behind: WindowPtr; goAwayFlag: Boolean;
                             refCon: LongInt; items: Handle):  DialogPtr;
PROCEDURE CloseDialog       (theDialog: DialogPtr);
PROCEDURE DisposeDialog     (theDialog: DialogPtr);
```

Manipulating Items in Alert and Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE GetDialogItem     (theDialog: DialogPtr; itemNo: Integer;
                             VAR itemType: Integer; VAR item: Handle;
                             VAR box: Rect);
PROCEDURE SetDialogItem     (theDialog: DialogPtr; itemNo: Integer;
                             itemType: Integer; item: Handle; box: Rect);
PROCEDURE HideDialogItem    (theDialog: DialogPtr; itemNo: Integer);
PROCEDURE ShowDialogItem    (theDialog: DialogPtr; itemNo: Integer);
FUNCTION FindDialogItem     (theDialog: DialogPtr; thePt: Point): Integer;
PROCEDURE AppendDITL        (theDialog: DialogPtr; theDITL: Handle;
                             theMethod: DITLMethod);
PROCEDURE ShortenDITL       (theDialog: DialogPtr; numberItems: Integer);
FUNCTION CountDITL          (theDialog: DialogPtr): Integer;
```

Handling Text in Alert and Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE ParamText         (param0: Str255; param1: Str255;
                             param2: Str255; param3: Str255);
PROCEDURE GetDialogItemText (item: Handle; VAR text: Str255);
PROCEDURE SetDialogItemText (item: Handle; text: Str255);
PROCEDURE SelectDialogItemText
                            (theDialog: DialogPtr; itemNo: Integer;
                             strtSel: Integer; endSel: Integer);
PROCEDURE DialogCut         (theDialog: DialogPtr);
PROCEDURE DialogCopy        (theDialog: DialogPtr);
PROCEDURE DialogPaste       (theDialog: DialogPtr);
PROCEDURE DialogDelete      (theDialog: DialogPtr);
```

Handling Events in Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE ModalDialog       (filterProc: ModalFilterProcPtr; VAR itemHit:
                             Integer);
FUNCTION IsDialogEvent      (theEvent: EventRecord): Boolean;
FUNCTION DialogSelect       (theEvent: EventRecord; VAR theDialog:
                             DialogPtr; VAR itemHit: Integer): Boolean;
PROCEDURE DrawDialog        (theDialog: DialogPtr);
PROCEDURE UpdateDialog      (theDialog: DialogPtr; updateRgn: RgnHandle);
```

Application-Defined Routines

```
PROCEDURE MyItem            (theWindow: WindowPtr; itemNo: Integer);
PROCEDURE MyAlertSound      (soundNo: Integer);
FUNCTION MyEventFilter      (theDialog: DialogPtr; VAR theEvent:
                             EventRecord; VAR itemHit: Integer): Boolean;
```

C Summary

Constants

```
enum {
/*checking for AppendDITL, ShortenDITL, CountDITL using Gestalt function*/
   #define gestaltDITLExtAttr 'ditl' /*Gestalt selector*/
   gestaltDITLExtPresent = 0     /*if this bit's set, then AppendDITL, */
                                 /* ShortenDITL, & CountDITL are available*/
};

enum {
/*item types for GetDItem, SetDItem*/
   ctrlItem       = 4,  /*add this constant to the next four constants*/
   btnCtrl        = 0,  /*standard button control*/
   chkCtrl        = 1,  /*standard checkbox control*/
   radCtrl        = 2,  /*standard radio button*/
   resCtrl        = 3,  /*control defined in a control resource*/
   statText       = 8,  /*static text*/
   editText       = 16, /*editable text*/
   iconItem       = 32, /*icon*/
   picItem        = 64, /*QuickDraw picture*/
   userItem       = 0,  /*application-defined item*/
   helpItem       = 1,  /*help balloons*/
   itemDisable    = 128,/*add to any of the above to disable it*/

/*item numbers of OK and Cancel buttons in alert boxes*/
   ok       = 1,  /*first button is OK button*/
   cancel   = 2,  /*second button is Cancel button*/

/*resource IDs of alert box icons*/
   stopIcon    = 0,
   noteIcon    = 1,
   cautionIcon = 2
};

enum {
/*constants used for theMethod parameter in AppendDITL*/
   overlayDITL       = 0,  /*overlay existing items*/
   appendDITLRight   = 1,  /*append at right*/
   appendDITLBottom  = 2   /*append at bottom*/
};

enum {
/*constants for procID parameter of NewDialog, NewColorDialog*/
   dBoxProc          = 1,  /*modal dialog box*/
   noGrowDocProc     = 4,  /*modeless dialog box*/
   movableDBoxProc   = 5   /*movable modal dialog box*/
};
```

Data Types

```
typedef WindowPtr DialogPtr;

typedef struct DialogRecord DialogRecord;
typedef struct DialogRecord *DialogPeek;

struct DialogRecord{
      WindowRecord   window;     /*dialog window*/
      Handle         items;      /*item list resource*/
      TEHandle       textH;      /*current editable text item*/
      short          editField;  /*editable text item number minus 1*/
      short          editOpen;   /*used internally*/
      short          aDefItem;   /*default button item number*/
};
```

```
typedef pascal void (*ResumeProcPtr)(void);
typedef pascal void (*SoundProcPtr)(void);
typedef pascal Boolean (*ModalFilterProcPtr)(DialogPtr theDialog,
                                    EventRecord *theEvent, short *itemHit);
typedef short DITLMethod;
```

Dialog Manager Routines

Initializing the Dialog Manager

```
pascal void InitDialogs     (ResumeProcPtr resumeProc);
pascal void ErrorSound      (SoundProcPtr soundProc);
pascal void SetDialogFont   (short fontNum); /*also spelled SetDAFont*/
```

Creating Alerts

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal short Alert          (short alertID, ModalFilterProcPtr filterProc);
pascal short StopAlert      (short alertID, ModalFilterProcPtr filterProc);
pascal short NoteAlert      (short alertID, ModalFilterProcPtr filterProc);
pascal short CautionAlert   (short alertID, ModalFilterProcPtr filterProc);
#define GetAlertStage()     (* (short*) 0x0A9A);
pascal void ResetAlertStage (void);
```

Creating and Disposing of Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal DialogPtr GetNewDialog
                             (short dialogID, void *dStorage,
                              WindowPtr behind);
pascal DialogPtr NewColorDialog
                             (void *dStorage, const Rect *boundsRect,
                              ConstStr255Param title, Boolean visible,
                              short procID, WindowPtr behind,
                              Boolean goAwayFlag, long refCon, Handle items);
pascal DialogPtr NewDialog
                             (void *dStorage, const Rect *boundsRect,
                              ConstStr255Param title, Boolean visible,
                              short procID, WindowPtr behind,
                              Boolean goAwayFlag, long refCon,
                              Handle items);
pascal void CloseDialog      (DialogPtr theDialog);
pascal void DisposeDialog    (DialogPtr theDialog);
```

Manipulating Items in Alert and Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void GetDialogItem   (DialogPtr theDialog, short itemNo,
                             short *itemType, Handle *item, Rect *box);
pascal void SetDialogItem   (DialogPtr theDialog, short itemNo, short
                             itemType, Handle item, const Rect *box);
pascal void HideDialogItem  (DialogPtr theDialog, short itemNo);
pascal void ShowDialogItem  (DialogPtr theDialog, short itemNo);
pascal short FindDialogItem (DialogPtr theDialog, Point thePt);
pascal void AppendDITL      (DialogPtr theDialog, Handle theDITL,
                             DITLMethod theMethod);
pascal void ShortenDITL     (DialogPtr theDialog, short numberItems);
pascal short CountDITL      (DialogPtr theDialog);
```

Handling Text in Alert and Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void ParamText       (ConstStr255Param param0,
                             ConstStr255Param param1,
                             ConstStr255Param param2,
                             ConstStr255Param param3);
pascal void GetDialogItemText
                            (Handle item, Str255 text);
pascal void SetDialogItemText
                            (Handle item, ConstStr255Param text);
pascal void SelectDialogItemText
                            (DialogPtr theDialog, short itemNo,
                             short strtSel, short endSel);
pascal void DialogCut       (DialogPtr theDialog);
pascal void DialogCopy      (DialogPtr theDialog);
pascal void DialogPaste     (DialogPtr theDialog);
pascal void DialogDelete    (DialogPtr theDialog);
```

Handling Events in Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void ModalDialog     (ModalFilterProcPtr filterProc, short *itemHit);
pascal Boolean IsDialogEvent(const EventRecord *theEvent);
pascal Boolean DialogSelect (const EventRecord *theEvent,
                             DialogPtr *theDialog, short *itemHit);
pascal void DrawDialog      (DialogPtr theDialog);
pascal void UpdateDialog    (DialogPtr theDialog, RgnHandle updateRgn);
```

Application-Defined Routines

```
pascal void MyItem          (WindowPtr theWindow, short itemNo);
pascal void MyAlertSound    (short soundNo);
pascal Boolean MyEventFilter(DialogPtr theDialog, *EventRecord theEvent,
                              *short itemHit);
```

Assembly-Language Summary

Data Structures

DialogRecord Data Structure

0	`dWindow`	156 bytes	window record for the alert box or dialog box
156	`items`	long	handle to the item list resource for the alert box or dialog box
160	`teHandle`	long	handle to the current editable text item
164	`editField`	word	current editable text item
166	`editOpen`	word	used internally
168	`aDefItem`	word	item number of the default button

Global Variables

`DAStrings`	Handles to text strings specified with the `ParamText` procedure
`DABeeper`	Address of current sound procedure
`DlgFont`	Font number for text in dialog boxes and alert boxes
`ACount`	Alert stage number (0 through 3) of the last alert
`ANumber`	Resource ID of last alert
`ResumeProc`	Address of resume procedure (should not be used in System 7)

C H A P T E R 7

Finder Interface

Contents

The **Finder** is an application that works with the system software to keep track of files and manage the user's desktop display. This chapter describes the programming interface your application should use to interact with the Finder.

To use this chapter, you should be familiar with the Resource Manager. See the chapter "Introduction to the Macintosh Toolbox" in this book for general information about resources; detailed information about the Resource Manager and its routines is provided in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. Virtually all software intended for Macintosh computers must use the Finder-related resources described in this chapter.

Read this chapter to learn how to

- set up the resources the Finder needs to display and start up your application
- set up the resources the Finder uses to display information about other files related to your application
- check or change Finder-related information stored in a volume's catalog file
- support stationery pads
- use the directories generally organized within the System Folder

This chapter does not explain how to use Apple events to communicate with the Finder. When a user opens or prints a file from the Finder, the Finder sends information to your application so that it can open or print the file. In System 7, applications that support high-level events receive this information through the required Apple events.

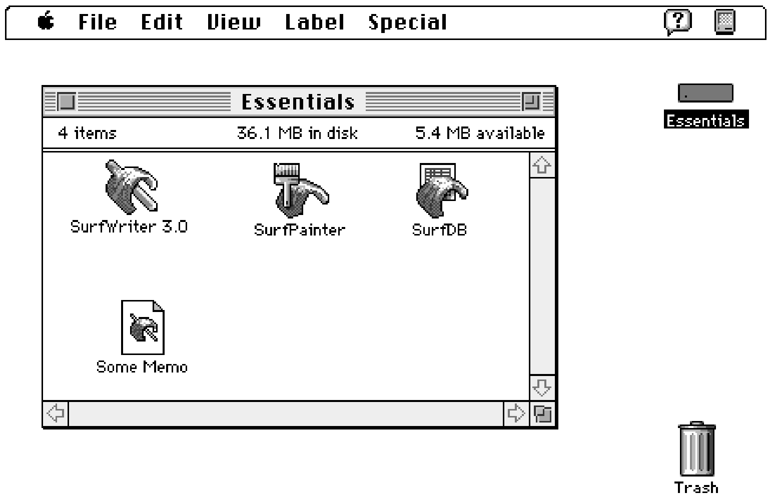
Refer to *Inside Macintosh: Interapplication Communication* for instructions on how your application should respond to these required Apple events that the Finder sends to your application: Open Application, Open Documents, Print Documents, and Quit Application. In addition, your application can use another set of Apple events—called Finder events—to request services from the Finder. For example, your application can ask the Finder to perform such operations as launching another application on your behalf. Refer to *Inside Macintosh: Interapplication Communication* for more details.

Introduction to the Finder Interface

The Finder is an application that manages the user's desktop interface. The **desktop** is the working environment displayed on the Macintosh computer—namely, the gray background area on the screen.

On the desktop, the Finder displays icons representing your application and the documents it creates, and it tracks user activity. An **icon** is an image that the Finder displays to graphically represent some object—such as a file, a folder, or the Trash—that the user can manipulate. For example, Figure 7-1 on the next page shows icons that the Finder displays for several sample applications (called SurfWriter 3.0, SurfPainter, and SurfDB) and for a text document (named Some Memo) that a user has created with the SurfWriter application. These icons are displayed in a window that the Finder uses to display the contents of the disk icon labeled Essentials.

Figure 7-1 Application and document icons in a window on the desktop



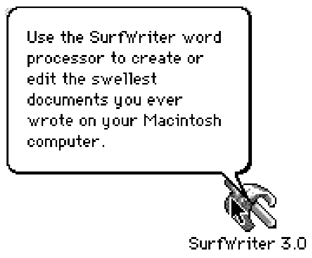
To distinguish your product for the user, you should design your own icons for all the files associated with your application. For each file type that your application uses or creates, you should define large, small, black-and-white, and 4-bit and 8-bit color icons—each in a separate resource. Your application can then use another resource, called a *bundle resource*, to assign these icons to all your files of a particular type. For example, the document icon representing Some Memo in Figure 7-1 is the icon that the SurfWriter application assigns to all text files that it creates. When double-clicking the icon for Some Memo, the user asks the Finder to launch the SurfWriter application, which in turn responds by opening the document Some Memo in a window.

Stationery pads are files that a user creates to serve as templates for other documents. **Editions** are special files that contain data to be shared among applications. **Query documents** contain commands and data in a format appropriate for a database or other data source. If your application supports any of these document types, you can create icons for the Finder that distinguish the stationery pads, editions, and query documents that users create with your application. For example, Plate 4 at the front of this book shows customized stationery pad and edition icons used for documents created with the SurfWriter application. (Editions are described in *Inside Macintosh: Interapplication Communication*. Query documents are described in *Inside Macintosh: Communications*.)

You might also like your application to create customized icons for documents on the desktop. Or, if instead of producing an application, you produce and distribute information documents (such as database files, stationery pads, query documents, clip art libraries, or dictionaries) to be used by other applications, you can also provide customized icons for the Finder that distinguish your documents.

Macintosh users have access to online assistance in the form of help balloons. You can customize the help balloon that the Finder displays for your application icon. For example, Figure 7-2 shows a customized help balloon for the SurfWriter application icon.

Figure 7-2 A customized help balloon for an application icon

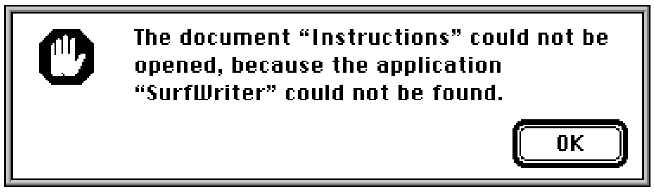

Use the SurfWriter word processor to create or edit the swellest documents you ever wrote on your Macintosh computer.

When appropriate, the Finder starts up your application and uses Apple events to tell your application what documents to open or print. To perform these tasks, the Finder relies on information you provide through resources. When the user creates or installs a file, the File Manager (described in *Inside Macintosh: Files*) initially stores some of this information in the volume's catalog file. (The **catalog file** is a special file, located on a volume, that contains information about the hierarchical organization of files and folders on that volume.)

The Finder extracts from the catalog file the information you provide in your resources and, for quick access to your resource information, the Finder uses that information to build either a desktop database for all volumes over 2 MB or a Desktop file for volumes under 2 MB. (The **desktop database** is a Finder-maintained database of icons, file types, applications, version data, and comments; the **Desktop file** is a resource file in which the Finder stores this information for volumes under 2 MB.)

You can even specify resources that identify your application when the user tries to open a document and your application is missing. For example, if a user tries to open a document named Instructions and the SurfWriter application is missing from the user's computer system, the Finder displays the alert box in Figure 7-3.

Figure 7-3 A Finder message identifying a missing application


The document "Instructions" could not be opened, because the application "SurfWriter" could not be found.

The **System Folder** is a directory that contains the software that Macintosh computers use to start up. The System Folder includes a set of folders for storing related files. Your application may use several of these folders for storing its files. For example, you may want to use the Preferences folder to store preferences files that your application needs when starting up.

About the Finder Interface

You can use the Finder interface to

- create the resources—such as those describing icons—that the Finder uses to extract and to display information about your application and its documents (Generally, all applications should provide these resources for their files.)
- determine and change the Finder information structure stored in a catalog file (Generally, most applications need to determine—and many might wish to set—information in the catalog file.)
- support stationery pads so that users can easily use templates for their documents (Generally, most applications that create documents should support stationery pads.)
- locate the directories typically located in the System Folder (Generally, many applications will want to access these directories.)

Using the Finder Interface

The Finder needs quick access to some key information about your application, such as what icons to use when displaying your application and its documents. You supply most of this information in the resource fork of your application file.

The Finder extracts this information and uses it to maintain its own database of the resources it needs. The Finder records the location of your application on disk in this database so that it can find your application quickly when the user opens one of your documents.

For compatibility with the Finder, your application should have

- a signature resource, so that the Finder can identify and start up your application when a user double-clicks documents created by your application
- a set of resources that describe icons that visually represent your application and any documents it creates
- a set of file reference resources, to link icons with the file types they represent and to allow users to launch your application by dragging document icons to your application icon
- a bundle resource, to group together your application's signature, icon, and file reference resources
- a size resource, to tell the Finder how much memory to allocate for your application when it starts up and whether your application supports various system software features
- either a missing-application name string resource in your application's documents (to display the name of your application if the user tries to open or print a document

created by your application when your application is missing) or an application-missing message string resource in your application's documents (to explain why the user can't open or print a document used only by your application)

Note

Supply a missing-application name string resource for documents that you intend for users to open with your application; supply an application-missing message string resource for documents (such as preferences files) that your application uses but that users shouldn't open. You supply only one of these resources in a document—never both. u

Your application can also make use of these resources:

- version resources, so that users can easily find out the version of your application and, if applicable, the version of your application's superset of files
- a help resource, which the Finder uses to display your customized Balloon Help message for your application, control panel, system extension, or desk accessory icon

If you sell or distribute data in the form of a document to be used by other applications, you can assist users by providing

- an appropriate file type to allow users to open your document from the Finder by dragging its icon to an application icon or by choosing the Open command from the File menu within an application
- the resources describing an icon family to represent your document to the user
- a missing-application name string resource or an application-missing message string resource, so the Finder can assist users who try to open or print your documents from the Finder
- version resources, so that users can easily find out the version of your document and, if your document file is one of a larger collection of files, the version of the entire superset of files

A catalog file exists on every volume to maintain relationships between the files and directories on that volume. (A volume is any storage medium formatted to contain files.) Although it's used mostly by the File Manager, the catalog file also contains information used by the Finder. You can always check the information in the catalog file. In particular, you may want to check the file type or creator for a file, or you may want to check or set one of the Finder flags for a document. When opening a document, your application should check a Finder flag to determine if the document is a stationery pad, and, if it is, your application should copy the document's contents into a new document and open the new document in an untitled window.

Your application might wish to use the folders located in the System Folder. Those you're most likely to want to access are Preferences, Temporary Items, and Trash. For example, you might wish to check for the existence of a user's configuration file in Preferences, create a temporary file in Temporary Items, or—if your application runs out of storage when trying to save a file—check how much storage is taken by items in the Trash directory and report this to the user. You can use the `FindFolder` function to get the path information you need to gain access to these system-related directories.

In System 7, users can create Finder objects called *aliases* to aid them in organizing their files. Ordinarily, when the user wants to open or print files, your application does not need to be concerned with whether they are aliases because the Finder resolves aliases before passing them to your application. However, if your application bypasses the Finder (or the Standard File Package, which is described in *Inside Macintosh: Files*) when manipulating documents, it should check for and resolve aliases itself by using the Alias Manager function `ResolveAliasFile`.

The rest of this chapter describes in detail how to use these Finder features in your application.

Giving a Signature to Your Application and a Creator and a File Type to Your Documents

The Finder identifies your application through its **signature,** a unique four-character sequence. The signature must not conflict with the signature of any other application. To ensure uniqueness, you must register your application's signature with Apple Computer, Inc., at Macintosh Developer Technical Support.

Note
There is no need to register your own resource types because they're usually used in only your own applications or documents. u

You must include in your resource file a special resource that has your application's signature as its resource type. By convention, the signature resource has a resource ID number of 0. The signature resource typically contains a string that specifies the name, version number, and release date of your application. If you do not provide specific version information through a version resource (described in "Providing Version Resources" beginning on page 7-31), the Finder displays the string stored in the signature resource when the user selects your application and chooses Get Info from the File menu.

Listing 7-1 illustrates a signature resource in Rez input format. (Rez is the resource compiler provided with Apple's Macintosh Programmer's Workshop [MPW], available from APDA.)

Listing 7-1 Rez input for a signature resource

```
type 'WAVE' as 'STR ';              /*WAVE is the signature*/
resource 'WAVE' (0, purgeable) {    /*resource ID is 0*/
   "SurfWriter 3.0 © 1992"          /*default Get Info string*/
};
```

Note
The signature resource alone is not sufficient to establish your application's signature. You must also supply a bundle resource, described in "Creating a Bundle Resource" beginning on page 7-20. u

Whenever your application creates a document, it assigns the document a creator and a file type. Typically, as described in “Using Finder Information in the Catalog File” beginning on page 7-32, your application sets its signature as the document’s creator. When a user double-clicks a document or selects it and chooses Open or Print from the Finder’s File menu, the Finder reads the creator field of that file to find the document’s creator. The Finder then searches for an application with a signature by that name. When it finds that application, the Finder launches it.

If the document’s creator is your application’s signature, for example, the Finder calls the Process Manager to start your application. The Finder then passes to your application the information it needs to open or print the document; since the introduction of System 7, the Finder has used Apple events to pass this information to your application. *Inside Macintosh: Interapplication Communication* describes how your application processes the required Apple events to open or print files.

As described in “Using Finder Information in the Catalog File” beginning on page 7-32, your application typically assigns a file type to a document when it creates one. The file type can be a type especially defined for your application, or it can be one of the existing general types, such as those listed here.

File type	Description
`'APPL'`	Launchable application
`'DFIL'`	File for storing desk accessories
`'DRVR'`	Driver
`'FFIL'`	File for storing fonts
`'INIT'`	System extension
`'PICT'`	QuickDraw picture
`'PRER'`	Printer driver
`'RDEV'`	Chooser extension
`'TEXT'`	Stream of ASCII characters
`'adev'`	Network extension (such as EtherTalk 2.0)
`'appe'`	Background-only application
`'cdev'`	Control panel
`'edtp'`	Edition for sharing graphics-oriented data
`'edts'`	Edition for sharing sound-oriented data
`'edtt'`	Edition for sharing text-oriented data
`'ffil'`	Font
`'ifil'`	Script system resource collection
`'kfil'`	Keyboard layout
`'pref'`	Preferences file
`'qery'`	Query document for database access
`'scri'`	System extension for script systems
`'sfil'`	Sound

File type	Description
`'tfil'`	TrueType font
`'ttro'`	TeachText read-only file
`'zsys'`	A system file (such as the System file itself)

Note

Apple reserves the use of all signatures and file types whose names contain only lowercase and nonalphabetic characters. Your signature and the file types created especially for your application must each contain at least one uppercase character. Since the system software never displays signatures and file types to users, signatures and file types can consist of character combinations that might otherwise be incomprehensible to anyone but you. u

Like signatures, file types must be registered with Apple. Your application must have a file type of `'APPL'`. The creator field of your application file should contain its own signature. Most programming environments provide a simple tool for setting the creator field of your application file.

Your application can create documents of any type, and it can specify any application as the creator. You could write a utility application, for example, that creates a new document by opening one text file and appending onto it another text file. The application would give the new document the same creator as the first original text file so that the Finder can call on that application when the user wants to open or print the new document.

Assign the standard file type `'TEXT'` to files that consist of only text—that is, a stream of characters with return characters at the ends of paragraphs. Most word processors allow the user to create text-only files. A document of file type `'TEXT'` can be opened or printed by any application that accepts such file types. Your application can still assign its own signature as the file's creator so that the Finder can call on it to open or print the file when appropriate.

Users can also open a document created by your application—as well as a document of a file type supported by your application—by selecting its icon and dragging it to your application's icon. Because the document's file type is stored in the catalog file and the Finder stores a list of your application's supported file types in the desktop database, the Finder can determine whether to launch your application. If the document's file type is supported by your application, the Finder launches your application and passes it the name of the document. (These topics are detailed in subsequent sections of this chapter.)

For example, if your application is a page-layout program, it might create documents of its own file type while also supporting documents of `'TEXT'` and `'PICT'` file types. A user can launch your application by dragging a document of any of these file types to your application icon.

Your application also relies on file types to determine which files to let the user open when your application is running. When your application calls the Standard File Package to open a file, your application supplies either a list of the file types that your application can open or a filter function for those types. The open file dialog box then displays only files of the specified types. (See *Inside Macintosh: Files* for details.)

Creating Icons for the Finder

The Finder represents your files as icons. To distinguish your product for the user, you can design your own icons for all the files associated with your application, including

- your application file itself
- standard documents created by your application
- stationery pads that users create from your application's documents
- data-sharing editions that users create from your application's documents
- other special documents, such as read-only, graphics, and query documents, which are either created by your Macintosh application or provided by you for use by other Macintosh applications

For most effective display, you should create an icon family for each of your files. An **icon family** is the set of icons that represent a single object, such as an application or a document, that the Finder displays. An entire icon family consists of large (32-by-32 pixel) and small (16-by-16 pixel) icons, each with a mask, and each available in three different versions of color: black and white, 4 bits of color data per pixel, and 8 bits of color data per pixel. Specifically, the following icons make up the icon family for a single file:

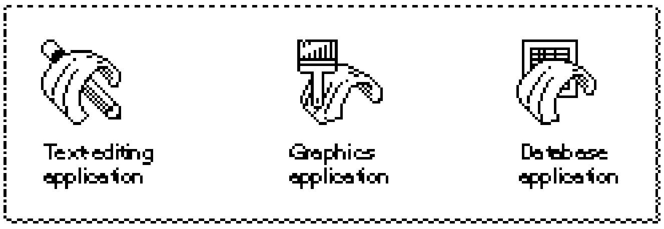
- a large (32-by-32 pixel) black-and-white icon and mask—both of which you define in an icon list (`'ICN#'`) resource
- a small (16-by-16 pixel) black-and-white icon and mask—both of which you define in a small icon list (`'ics#'`) resource
- a large (32-by-32 pixel) color icon with 4 bits of color data per pixel—which you define in a large 4-bit color icon (`'icl4'`) resource
- a small (16-by-16 pixel) color icon with 4 bits of color data per pixel—which you define in a small 4-bit color icon (`'ics4'`) resource
- a large (32-by-32 pixel) color icon with 8 bits of color data per pixel—which you define in a large 8-bit color icon (`'icl8'`) resource
- a small (16-by-16 pixel) color icon with 8 bits of color data per pixel—which you define in a small 8-bit color icon (`'ics8'`) resource

Plate 3 in the front of this book shows how the SurfWriter sample application uses these resources to define the icon family for its application icon.

Somewhat related to these resources are the icon (`'ICON'`) resource and the color icon (`'cicn'`) resource. You can use either to describe a 32-by-32 pixel icon within some element of your application. However, the Finder does *not* use or display any resources that you create of type `'ICON'` or type `'cicn'`. Instead, your application uses these resources to display icons within your application. Generally, you use an icon resource to display a black-and-white icon in a menu or dialog box, as described in the chapters "Menu Manager" and "Dialog Manager" in this book. (For example, the color alert box in Plate 2 in the front of this book specifies a resource of type `'cicn'` for the color icon in the upper-left corner of the alert box.) If you provide a color icon (`'cicn'`) resource with the same resource ID as the icon (`'ICON'`) resource, the Menu Manager and the Dialog Manager display the color icons instead of the black-and-white icons for users with color monitors.

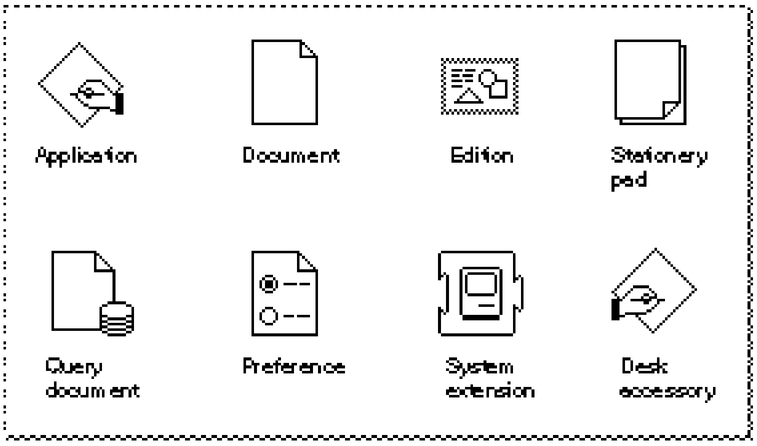
Before creating icon families for your files, you should begin by designing a graphic element that all of your icon families can share and that can help the users quickly identify the files associated with your product. Figure 7-4, for example, illustrates how a company uses the image of a wave in all of its application icons; these icons represent the SurfWriter text-editing application, the SurfPainter graphics application, and the SurfDB database application. As illustrated in Plate 4 at the front of this book, the wave element is also included in icons representing the documents, stationery pads, and editions that users create with these applications.

Figure 7-4 Large black-and-white application icons for a company's product line



If you do not design your own icons, the Finder uses a set of its own default application and document icons for display. Figure 7-5 shows the Finder's default large black-and-white icons.

Figure 7-5 Default large black-and-white icons



Note
Desk accessories, displayed by default with the icon shown in Figure 7-5, were designed for early versions of Macintosh system software that did not support cooperative multitasking. Desk accessories and applications are much more alike in their appearance and behavior in System 7. Because there are no longer any compelling reasons for creating desk accessories, you should generally write a small application instead of a desk accessory if you wish to create a small or simple program. u

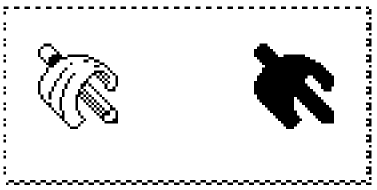
If you don't want the Finder to display the default icons for your application or documents, you must at least define an icon list (`'ICN#'`) resource for each icon.

The term *icon list* has become a bit of a misnomer, because you can define only two images in the icon list resource: a 32-by-32 pixel black-and-white icon and its mask. To define color and 16-by-16 pixel icons for a file, you create additional resources, as described later in this section. (If you don't define color versions of your icons, the Finder displays the black-and-white icon defined in your icon list resource on all displays, and if you don't define 16-by-16 pixel icons, the Finder algorithmically reduces the 32-by-32 pixel icon to half size when needed.)

An icon list resource defines one icon. It contains two icon descriptions: the actual icon for display and an all-black mask that shows the area covered by the icon. The Finder uses the mask to crop the icon's outline into whatever background color or pattern is on the desktop. The Finder then draws the icon into this shape. Therefore, it's important that the mask be exactly the same shape as the icon. The mask also defines the area that users need to click to select the icon. Therefore, it's best not to have any holes in the mask; otherwise, users may have trouble selecting your icon.

Figure 7-6 illustrates a black-and-white icon and its mask for an application. The area around the pencil just underneath the wave creates a problem with this sample icon and its mask: like a hole in a mask, it creates two small areas within the middle of the icon that the user cannot select with the cursor.

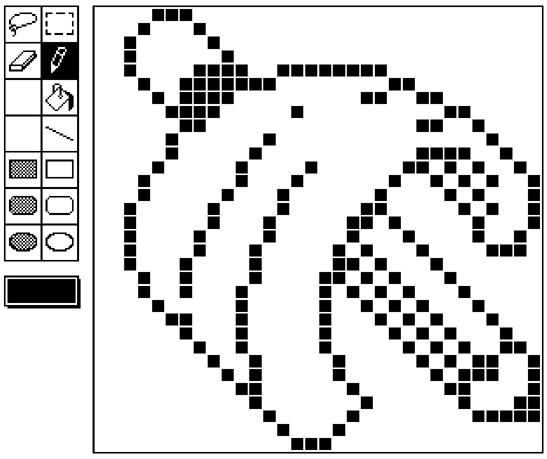
Figure 7-6 A black-and-white icon and its mask for an application



An icon list resource is defined to be an array of two items of type `String[128]`; each bit in the first array represents a pixel in the 32-by-32 pixel icon, and each bit in the second array represents a pixel in the 32-by-32 pixel mask. Typically, you use a high-level tool such as the ResEdit application, which is available through APDA, to create your icon list resources. Figure 7-7 shows how the icon list resource for the icon in Figure 7-6 was created using the ResEdit icon editor. When you are satisfied with the appearance of your icons, you can use the DeRez decompiler to convert your icon list resources into Rez input.

Listing 7-2 is a partial listing of the icon list resource's Rez input that describes the application icon shown in Figure 7-7; Listing 7-2 also shows partial listings for the icon list resources used for the icons that represent the documents created by the application. This listing and those that follow in this chapter use Rez input format to help you understand the format of the resources and see how they work together.

Figure 7-7 The ResEdit view of an icon



Listing 7-2 Rez input for an icon list resource

```
data 'ICN#' (128, purgeable) {      /*application icon & mask*/
                        /*array: 2 elements*/
                        /*[1]: the application icon*/
      $"0E 00 00 00"    /*1st line of icon: 4 bytes (32 bits)*/
      .                 /*32 lines total in icon*/
      .
      .
      ,                 /*[2]: the mask*/
      $"0E 00 00 00     /*1st line of mask: 4 bytes (32 bits)*/
      .                 /*32 lines total in mask*/
      .
      .
};
```

```
data 'ICN#' (129, purgeable) {    /*text document icon and mask*/
                        /*icon data goes here*/
};
data 'ICN#' (130, purgeable) {      /*stationery pad icon & mask*/
                        /*icon data goes here*/
};
data 'ICN#' (131, purgeable) {      /*edition icon & mask*/
                        /*icon data goes here*/
};
```

You can also define a small (16-by-16 pixel) version of your icon in a small icon list resource (that is, in a resource of resource type `'ics#'`). On black-and-white monitors, the Finder displays the small icon in windows when the user chooses by Small Icon from the View menu. On black-and-white monitors, the small icon also appears in the Application menu after the user launches your application and in the Apple menu if the user places your application or an alias to it in the Apple Menu Items folder. (Alias files and the Apple Menu Items folder are described, respectively, in "Using Aliases" beginning on page 7-39 and "Using the System Folder and Its Related Directories" beginning on page 7-41.)

You should also define color versions of both large and small icons by using several resource types. The resource for each icon variation has the same resource ID as the icon list resource that defines the large black-and-white icon. For example, if the resource ID number of your application icon's icon list resource is 128, its small icon list resource should have a resource ID number of 128; and the following resources should also have resource IDs of 128: the large 4-bit color icon resource, the small 4-bit color icon resource, the large 8-bit color icon resource, and the small 8-bit color icon resource.

Don't define masks for the resources that define color icons. The large 4-bit color icon resource and large 8-bit color icon resource use the black-and-white icon mask defined in their companion icon list resource, and the small 4-bit color icon resource and small 8-bit color icon resource use the black-and-white icon mask defined in their companion `'ics#'` resource. Because of this, the outline shapes of your color icons should exactly match those defined in your `'ICN#'` and `'ics#'` resources.

ResEdit 2.1 includes an icon family editor to help you easily manage the creation of these related resources. See the *ResEdit Reference* for details.

See *Macintosh Human Interface Guidelines* for information about the most effective use of color and shape for your icons. It is generally best that you first create the black-and-white icons in the icon list resource and small icon list resource and then add color to them using the resources that define color icons. Don't alter the shapes of your icons among these resources; otherwise, the masks defined in the icon list resource and the small icon list resource won't match these shapes. Choose your colors from the 36 recommended icon colors in the system palette. (If you use ResEdit 2.1, these colors appear in a palette when you choose Apple Icon Colors from the Color menu.) Note that you cannot specify your own color table for these resources.

For more information about color palettes, see *Inside Macintosh: Imaging*. Although the Palette Manager allows you to define a palette for the system to use when it needs to define the color environment, you should rely on the system palette colors for your icons. Users may often use the Finder when your application is not running, and the user can switch to another application when your application is running. Relying on the system palette gives your icons a more consistent look in the Finder regardless of what the active application is. Also, because users can change the desktop color and pattern, your application gives users more control over their work environment if your icons rely on the system palette. Users can always alter your color definitions by selecting an icon and choosing a color from the Label menu. The Finder then blends the chosen color into those of the selected icon. To restore the original colors, users must choose None from the Label menu.

If your application creates documents, it should also define at least two additional icon families: one to be displayed for documents created by your application and another to be displayed when the user creates a stationery pad from one of your application's documents. ("Supporting Stationery Pads" beginning on page 7-34 describes stationery pads.)

If your application creates other variations of its documents, you can assist your users by providing different icons for the different documents. For example, TeachText has separate icon families to distinguish its read-only and graphics documents.

If your application supports data sharing through the Edition Manager, your application should also define an icon family for editions. The Edition Manager (described in *Inside Macintosh: Interapplication Communication*) allows users to share and automatically update data from numerous documents and applications. For example, a user might want to capture sales figures and totals from within a spreadsheet and then include this information in a word-processing document that summarizes sales for a given month. If both the spreadsheet and word-processing applications support the Edition Manager, the user begins by selecting data within the spreadsheet document and creating a publisher. The spreadsheet application then writes a copy of that data to a separate file, called an *edition*. The edition is represented by an icon; by default, it appears as the edition icon shown in Figure 7-5 on page 7-12. If the user opens a word-processing document and creates a subscriber to the spreadsheet document's edition, the word-processing application then incorporates the desired sales figures and totals from the spreadsheet document's edition into the document.

If you design your application to create editions, consider creating an icon that uniquely identifies your editions and that associates them with your application's documents. The file type for your edition containers should be `'edtt'` (for text-oriented data), `'edtp'` (for graphics-oriented data), or `'edts'` (for sound-oriented data); and the creator, of course, should be the signature of your application.

If your Macintosh application is a database program or serves as a source for data (as a spreadsheet program often does), you might wish to create query documents so that other Macintosh applications can gain access to that data through the Data Access Manager; in this case, your application should also define an icon family for its query documents. (See *Inside Macintosh: Communications* for information on sharing data in this manner.)

Plate 4 at the front of this book shows the large color icons for the various documents that the sample SurfWriter application creates: text documents, stationery pads, and editions.

Defining icon resources is not enough to display your icons. In addition, you must follow one of two sets of procedures:

- If you are an application developer, you must define file reference resources and a bundle resource for your application, as described in “Creating File Reference Resources” beginning on page 7-18 and “Creating a Bundle Resource” beginning on page 7-20.
- If you are an information provider or a database developer—that is, if you provide documents that are used by other applications—you don’t need to create file reference resources or a bundle resource to provide document icons on Macintosh computers running System 7. You can instead create customized icons for your documents as described in the following section.

Creating Customized Document Icons

You can create customized icons for your documents. Users can also create customized icons. When an icon list resource is stored with a resource ID of –16455 in the resource fork of a file, the Finder uses the large, small, 4-bit and 8-bit color, and black-and-white icons defined in resources with that resource ID as **customized icons** in place of the Finder’s default icon and in place of any icons listed in the file’s bundle resource.

Note

Although an application can assign icons to it all of its documents by associating their icons with the documents’ file types in a bundle resource (as explained in “Creating File Reference Resources” beginning on page 7-18 and “Creating a Bundle Resource” beginning on page 7-20), a customized icon can represent only one specific file—that file that has an icon list resource with a resource ID of –16455 in its resource fork. u

Users of System 7 are able to customize individual icons. By selecting a file and choosing Get Info from the File menu, the user sees the information window for that file. The user can then select the icon displayed in the upper-left corner of the information window and use the Paste command in the Edit menu to replace it with a picture from the Clipboard. The Finder creates a family of icons based on the user’s customized icon, assigns a resource ID number of –16455 to each resource in the icon family, stores these resources in the resource fork of the file that the icon represents, and sets the `hasCustomIcon` bit in the file’s Finder flags field. (Finder flags are described in detail in “File Information Record” beginning on page 7-47.)

Your application can use the same strategy to provide customized icons for the documents that it creates. For example, a drawing application might create miniature versions of the illustrations contained within its documents and use those for the documents’ icons.

If you are a database developer who creates and distributes query documents that support the Data Access Manager, you can also use this strategy to create icons that identify your database’s query documents. Similarly, if instead of producing an application you produce and distribute information (such as database files, stationery pads, clip art libraries, or dictionaries) to be used by other applications, you might want to provide icons that distinguish your documents.

To make the Finder display customized icons for a document, you must create—at least—an icon list resource with resource ID –16455 and store it in the document's resource fork. (To create this while your application is running, your application can call the `AddResource` procedure, described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.) You can use the following constant in place of the ID number:

```
CONST kCustomIconResource = –16455; {res ID for custom icon}
```

If you provide only an icon list resource, the Finder uses a black-and-white icon on all screen displays and automatically reduces it when a small version of the icon is required. To create color versions and to define a small version of the icon, create an entire icon family as described in "Creating Icons for the Finder" beginning on page 7-11.

After creating resources for icons using the `kCustomIconResource` constant as their IDs, you must set the `hasCustomIcon` bit in the file's Finder flags field. To prevent users from changing these icons, set the `nameLocked` bit in the file's Finder flags field. (Most development environments provide tools for setting these bits. "Using Finder Information in the Catalog File" beginning on page 7-32 describes how to determine and set these Finder flags.)

Creating File Reference Resources

File reference (`'FREF'`) resources perform two main functions. First, they associate icons you define with file types used by your application. Second, they allow users to drag document icons to your application icon in order to open them from your application.

Create a file reference resource for your application file itself and create separate file reference resources for each file type that your application can open. Listing 7-3 shows, in Rez input format, the file reference resources for the SurfWriter application file, text documents, stationery pads, and editions and for TeachText read-only documents.

Each file reference resource specifies the following items:

- a file type
- the local ID of an icon list resource as assigned in the bundle resource
- an empty string

The file type can be defined for files created by your application only, for files created by other applications that your application supports, or for files of the existing general types, such as `'TEXT'` and `'PICT'`.

As described in the next section, "Creating a Bundle Resource," the local ID maps the file type to an icon list resource that is assigned the same local ID in the bundle resource. If you wanted two file types to share the same icon, for example, you could create two separate file reference resources that share the same local ID, which the bundle resource would map to the same icon list resource. (Creating two file types that share the same icon is not recommended, however, because a shared icon would make it very difficult for the user to distinguish between the different file types while using the Finder.)

Listing 7-3 Rez input for file reference resources

```
resource 'FREF' (208, purgeable) {  /*SurfWriter application*/
   'APPL',  /*type 'APPL'*/
   0,       /*maps to icon list resource w/ local ID 0 in bundle resource*/
   ""       /*leave empty string for name: not implemented*/
};
resource 'FREF' (209, purgeable) {  /*SurfWriter document*/
   'TEXT',  /*type 'TEXT'*/
    1,      /*maps to icon list resource w/ local ID 1 in bundle resource*/
   ""
};
resource 'FREF' (210, purgeable) {  /*SurfWriter stationery pad*/
   'sEXT',  /*type 'sEXT'*/
   2,       /*maps to icon list resource w/ local ID 2 in bundle resource*/
   ""
};
resource 'FREF' (211, purgeable) {  /*SurfWriter edition*/
   'edtt',  /*type 'edtt'*/
   3,       /*maps to icon list resource w/ local ID 3 in bundle resource*/
   ""
};
resource 'FREF' (212, purgeable) {/*TeachText read-only files*/
   'ttro', 4, ""  /*These documents have TeachText as their */
                  /* creator. Finder uses TeachText's icon list resource */
                  /* for these documents. Included here so users */
                  /* can drag these docs to SurfWriter's app icon*/
};
```

If you provide your own icon for the stationery pads that users create from your application's documents, create a file reference resource for your stationery pads. Assign this file reference resource a file type in the following manner: use the file type of the document upon which the stationery pad is based, but replace the first letter of the original document's file type with a lowercase *s*. As with other file reference resources, you map this to an icon list resource in the bundle resource. (This convention necessitates that you make the names of your documents' file types unique in their last three letters.)

For example, in Listing 7-3, the `'sEXT'` file type assigned within the file reference resource is used for stationery pads created from documents of the `'TEXT'` file type. In this case, when the `isStationery` bit (described in "Using Finder Information in the Catalog File" beginning on page 7-32) is set on a document of file type `'TEXT'`, the Finder looks in the SurfWriter application's bundle (`'BNDL'`) resource to determine what icon is mapped to documents of type `'sEXT'`. The Finder then displays the document using the stationery pad icon shown in Plate 4 at the front of this book.

When the user drags a document icon to your application icon, the Finder checks a list that it maintains of your file reference resources. If the document's file type appears in this list, the Finder launches your application with a request to open that document.

If your application supports file types for which it doesn't provide icons, you can still define file reference resources for them, and then users can launch your application by dragging these document icons to your application icon. For example, the file reference resource with resource ID 212 in Listing 7-3 on page 7-19 is created so that the Finder launches the SurfWriter application when users drag TeachText read-only documents to the SurfWriter application icon. Since these documents have TeachText as their creator, the Finder displays the icon that the TeachText application defines for them in its own bundle resource.

By supporting the Open Documents event, you can also specify disks, folders, and a pair of wildcard file types in your file reference resources so that users can launch your application by dragging their icons to your application icon. As explained in *Inside Macintosh: Interapplication Communication*, the Open Documents event is one of the four required Apple events. After the Finder uses the Process Manager to launch an application that supports high-level events, the Finder sends your application an Open Documents event, which includes a list of alias records for objects that the application should open.

Because alias records can specify volumes and directories as well as files, an Open Documents event gives you the opportunity to handle cases in which users drag disk or folder icons to your application. (Alias records are described in "Using Aliases" beginning on page 7-39.) Create a file reference resource and specify `'disk'` as the file type to allow users to drag hard disk and floppy disk icons to your application icon. Create a file reference resource and specify `'fold'` as the file type to allow users to drag folder icons to your application icon.

You can create a file reference resource that specifies `'****'` as the file type to allow users to drag all file types—including applications, system extensions, documents, and so on, but not including disks or folders—to your application icon. If you create three file reference resources that specify `'disk'`, `'fold'`, and `'****'` as their file types and if your application supports the Open Documents event, you effectively allow users to launch your application by dragging any icon to your application icon. It is up to your application to open disks, folders, or all possible file types in a manner appropriate to the needs of the user.

Creating a Bundle Resource

A bundle (`'BNDL'`) resource associates all of the resources used by the Finder for your application; in particular, it associates your application and its documents with their icons. The bundle resource contains

- the application's signature
- the resource ID number of its signature resource (which should always be 0)
- the assignment of local IDs to the resource IDs of all icon list resources defined for the application; the local IDs must be the same as those assigned within corresponding file reference resources

- the assignment, for compatibility reasons, of local IDs to file reference resource IDs (For consistency, these can be the same local IDs that are assigned inside the file reference resources, but they don't have to be—they only need to be unique for every file reference resource.)

When the Finder first displays your application on the user's desktop, it checks the catalog file (described in detail in "Using Finder Information in the Catalog File" beginning on page 7-32) to see if your application has a bundle resource. If it doesn't, the Finder displays the default icons shown in Figure 7-5 on page 7-12. If your application has a bundle resource, the Finder installs the information from the bundle resource and all its bundled resources into either the desktop database for a hard disk or into the Desktop file for a floppy disk and uses this information to display icons for the file types associated with your application.

You must assign local IDs to your icon list resources within your bundle resource. Make sure that for all your file types with icons, these local IDs match the local IDs you assigned inside their corresponding file reference resources. In the Desktop file on floppy disks (and on hard disks running earlier versions of system software), the Finder renumbers the resource IDs that you've assigned to your resources to avoid conflicts with the resources of other applications. Therefore, the bundle resource has to rely on these local IDs to map icon list resources to their file reference resources; that is, the bundle resource uses the local ID you assign to an icon list resource to map it to the file reference resource that has specified the same local ID.

For example, the file reference resource with resource ID 208 in Listing 7-3 on page 7-19 shows that the file type `'APPL'` (the SurfWriter application file) is assigned a local ID of 0. In the bundle resource shown in Listing 7-4, you see that local ID 0 is assigned to the icon list resource with resource ID 128. This maps the icon defined by this resource (see Figure 7-7 on page 7-14) to the SurfWriter application file. Listing 7-4 shows the bundle resource for the icons and file reference resources defined in Listing 7-2 on page 7-14 and in Listing 7-3 on page 7-19.

Listing 7-4 Rez input for a bundle resource

```
resource 'BNDL' (128, purgeable) {  /*SurfWriter bundle resource*/
   'WAVE',     /*SurfWriter signature*/
   0,          /*resource ID of signature resource: should be 0*/
   {
      'ICN#', {   /*mapping local IDs in 'FREF's to 'ICN#' IDs*/
      0, 128,  /*'FREF' w/ local ID 0 maps to 'ICN#' res ID 128*/
      1, 129,  /*'FREF' w/ local ID 1 maps to 'ICN#' res ID 129*/
      2, 130,  /*'FREF' w/ local ID 2 maps to 'ICN#' res ID 130*/
      3, 131   /*'FREF' w/ local ID 3 maps to 'ICN#' res ID 131*/
               /*no 'FREF' with local ID 4 in this list: */
               /* TeachText's icons used for 'ttro' file type*/
      },
      'FREF', {   /*local res IDs for 'FREF's: no duplicates*/
      10, 208, /*local ID 10 assigned to 'FREF' res ID 208*/
      11, 209, /*local ID 11 assigned to 'FREF' res ID 209*/
      12, 210, /*local ID 12 assigned to 'FREF' res ID 210*/
      13, 211, /*local ID 13 assigned to 'FREF' res ID 211*/
      14, 212  /*local ID 14 assigned to 'FREF' res ID 212*/
      }
   }
};
```

In Listing 7-4, notice that you also assign local IDs to file reference resources inside the bundle resource. This assignment is superfluous because the Finder doesn't map these local IDs to any other resources. The local ID assignment for file reference resources inside the bundle resource was implemented for the earliest versions of Macintosh system software, and it remains this way today to maintain backward compatibility. For compatibility with the format of the bundle resource, assign local IDs to file reference resource IDs. You may number them any way you like, except that each local ID in this particular list must be unique.

Of all the icon resource types that make up an icon family, you need to list only the icon list resource in the bundle resource. The Finder automatically recognizes and loads all the other members of the icon family—provided that you have given them the same resource IDs that you have assigned to your icon list resource.

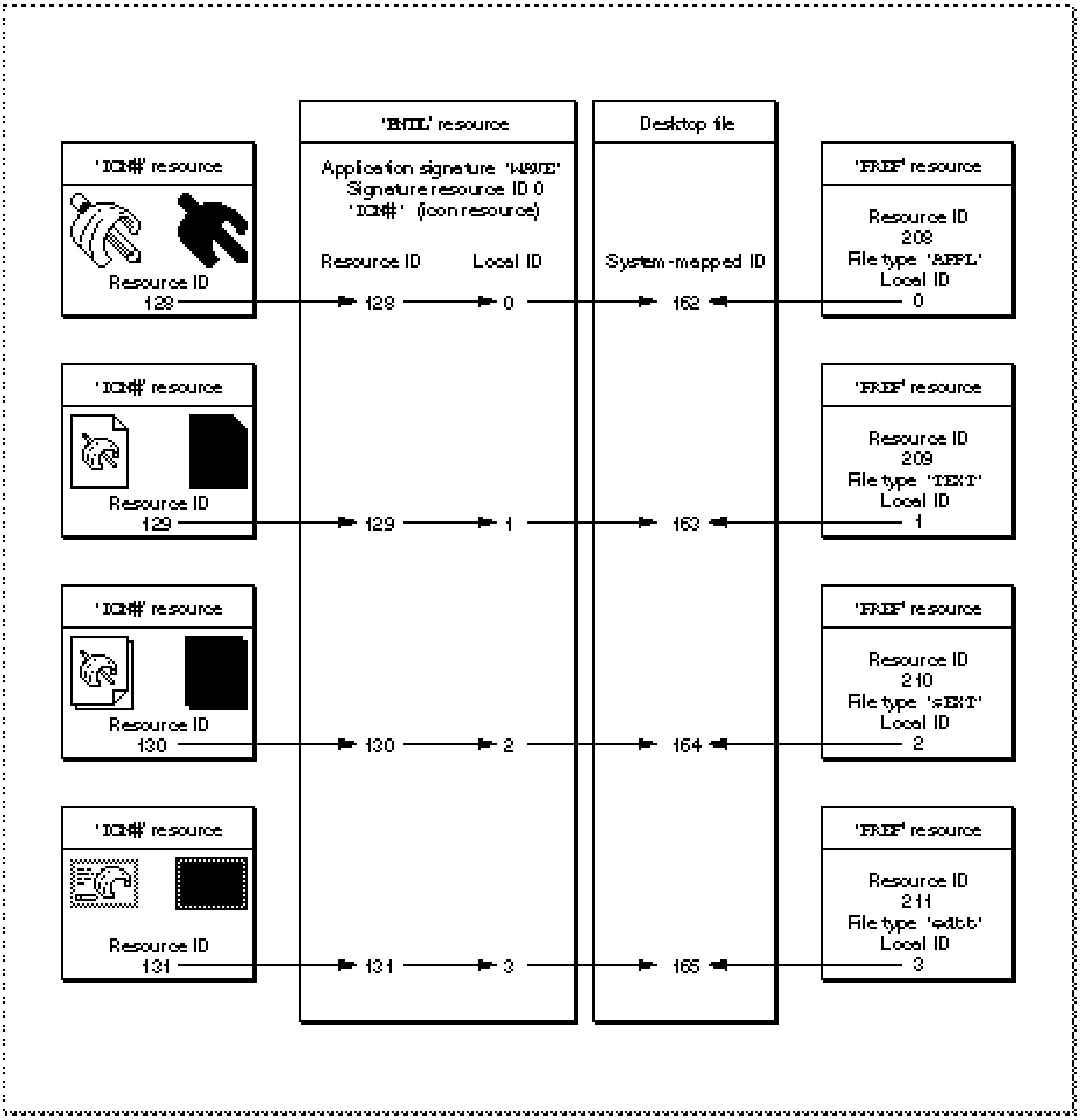
If the user drags documents created by other applications to your application icon, and if you have created file reference resources for these documents' file types, the Finder launches your application and passes it the names of the documents. You should create file reference resources for all file types that your application supports. Do not provide icon resources for file types created by other applications because the Finder won't use them, but will instead use the icon resources defined by the documents' creators. Though the local IDs of such a file reference resource are superfluous in the file reference resource and at the bottom of the bundle resource, the resource formats require that you provide local IDs in both.

For example, notice in Listing 7-3 on page 7-19 that the file reference resource with resource ID 212 is assigned a local ID of 4, but that no icon list resource is assigned to local ID 4 in the bundle resource in Listing 7-4 on page 7-21. This file reference resource, which specifies a file type of `'ttro'`, was created in Listing 7-3 to make the Finder launch the SurfWriter application when users drag TeachText read-only documents to the SurfWriter application icon. No icon mapping is made for this file type in the SurfWriter application's bundle resource because the Finder displays the icons defined for it by the TeachText application. The file reference resource with resource ID 212 is assigned to local ID 14 in the bundle resource in Listing 7-4 because the format of the resource requires a local ID for all associated file reference resources.

You alert the Finder that your application has a bundle resource by setting a bit in the file's Finder flags field. (Most development environments provide a simple tool for setting the bundle bit. "Using Finder Information in the Catalog File" beginning on page 7-32 describes Finder flags.)

Figure 7-8 illustrates how the bundle resource created in Listing 7-4 uses local IDs to map icon list resources to file reference resources. This figure illustrates two main concepts: first, that one bundle resource ties together all the icon resources and file reference resources for your application and all of its documents; and second, that the icon resources and their associated file reference resources are mapped together by local IDs.

Figure 7-8 Linking icon list resources and file reference resources in a bundle resource



In Figure 7-8, the application file's icon list resource has resource ID 128 while its file reference resource has resource ID 208. For easier code maintenance, you should probably assign the same resource ID to a file's file reference resource that you assign to its icon list resource. However, because the Finder renumbers these whenever it adds them to a Desktop file on floppy disks, you must map them by using local IDs. In Figure 7-8, the application file's icon list resource is assigned local ID 0. This maps the icon to the file type described by the file reference resource with local ID 0—in this case, the file reference resource with resource ID 208.

The general steps you must take to provide icons for applications and documents are enumerated here and assume that you are using a tool, such as ResEdit, that allows you to open and edit several resources simultaneously. (Remember that these resources must have resource IDs of 128 or greater.)

To provide your application with icon families for itself and for its documents, follow these steps:

1. Design a graphic element that all of your icon families can share in common and that can help users quickly identify the files associated with your product.
2. Create an icon list (`'ICN#'`) resource for your application file.
3. Create the other members of the icon family of the application file—resources of types `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`—and give each of these the same resource ID as the icon list resource.
4. Create a bundle (`'BNDL'`) resource.
5. Within the bundle resource, list the resource ID number of the application file's icon list resource and assign it a local ID of 0.
6. Create a file reference resource for the application file.
7. Within the file reference resource, assign the application a file type of `'APPL'` and assign it a local ID of 0.
8. Within the bundle resource, list the resource ID number of the file reference resource for the application file and assign it a unique local ID—for example, 0 to maintain consistency with the local ID assigned in the file reference resource.
9. Create another icon family—consisting of resources of types `'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`—to represent one type of document that your application creates.
10. Within the application's bundle resource, list the resource ID number of the document's icon list resource and assign it a local ID of 1.
11. Create a file reference resource for the document.
12. Within the file reference resource for the document, assign it a file type (for example, `'TEXT'` or `'edtt'`) and assign it a local ID of 1.
13. Within the bundle resource, list the resource ID number of the file reference resource for the document and assign it a unique local ID—for example, 1 to maintain consistency with the local ID assigned in the file reference resource.
14. Assigning unique local IDs for every type of document your application creates, repeat steps 9 through 13.

15. If your application supports file types of other applications, define file reference resources for them, but do not create icon resources for them.
16. Create a signature resource (as described in “Giving a Signature to Your Application and a Creator and a File Type to Your Documents” beginning on page 7-8) with resource ID 0.
17. Set the file’s `hasBundle` bit and clear the `hasBeenInited` bit in the file’s Finder flags. (Finder flags are described in “Using Finder Information in the Catalog File” beginning on page 7-32.)
18. Save and close all of the resources. (When you restart your Macintosh computer, your application should appear with its own icon. If you later alter any of your icons, clear the `hasBeenInited` bit and rebuild your desktop database by pressing Command-Option when restarting.)

How and When the Finder Launches Your Application

The previous sections in this chapter explain the resources that the Finder uses to display and launch your application. This section provides a brief summary of how the Finder—using the previously described resources—starts up your application whenever the user requests the Finder to launch your application or to open or print a document supported by your application.

The simplest scenarios under which the Finder launches your application occur when the user double-clicks your application icon or selects it and chooses Open from the Finder’s File menu. In these cases, the Finder calls the Process Manager to start your application. As explained in *Inside Macintosh: Processes*, the Process Manager creates a partition of memory for your application, loads your code into this partition, and sets up the stack, heap, and A5 world for your application. The Process Manager returns control to the Finder.

If your application supports the required Apple events (as explained in *Inside Macintosh: Interapplication Communication*), the Finder sends your application an Open Application event and then relinquishes control to your application. Your application then performs the tasks necessary to open itself—displaying an untitled document window, for example.

When the user requests the Finder to open or print a document supported by your application, the Finder calls the Process Manager and launches your application in the same way, except that the Finder sets up the information your application needs to open or print the document and passes this information to your application. This information includes a list of files to open or print. In System 7, applications receive this information through Apple events, which are described in *Inside Macintosh: Interapplication Communication*.

The user can request the Finder to open documents created by your application by double-clicking one of their icons, and the user can request the Finder to open or print documents by selecting one or more icons and choosing Open or Print from the Finder’s File menu. The Finder reads the creator field of each selected file to find the document’s creator. Typically (as described in “Using Finder Information in the Catalog File” beginning on page 7-32), your application sets the four-character string specified in its signature resource as the creator of its documents. The Finder searches for the application whose signature matches each document's creator. If the document's creator matches your application's signature, the Finder calls the Process Manager, launches your application, and then passes your application the name of the selected document or selected multiple documents in an Open Documents or a Print Documents event. Your application should then open the documents in titled windows or print them, as appropriate. (See *Inside Macintosh: Files* for detailed information about opening documents; see *Inside Macintosh: Imaging* for detailed information about printing them.)

If the user tries to open documents created by your application and your application is missing, the Finder displays an alert box telling the user that your application is missing. The Finder displays the name of your application in this alert box if you provide your documents with a missing-application name string resource, as described in "Displaying Messages When the Finder Can't Find Your Application" beginning on page 7-27.

Sometimes when your application is already running, the user might double-click a document created by your application. In this case, the Finder sends your application the Open Documents event.

The user can also request the Finder to launch your application by dragging one icon or several icons to your application's icon. The Finder determines whether to launch your application by comparing the document's file type (which is stored in the catalog file) against the list of your application's supported file types. The Finder compiles this list from the file reference resources you create for your application; the Finder stores this list in the desktop database. If the document's file type appears in the file reference resource list for your application, the Finder calls the Process Manager, launches your application, and passes it the name of the selected document or selected multiple documents in an Open Documents event. Your application should then open the documents in titled windows.

You can also specify disks, folders, and a wildcard file type for all other files in your file reference resources so that users can launch your application by dragging their icons to your application icon, in which case the Finder launches your application and sends it an Open Documents event. An Open Documents event includes a list of alias records for objects that the application should open. It is up to your application to open disks, folders, or all possible file types in a manner appropriate to the needs of the user. (Alias records are described in "Using Aliases" beginning on page 7-39.)

To support stationery, your application should specify the `isStationeryAware` constant in its `'SIZE'` resource and always check the `isStationery` bit of a document passed to it by the Finder. If the `isStationery` bit is set for a file that the user wants to open, your application should copy the stationery pad's contents into a new document and open the document in an untitled window. This is described in "Supporting Stationery Pads" beginning on page 7-34.

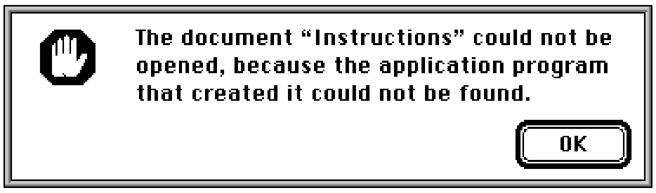
In System 7, users can create aliases, which are objects that represent other files, directories, or volumes. If the user opens an alias that represents a document created by your application, the Finder resolves the alias for you; that is, it passes your application the name and location of the document itself, not the alias.

Displaying Messages When the Finder Can’t Find Your Application

When the user double-clicks a file or selects it and chooses either the Open or the Print command from the Finder’s File menu, the Finder looks for the application whose signature is stored in the file’s creator field. The Finder starts up that application and tells it which documents the user wants to open or print. If the Finder cannot find the creator application, it displays an alert box.

If the document is of file type `'TEXT'` or `'PICT'` and if the TeachText application is available, an alert box asks the user whether the TeachText application should be used to open the document. For documents of any other file type, or if the TeachText application is not present, the Finder displays an alert box like the one shown in Figure 7-9. Your application should store one of two string resources in its documents to make the alert box message more useful than the default shown in Figure 7-9.

Figure 7-9 The default application-unavailable alert box



The document “Instructions” could not be opened, because the application program that created it could not be found.

Before displaying the default message shown in Figure 7-9, the Finder looks in the document for one of two special `'STR '` resources with resource ID numbers of –16396 and –16397: the missing-application name string and the application-missing message string, respectively. If the Finder can’t find the document’s creator on any mounted volume, it looks first for the application-missing message string resource. Provide an application-missing message string resource if you do not intend for users to open the file. The message should explain why the file can’t be opened. If the Finder does not find an application-missing message string resource, it looks for the missing-application name string resource. Provide a missing-application name string resource if you intend for users to open the file. The missing-application name string should be your application’s name; the Finder displays it in an alert box to inform the user that your application is needed.

Supply either the application-missing message string resource or the missing-application name string resource; don’t supply both. Supply an application-missing message string resource for documents (such as a preferences file) that your application uses but that users should not open; supply a missing-application name string resource for documents that you intend for users to open with your application.

Your missing-application name string resource (an `'STR '` resource with a resource ID number of –16396) should contain the name of your application. Listing 7-5 on the next page shows a missing-application name string resource for the SurfWriter application.

Listing 7-5 Rez input for a missing-application name string resource

```
resource 'STR ' (-16396, purgeable) {  /*the application name*/
   "SurfWriter"
};
```

You can store this resource in the resource fork of your application. When your application saves a document for the first time, it should copy the missing-application name string resource from your application's resource fork to the resource fork of the newly created document. Listing 7-6 shows a fragment of an application-defined function called `DoSaveAsCmd`, which the application calls when the user chooses the Save As command from the File menu. (For a description of the File Manager routines used here to create, open, and save the resource file, see *Inside Macintosh: Files*.)

Listing 7-6 Storing a missing-application name string resource in the resource fork of a document

```
VAR
   myData:  MyDocRecHnd;   {handle to document record}
   myErr:   OSErr;
   myFile:  Integer;       {file reference number}

{with the DoSaveAsCmd routine: create document's resource fork}
FSpCreateResFile(myData^^.fileFSSpec, 'MYAP', 'TEXT',
                  smSystemScript);
myErr := ResError;
IF myErr = noErr THEN   {open the resource fork}
   myFile := FSpOpenResFile(myData^^.fileFSSpec, fsRdWrPerm);
IF myFile > 0 THEN   {copy the missing-application name string}
   myErr := DoCopyResource('STR ', -16396, gAppsResFile, myFile)
ELSE
   myErr := ResError;
IF myErr = noErr THEN
   myErr := FSClose(myFile);  {close the resource fork}
```

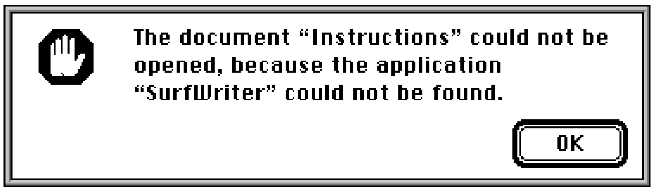
Listing 7-7 shows the application-defined function `DoCopyResource`, which copies the missing-application name string resource from the application's resource fork into the newly created document's resource fork. (For a description of the Resource Manager routines used here to set, open, and write the resource file, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.)

Listing 7-7 Copying the missing-application name string resource into the resource fork of a document

```
FUNCTION DoCopyResource (theType: ResType; theID: Integer;
                         source: Integer; dest: Integer): OSErr;
VAR
   myHandle:   Handle;  {handle to resource to copy}
   myName:     Str255;  {name of resource to copy}
   myType:     ResType; {ignored; used for GetResInfo}
   myID:       Integer; {ignored; used for GetResInfo}

BEGIN
   UseResFile(source);  {set the source resource file}
   myHandle := GetResource(theType, theID);  {open the source}
   IF myHandle <> NIL THEN
   BEGIN
      GetResInfo(myHandle, myID, myType, myName);  {get resource }
                                                   { name}
      DetachResource(myHandle);  {detach resource}
      UseResFile(dest);    {set the destination resource file}
      AddResource(myHandle, theType, theID, myName);
      IF ResError = noErr THEN
         WriteResource(myHandle);   {write resource data}
   END;
   DoCopyResource := ResError;      {return result code}
END;
```

If a user tries to open or print one of the application's documents when the application is not present, the Finder specifies the application's name in the alert box, as illustrated in Figure 7-10.

Figure 7-10 The application-unavailable alert box specifying an application's name


The document "Instructions" could not be opened, because the application "SurfWriter" could not be found.

Your application-missing message string resource (an 'STR ' resource with a resource ID number of –16397) should explain why the user cannot open or print a document. Use this resource for files—such as your application's preferences file—that are not intended to be opened or printed by the user. Register a signature (as explained in "Giving a Signature to Your Application and a Creator and a File Type to Your Documents" beginning on page 7-8) that is different from the signature of your application and set this signature as the creator of files that you don't want your users to open. This ensures that the Finder displays your message instead of launching your application when the user double-clicks these documents.

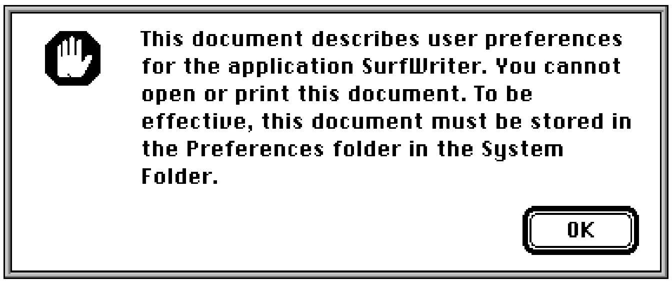
Listing 7-8 illustrates an application-missing string resource that explains why a user cannot open a preferences file.

Listing 7-8 Rez input for an application-missing message string resource

```
resource 'STR ' (-16397, purgeable) {/*the message*/
   "This document describes user preferences for the application "
"SurfWriter. You cannot open or print this document. To be "
"effective, this document must be stored in the Preferences "
"folder in the System Folder."
};
```

Figure 7-11 shows the alert box generated by Listing 7-8.

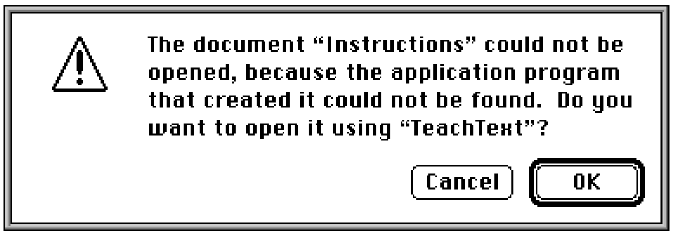
Figure 7-11 The application-unavailable alert box with a customized message



This document describes user preferences for the application SurfWriter. You cannot open or print this document. To be effective, this document must be stored in the Preferences folder in the System Folder.

Note that if your application creates documents of file type `'TEXT'` or `'PICT'`, if the TeachText application is available, and if your application is missing when the user tries to open these documents from the Finder, the Finder always displays the alert box shown in Figure 7-12. For these file types, the Finder displays this alert box even if you provide missing-application name string resource or application-missing message string resource.

Figure 7-12 The application-unavailable alert box for `'TEXT'` and `'PICT'` documents



The document “Instructions” could not be opened, because the application program that created it could not be found. Do you want to open it using “TeachText”?

Providing Version Resources

You can use version (`'vers'`) resources to record version information for your application. If the user opens the Views control panel, clicks the Show version box, and then chooses any command from the View menu other than by Icon or by Small Icon, filenames and their version numbers from the version resource appear in the active Finder window. The Finder also displays version information when the user selects your application and chooses Get Info from the File menu.

The version resource allows you to store a version number, a version message, and a region code. (Because the Get Info command's information window already displays the name of your application, the version message should not include the name of your application.) You can use version resources to assign version information to an individual file and, if it is a part of a larger collection of files, to the entire superset of files. The version resource with a resource ID number of 1 specifies the version of the file; the version resource with a resource ID number of 2 specifies the version of the set of files.

Each version resource should contain these elements:

- Major revision level in binary-coded decimal format. Although the Finder doesn't display it anywhere, you can store this information here; most programming environments provide a tool for setting this element.
- Minor revision level in binary-coded decimal format. Although the Finder doesn't display it anywhere, you can store this information here; most programming environments provide a tool for setting this element.
- Development stage. You can use any of these values or the constants that represent them:

Value	Constant	Description
0x20	`development`	Prealpha file
0x40	`alpha`	Alpha file
0x60	`beta`	Beta file
0x80	`release`	Released file

- Prerelease revision level. This number specifies the version if the software is still prerelease.
- Region code. This identifies the script system for which this version of the software is intended. See the chapter "Script Manager" in *Inside Macintosh: Text* for information about the values represented by the various region codes that can be specified here.
- Version number. This string identifies the version number of the software. When the user opens the Views control panel, clicks the Show version box, and then chooses any command from the View menu other than by Icon or by Small Icon, the Finder window containing this application displays this string.

- Version message. This string identifies the version number and either a company copyright for a file or a product name for a superset of files. When the user selects this file and chooses the Get Info command, the Finder displays this string in the information window as follows:
    - For a version resource with a resource ID number of 1, this string is displayed in the version field of the information window.
    - For a version resource with a resource ID number of 2, this string is displayed beneath the file's name next to the file's icon at the top of the information window.

Listing 7-9 illustrates the version resources for a graphics application and for the document-processing system of which it is a part. Notice that the paint program is version 1.0 while the set of files that compose the entire document-processing system is version 2.0.

Listing 7-9 Rez input for a pair of version resources

```
resource 'vers' (1, purgeable) {
   0x01, 0x00, release, 0x00, verUS,
   "1.0",
   "1.0 (US), © My Company, Inc. 1992"
};
resource 'vers' (2, purgeable) {
   0x02, 0x00, release, 0x00, verUS,
   "2.0",
   "(for SurfWriter 3.0)"
};
```

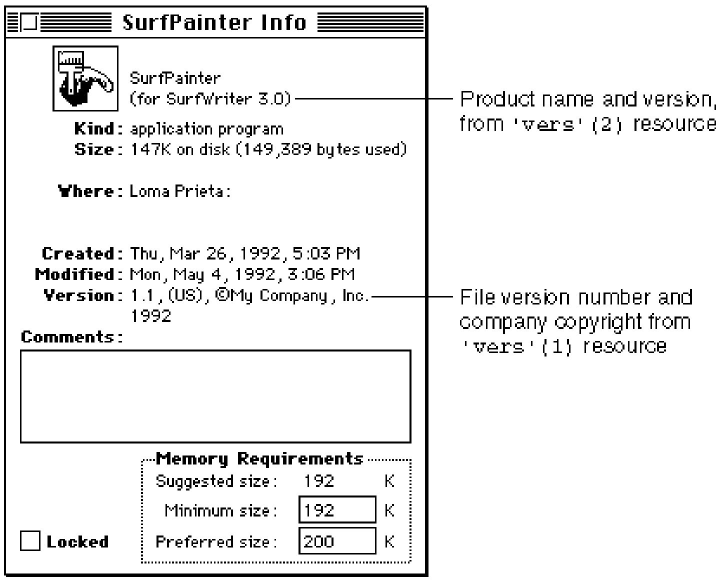
Figure 7-13 illustrates how the Finder displays the information from these resources in its information window.

You can store version resources in any kind of file, not just an application. If your application does not contain a version resource with a resource ID number of 1, the Finder displays the string from your signature resource as the version information in the information window for your application.

Using Finder Information in the Catalog File

A catalog file exists on every volume to maintain relationships between the files and directories on that volume. (A volume is any storage medium formatted to contain files.) Although it's used mostly by the File Manager, the catalog file also contains information used by the Finder. The information for files is listed in file information records (data structures of type `FInfo`) and in extended file information records (data structures of type `FXInfo`). The information for directories is listed in directory information (`DInfo`) records and in extended (`DXInfo`) directory information records.

Figure 7-13 The version data in the information window



SurfPainter Info

SurfPainter
(for SurfWriter 3.0)

Kind: application program
Size: 147K on disk (149,389 bytes used)

Where: Loma Prieta:

Created: Thu, Mar 26, 1992, 5:03 PM
Modified: Mon, May 4, 1992, 3:06 PM
Version: 1.1, (US), ©My Company, Inc. 1992
Comments:

Memory Requirements

Suggested size:	192	K
Minimum size:	192	K
Preferred size:	200	K

☐ **Locked**

Product name and version, from 'vers' (2) resource
File version number and company copyright from 'vers' (1) resource

The Finder manipulates the fields in the file information, directory information, and extended directory information records; your application shouldn't have to directly check or set any of these fields.

Normally, your application sets the file type and the creator information in fields of the file's file information record when your application creates a new file; for example, the File Manager function `FSpCreate` (described in *Inside Macintosh: Files*) takes a creator and a file type as parameters. The Finder manipulates the other fields in the file information record, which is shown here:

```
TYPE  FInfo =
      RECORD
         fdType:     OSType;     {file type}
         fdCreator:  OSType;     {file creator}
         fdFlags:    Integer;    {Finder flags}
         fdLocation: Point;      {file's location in window}
         fdFldr:     Integer;    {directory that contains file}
      END;
```

After you have created a file, you can use the File Manager function `FSpGetFInfo` to return the file information record, then change the `fdType` and `fdCreator` fields by using the File Manager function `FSpSetFInfo`.

You can check the information in this record by calling the File Manager function `FSpGetFInfo` or `PBGetCatInfo`. In particular, you may want to check the file type or creator for a file, or you may want to check or set one of your document's Finder flags. See "File Information Record" beginning on page 7-47 for a list of all the Finder flags. The only Finder flags you might ever want to set are described here:

- `isInvisible`. This flag specifies that a file is invisible from the Finder and from the Standard File Package dialog boxes. Making a file invisible is generally not recommended. Not even temporary files need to be invisible because the Temporary Items folder into which they should be written is invisible. The Temporary Items folder is described in "Using the System Folder and Its Related Directories" beginning on page 7-41.
- `hasBundle`. This flag specifies that a file has a bundle resource that associates the file with your own icons. When the Finder displays or manipulates a file, it checks the file's `hasBundle` bit (also called the **bundle bit**). If that bit is not set, the Finder displays a default icon for that file type. If the `hasBundle` bit is set, the Finder checks the `hasBeenInited` bit. If the `hasBeenInited` bit is set, the Finder uses the information in the desktop database to display that file's icon. If the `hasBeenInited` bit is not set, the Finder installs the information from the bundle resource in the desktop database and sets the `hasBeenInited` bit. Most development environments provide a simple tool for setting the bundle bit when you create your application.
- `nameLocked`. This flag specifies that a file cannot be renamed from the Finder and that the file cannot have customized icons assigned to it by users.
- `isStationery`. This flag specifies that a file is a stationery pad. To support stationery pads, your application should check this bit for every document passed to it by either the Finder or the Standard File Package. (The File Manager functions `StandardGetFile` and `CustomGetFile` return this flag in the `sfFlags` field of the standard file reply record.) If the `isStationery` bit is set for a file that a user wants to open, your application should copy the template's contents into a new document and open the document in an untitled window. Stationery pads are described in the next section.
- `isShared`. This flag specifies that a file is an application that multiple users on a network can execute simultaneously.
- `hasCustomIcon`. This flag specifies that a file has a customized icon. "Creating Customized Document Icons" beginning on page 7-17 explains how users or your application can use customized icons.

Supporting Stationery Pads

Stationery pads are special documents that the user creates as templates. Opening a stationery pad should not open the document itself; instead, it should open a new document with the same contents as the stationery pad. To turn any document into a stationery pad, the user selects it, chooses Get Info from the File menu, and clicks the Stationery pad checkbox in the information window. The Finder tags a document as being a stationery pad by setting the `isStationery` bit in the file's Finder flags field.

When the user opens a stationery pad from the Finder, the Finder first checks your application's size resource to see if your application supports stationery. The 'SIZE' resource tells the Finder and the Process Manager which features your application supports and how much memory to allocate when it starts up your application. Listing 7-10 illustrates a size resource.

Listing 7-10 Rez input for a size resource

```
resource 'SIZE' (-1, purgeable)  {
   reserved,
   acceptSuspendResumeEvents,
   reserved,
   canBackground,
   doesActivateOnFGSwitch,
   backgroundAndForeground,
   dontGetFrontClicks,
   ignoreAppDiedEvents,
   is32BitCompatible,
   isHighLevelEventAware,
   localAndRemoteHLEvents,
   isStationeryAware,         /*support stationery pads*/
   dontUseTextEditServices,
   reserved, reserved, reserved,
   kPrefSize * 1024,
   kMinSize * 1024
};
```

Notice that the twelfth field, `isStationeryAware`, tells the Finder that this application supports stationery pads.

If the `isStationeryAware` bit is not set in the size resource, the Finder creates a new document from the template and prompts the user for a name. The Finder then starts up your application as usual, passing it the name of the new document.

If the `isStationeryAware` bit is set, as shown in Listing 7-10, the Finder informs your application that the user has opened a document and passes your application the name of the stationery pad.

To support stationery, your application should

- specify the `isStationeryAware` constant in its size resource
- always check the `isStationery` bit of a document before opening it

Listing 7-11 on page 7-36 illustrates a simple function that takes a file system specification record and returns TRUE or FALSE, indicating whether the file is a stationery document or not.

Listing 7-11 Determining whether a document is a stationery pad

```
FUNCTION IsStationeryDoc (myFSSpec: FSSpec): Boolean;
VAR
    myErr:      OSErr;
    myFInfo:    FInfo;
BEGIN
    myErr := FSpGetFInfo(myFSSpec, myFInfo);
    IF myErr = noErr THEN
        IsStationeryDoc := BTST(myFInfo.fdFlags, isStationery)
    ELSE
        IsStationeryDoc := FALSE;
END;
```

The `isStationery` bit alone identifies whether a document is stationery. If the `isStationery` bit is set for a file that the user wants to open, your application should copy the template's contents into a new document and open the document in an untitled window. (For information about opening documents and about the File Manager function `FSpGetFInfo`, see *Inside Macintosh: Files*.)

Your application can check the `sfFlags` field of the standard file reply record to determine whether the `isStationery` bit is set. Unlike the Finder, the Standard File Package always passes your application the stationery pad itself, not a copy of it, regardless of the setting of the `isStationery` bit. When the user opens a stationery pad from within your application, the Standard File Package checks your application's size resource. If your application does not support stationery, the Standard File Package displays an alert box warning the user that the stationery pad itself, not a copy of it, is being opened. As you can see, the user can still easily change the template and mistakenly write over it by choosing Save without assigning a new name. You can prevent this unnecessary user frustration by making your application stationery-aware.

You can supply the icon to be displayed for stationery pads created from your application's documents by using the resources described in "Creating Icons for the Finder" beginning on page 7-11. If you do not supply your own stationery pad icon, the Finder uses the default stationery pad icon illustrated in Figure 7-5 on page 7-12.

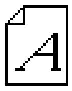
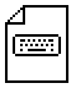
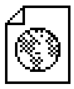
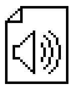
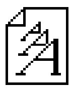
In your documentation, tell users to choose the Get Info command to make stationery pads. You may also want to give examples of useful stationery pads created with your application. For example, if your application supports text and graphics, you may provide samples of stationery pads for business letterheads or billing statements.

Distributing Fonts, Sounds, and Other Movable Resources

If you create fonts, sounds, keyboard layouts, and script system resource collections, you can distribute them in individual, movable resource files.

Movable resources such as fonts, keyboard layouts, and sounds are represented on the screen by icons. To install these resources, the user drags their icons to the System Folder icon. The Finder puts font resources in the Fonts folder, and it puts the other resources in the System file. The user can determine which fonts are currently installed by double-clicking the System Folder to open it and then double-clicking the Fonts folder. By double-clicking the System file so that it opens like a folder, the user can see which other movable resources are installed. (For a description of the new organization of the System Folder, see “Using the System Folder and Its Related Directories” beginning on page 7-41.)

To make one of these resources visible on the screen, assign it one of the special file types defined by the Finder for movable resources. The following list shows the resources that can be moved, their assigned file types, and their icons:

Resource	**File type**	**Large black-and-white icon**
Font	`'ffil'`	
Keyboard layout	`'kfil'`	
Script system resource collection	`'ifil'`	
Sound	`'sfil'`	
TrueType font	`'tfil'`	

Note

You or your users can give customized icons to these file types (as described in “Creating Customized Document Icons” beginning on page 7-17) as long as the files are not installed in the System file or in a suitcase file. As soon as users install them in the System file or in a suitcase file, the Finder displays them using the icons shown in the previous list. Font and TrueType font movable resources retain their custom icons when installed in the Fonts folder. u

The user can still store fonts (as well as desk accessories) in files that have suitcase icons, which is how they were distributed for installation or saved by the user using the Font/DA Mover in versions of system software that preceded System 7. A suitcase file that holds desk accessories is of type `'DFIL'`, and a suitcase file that holds fonts is of type `'FFIL'`. All suitcase files have a creator of `'DMOV'`.

In your documentation, tell users to install fonts, sounds, or script system resource collections by dragging their icons to the System Folder icon. A dialog box appears asking the user to verify that the resource should be installed in either the Fonts folder or the System file. The user clicks OK to accept the installation. The user also has the option to click Cancel to prevent the installation.

Note

If users drag icons to the open System Folder window instead of to the System Folder icon, the Finder copies or moves the files into the System Folder directory instead of installing them into either the Fonts folder or the System file. u

Providing Balloon Help for Nondocument Icons

The Finder offers Balloon Help online assistance for users. After the user chooses Show Balloons from the Help menu, descriptive help balloons appear when the user moves the cursor to an area of the screen (such as a menu, a window control, or a dialog box) that has a help resource associated with it.

The Finder provides default help balloons for application, control panel, and system extension icons. You can provide a customized help balloon for your application, control panel, or system extension icon by adding an `'hfdr'` resource with resource ID –5696 to the resource fork of your application. Figure 7-14 compares the default help balloon with a customized help balloon for the SurfWriter application icon.

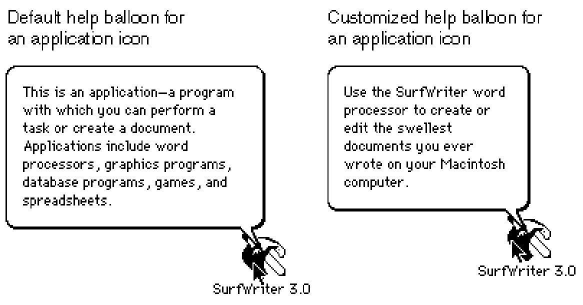


Figure 7-14 Default and customized help balloons for application icons

Default help balloon for an application icon
Customized help balloon for an application icon
SurfWriter 3.0
SurfWriter 3.0

This is an application—a program with which you can perform a task or create a document. Applications include word processors, graphics programs, database programs, games, and spreadsheets.

Use the SurfWriter word processor to create or edit the swellest documents you ever wrote on your Macintosh computer.

Listing 7-12 shows a Finder help override resource and its associated `'STR '` resource, which are used for the customized help balloon shown in Figure 7-14.

Note

You cannot override the default help balloon that the Finder uses for document icons. u

The chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox* describes in detail how to provide Balloon Help for your application icon and for other elements of your application.

Listing 7-12 Rez input for a help balloon resource for an application icon

```
resource 'hfdr' (-5696, purgeable) {   /*help for SurfWriter icon*/
   HelpMgrVersion, hmDefaultOptions, 0, 0,   /*header information*/
   {HMSTRResItem {kIconHelpString}}
};
resource 'STR ' (kIconHelpString, purgeable) {/*help message for app icon*/
   "Use the SurfWriter word processor to create or edit the "
   "swellest documents you ever wrote on your Macintosh computer."
};
```

Using Aliases

The Finder allows the user to create multiple icons to represent a single document or other desktop object (such as a disk, a folder, or the Trash). One of the icons represents the actual file; the others are aliases that point to the file. An **alias** is an object that represents some other file, directory, or volume. An alias looks like the icon of its target, but its name is displayed in a different style. The style depends on the system script; for Roman and most other scripts, alias names are displayed in italic.

To the user, the icons of the actual file and its aliases are functionally identical. Aliases give the user more flexibility in organizing files and offer a convenient way to store a local copy of a large or dynamic file that resides on a file server.

Ordinarily, when the user wants to open or print files, your application does not need to be concerned with whether they are aliases because both the Finder and the Standard File Package resolve aliases before passing them to your application. If the user opens an alias that represents a document created by your application, the Finder passes your application the name and location of the document itself, not the alias. Similarly, when the user opens an alias from within your application, the Standard File Package passes your application the name of the target document.

If your application opens a file or a directory without going through the Finder or the Standard File Package (if, for example, it uses preference files or dictionary files), your application should always call the `ResolveAliasFile` function just before opening the file.

As a Finder object, the alias depicts a file called the **alias file,** which contains a record that points to the file, directory, or volume represented by the icon. Alias files are created and managed by the user through the Finder.

Although your application shouldn’t create alias files or change users’ aliases, your application can create and use its own alias records for storing identifying information about files or directories. An **alias record** is a data structure that identifies a file, folder, or volume. Whenever your application needs to store file or directory information, you can record the location and other identifying information in an alias record. The next time your application needs the file or directory, you can use the Alias Manager to locate it, even if the user has renamed it, copied it, restored it from backup, or moved it. You can also use alias records to identify objects on other volumes, including AppleShare volumes. See the chapter "Alias Manager" in *Inside Macintosh: Files* for details about creating and managing information in alias records.

An alias file contains an alias record, stored as a resource of type `'alis'`, that points to the target of the alias. (The **alias target** is the file, directory, or volume described by the alias record.) The alias file might also contain the target object's icon descriptions. The Finder identifies an alias file by setting the `isAlias` bit in the file's Finder flags field (see "File Information Record" beginning on page 7-47 for a description of Finder flags).

An alias file that represents a document typically has the same type and creator as the file it represents. However, many Finder objects—such as disks, folders, and the Trash—do not have file types. Instead, alias files for these objects are assigned special file types, called *alias types.* Here are the alias types for those objects for which users can create aliases:

Object	Alias type	Constant
Apple Menu Items folder	`'faam'`	`kAppleMenuFolderAliasType`
AppleShare drop folder	`'fadr'`	`kDropFolderAliasType`
Application	`'adrp'`	`kApplicationAliasType`
Control Panels folder	`'fact'`	`kControlPanelFolderAliasType`
Exported AppleShare folder	`'faet'`	`kExportedFolderAliasType`
Extensions folder	`'faex'`	`kExtensionFolderAliasType`
File server	`'srvr'`	`kContainerServerAliasType`
Floppy disk	`'flpy'`	`kContainerFloppyAliasType`
Folder	`'fdrp'`	`kContainerFolderAliasType`
Hard disk	`'hdsk'`	`kContainerHardDiskAliasType`
Mounted AppleShare folder	`'famn'`	`kMountedFolderAliasType`
Other objects that can hold files	`'drop'`	`kContainerAliasType`
Preferences folder	`'fapf'`	`kPreferencesFolderAliasType`
PrintMonitor Documents folder	`'fapn'`	`kPrintMonitorDocsFolderAliasType`
Shared AppleShare folder	`'fash'`	`kSharedFolderAliasType`
Startup Items folder	`'fast'`	`kStartupFolderAliasType`
System Folder	`'fasy'`	`kSystemFolderAliasType`
Trash	`'trsh'`	`kContainerTrashAliasType`

(The Extensions, Preferences, Apple Menu Items, Control Panels, Startup Items, and PrintMonitor Documents folders are described in "Using the System Folder and Its Related Directories" beginning on page 7-41.)

When opening a file without going through the Finder or the Standard File Package, you call `ResolveAliasFile` immediately before opening the file. (The `ResolveAliasFile` function is described in detail on page 7-52.) In Listing 7-13, the customized open function `MyOpen` ensures that the file to be opened is the target file and then opens the data fork with the File Manager function `FSpOpenDF`.

Listing 7-13 Using the `ResolveAliasFile` function to open a file

```
FUNCTION MyOpen (VAR theSpec: FSSpec; perm: SignedByte;
                 VAR fRefNum: Integer): OSErr;
VAR
   myErr:            OSErr;
   targetIsFolder:   Boolean;
   wasAliased:       Boolean;
BEGIN
   myErr := ResolveAliasFile(theSpec, TRUE, targetIsFolder, wasAliased);
   IF targetIsFolder THEN
      myErr := paramErr                {cannot open a folder}
   ELSE IF (myErr <> noErr ) THEN      {try to open it}
      myErr := FSpOpenDF(theSpec, perm, fRefNum);
   MyOpen := myErr;
END;
```

Using the System Folder and Its Related Directories

The System Folder is a directory that stores essential system software such as the System file, the Finder, and printer drivers. System 7 introduced a new organization for the System Folder, which contains a set of new subdirectories to hold related files. The Finder uses these subdirectories to facilitate file management for the user. For example, by sorting and storing such files as desk accessories, control panels, fonts, preferences files, system extensions, and temporary files into separate folders for the user, the Finder keeps the top level of the System Folder from being cluttered with dozens, or even hundreds, of files.

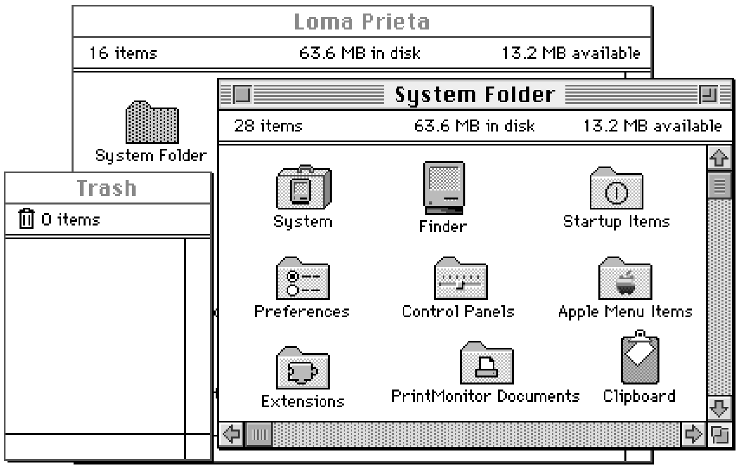
The user can easily install and remove fonts, sounds, keyboard layouts, control panels, and system extensions by dragging their icons to the System Folder icon. The Finder then moves them into the proper subdirectories. When a control panel icon is dragged to the System Folder icon, for example, the Finder presents a dialog box that asks the user, “Place this control panel into the ‘Control Panels’ folder?” The user accepts by clicking the OK button or declines by clicking the Cancel button.

Note

If users drag icons to the open System Folder window instead of to the System Folder icon, the Finder copies or moves the files into the System Folder directory instead of copying or moving them to the proper subdirectories. u

Figure 7-15 shows a user’s view of the new directory organization typically found within the System Folder.

Figure 7-15 The System Folder and related folders



Additional related directories are located at the root directory. Notice the Trash window. It shows the contents of the Trash directory, which is represented to the user by the Trash icon. The Trash directory exists at the root level of the volume. A Macintosh sharing files among users in a network environment maintains separate Trash subdirectories within a shared Trash directory. That is, the server creates a separate, uniquely named Trash subdirectory for every user who opens a volume on a Macintosh server and drags an object to the Trash icon. All Trash subdirectories within a shared Trash directory are invisible to users. On the desktop, the user sees only the Trash icon of the local Macintosh computer. When the user double-clicks the Trash icon, a window reveals the names of only those files that the user has thrown away; no distinction is made to the user as to which computers any of these files originated on.

At the root level of the volume, the Finder also maintains a Temporary Items folder and a Desktop Folder, both of which are invisible to the user and so don’t appear in Figure 7-15.

Figure 7-15 illustrates the folder organization typically found on single-user systems. Of all the related directories shown, your application is likely to use only the Preferences folder and the Temporary Items folder. However, you cannot be certain of the location of these or any of the other system-related directories. In the future, these system-related directories may not be located in the System Folder or in the root directory.

You can use the `FindFolder` function (described on page 7-54) to get the path information to these directories. Of these directories, the only ones you are ever likely to need are Preferences, Temporary Items, and Trash. For example, you might wish to check for the existence of a user’s configuration file in Preferences, create a temporary file in Temporary Items, or—if your application runs out of storage when trying to save a file—check how much storage is taken by items in the Trash directory and report this to the user.

Your application may freely use these two directories for storing and locating important files:

- **Preferences,** located in the System Folder, holds preferences files to record local configuration settings. Your application can store its preferences file in this directory. The active Finder Preferences file is always stored in the Preferences folder. Do not use the Preferences folder to hold information that is to be shared by users on more than one Macintosh computer on a network. Ensure that your application can always operate even if its preferences file has been deleted.
- **Temporary Items,** located at the root level of the volume, holds temporary files created by applications. The Temporary Items folder is invisible to the user. Your application can place its temporary files in this directory. A temporary file should exist only as long as your application needs to keep it open. As soon as your application closes the file, your application should remove the temporary file. You should also ensure that you are assigning a unique name to your temporary file so that you don't write over another application's file.

It's important to bear in mind a few rules about storing your application's files. First, don't store any files at the top level of the System Folder. Use the Preferences directory or one of the other directories described in the following list.

Second, use the `FindFolder` function to locate or put files in the right place. Don't assume files are on the same volume as your application; they could be on a different local volume, or on a remote volume on the network.

Third, don't store any files that multiple users may need to access, such as dictionaries and format converters, in the Preferences directory or in any of the directories located in the System Folder. Remember that the files in the System Folder are generally accessible only to the person who starts up from the System file in that System Folder.

There are additional directories that either the user or the Finder uses for storing and locating important files; these directories are described here. Generally, your application should not store files in these directories.

- **Apple Menu Items,** located in the System Folder, holds the standard desk accessories plus any other desk accessories, applications, files, folders, or aliases that the user wants to display in the Apple menu. Only the user and the Installer should put things into the Apple Menu Items folder.
- **Control Panels,** located in the System Folder, holds control panels. The Apple Menu Items folder holds an alias to the Control Panels folder so that the user can also reach the control panels through the Apple menu. Only the user and the Installer should put things into the Control Panels folder.
- **Desktop Folder,** which is invisible to users, is located at the root level of the volume. The Desktop Folder stores information about the icons that appear on the desktop area of the screen. The user controls the contents of the Desktop Folder by arranging icons on the screen. What appears on the screen to the user is the union of the contents of Desktop Folders for all mounted volumes.
- **Extensions,** located in the System Folder, holds extensions—that is, code that is not part of the basic system software but that provides system-level services, such as printer drivers and system extensions. Files of type `'INIT'`, previously called startup documents, and of type `'appe'`, also known as background-only applications, are

routed by the Finder to this folder. Files of type `'scri'` (system extensions for script systems) are also routed to this folder. Only the user and the Installer should put files into the Extensions folder.

- **Fonts,** located in the System Folder on computers using system software version 7.1 or later, holds fonts. Only the user and the Installer should put fonts into the Fonts folder.
- **PrintMonitor Documents,** located in the System Folder, holds spooled documents waiting to be printed. Only the printing software uses the PrintMonitor Documents folder.
- **Rescued Items from *volume name,*** located in the Trash directory, is a directory created by the Finder at system startup, restart, or shutdown only when the Finder finds items in the Temporary Items folder. Since applications should remove their temporary files when they close them, the existence of a file in a Temporary Items folder indicates a system crash. When the Finder discovers a file in the Temporary Items folder, the Finder creates a Rescued Items from *volume name* directory that is named for the volume on which the Temporary Items folder exists. For example, the Finder creates a directory called Rescued Items from Loma Prieta when a file is discovered in the Temporary Items folder on a volume named Loma Prieta. The Finder then moves the temporary file to that directory so that users can examine the file in case they want to recreate their work up to the time of the system crash. When a user empties the Trash, all Rescued Items folders disappear. Only the Finder should put anything into Rescued Items directories.
- **Startup Items,** located in the System Folder, holds applications and desk accessories (or their aliases) that the user wants started up every time the Finder starts up. Only the user should put things into the Startup Items folder. Note that there is a distinction between startup applications that users put in the Startup Items folder and system extensions of file type `'INIT'` (previously called startup documents), which are typically installed in the Extensions folder.
- **System file,** located in the System Folder, contains the basic system software plus some system resources, such as sound and keyboard resources. The System file behaves like a folder in this regard: although it looks like a suitcase icon, double-clicking it opens a window that reveals movable resource files (such as sounds, keyboard layouts, and script system resource collections) stored in the System file. ("Distributing Fonts, Sounds, and Other Movable Resources" beginning on page 7-36 describes the resources that can be moved into the System file.) Only the user and the Installer should put resources into the System file.
- **Trash,** located at the root level of a volume, holds items that the user moves to the Trash icon. After opening the Trash icon, the user sees the collection of all items that he or she has moved to the Trash icon—that is, the union of all appropriate Trash directories from all mounted volumes. A Macintosh set up to share files among users in a network environment maintains separate Trash subdirectories for remote users within its shared Trash directory. That is, the server creates a separate, uniquely named Trash subdirectory for every remote user who opens a volume on a Macintosh file server and drags an object to the Trash icon. All Trash subdirectories and the shared Trash directory are invisible to users. The Finder empties a Trash directory (or, in the case of a file server, a Trash subdirectory) only when the user of that directory chooses the Empty Trash command.

Although the names of the visible system-related folders vary on different international systems, the invisible directories Temporary Items and Desktop Folder keep these names on all systems. System software assigns unique names for invisible Trash subdirectories.

Generally, you should store application-specific files in the folder with your application, not in any of these system-related directories. Your application may want to provide users with a mechanism to specify a directory in which to look for auxiliary files. For example, you could design a customized version of the open file dialog box that allows users to specify a path to locations where files are stored. This technique may be useful for finding files that are shared by several applications. It's also possible to track the location of files by using the Alias Manager. For details, see the chapter "Alias Manager" in *Inside Macintosh: Files*.

When you design your application, it's important to consider the user's view of the tools that you provide. In most cases you'll want to build your application so that the user deals with one icon that represents the entire set of abilities your application provides. This scheme simplifies the user's world by restricting the complexity of installing and maintaining your product. If you provide optional tools—such as a dictionary and thesaurus—that have their own icons, it's a good idea to allow these tools to work from any location in the file system rather than relying on their storage somewhere in the System Folder.

The Desktop Database

For quick access to the resources it needs, the Finder maintains a central desktop database of information about the files and directories on a volume. The Finder updates the database when applications are added, moved, renamed, or deleted.

Normally, your application won't need to use the information in the desktop database or to use Desktop Manager routines to manipulate it. Instead, your application should let the Finder manipulate the desktop database and handle such Desktop Manager tasks as launching applications when users double-click icons, maintaining user comments associated with files, and managing the icons used by applications.

In case you discover some important need to retrieve information from the desktop database or even to change the desktop database from within your application, Desktop Manager routines are provided for you to do so. While your application probably won't ever need to use them, for the sake of completeness they are described in *Inside Macintosh: More Macintosh Toolbox*.

Much of the information in the desktop database comes from the bundle resources for applications and other files on the volume. (See "Using Finder Information in the Catalog File" beginning on page 7-32 for a discussion on setting the bundle bit of an application so that its bundled resources get stored in the desktop database.) The desktop database contains all icon definitions and their associated file types. It lists all the file types that each application can open and all copies or versions of the application that's listed as the creator of a file. The desktop database also lists the location of each application on the disk and any comments that the user has added to the information windows for desktop objects.

The Finder maintains a desktop database for each volume with a capacity greater than 2 MB. For most volumes, such as hard disks, the database is stored on the volume itself. For read-only volumes—such as some compact discs—that don't contain their own desktop database, the Desktop Manager creates it and stores it in the System Folder of the startup drive.

For compatibility with older versions of system software, the Finder keeps the information for ejectable volumes with a capacity smaller than 2 MB in a resource file instead of a database.

Finder Interface Reference

This section describes the data structures, routines, and resources that are specific to the Finder interface.

The "Data Structures" section shows the data structures for the file information record, the extended file information record, the directory information record, and the extended directory information record. The "Routines" section describes the routines for resolving alias files and for finding system-related folders. The "Resources" section describes the resources you supply for your files so that the Finder can relay information about them to your users.

Data Structures

A catalog file exists on every volume to maintain relationships between the files and directories on that volume. (A volume is any storage medium formatted to contain files.) Although it's used mostly by the File Manager, the catalog file also contains information used by the Finder. The information for files is listed in file information records and extended file information records; the information for directories is listed in directory information records and extended directory information records.

Normally, your application sets the file type and the creator information in fields of a file information record when your application creates a new file. (For a complete discussion of the File Manager and the functions available for creating files, see *Inside Macintosh: Files*.) The Finder manipulates the other fields in the file information record. You can check the information in this record by calling the File Manager function `FSpGetFInfo` or `PBGetCatInfo`. In particular, you may want to check the file type or creator for a file, or you may want to check or set one of your document's Finder flags.

The Finder manipulates the fields in the extended file information, directory information, and extended directory information records; your application shouldn't have to directly check or set any of these fields. These data structures are described here for completeness.

File Information Record

You typically set a file's type and creator when you create the file; for example, you pass a creator and a file type to the File Manager function `FSpCreate` as parameters. The Finder manipulates the other fields in the file information record, which is a data structure of type `FInfo`. After you have created a file, you can use the File Manager function `FSpGetFInfo` to return the file information record, then change the `fdType` and `fdCreator` fields by using the File Manager function `FSpSetFInfo`.

```
TYPE  FInfo =
      RECORD
         fdType:     OSType;     {file type}
         fdCreator:  OSType;     {file creator}
         fdFlags:    Integer;    {Finder flags}
         fdLocation: Point;      {file's location in window}
         fdFldr:     Integer;    {window that contains file}
      END;
```

Field descriptions

`fdType`	File type. For a discussion of file types, see "Giving a Signature to Your Application and a Creator and a File Type to Your Documents" beginning on page 7-8.
`fdCreator`	The signature of the application that created the file. For a discussion about creators, see "Giving a Signature to Your Application and a Creator and a File Type to Your Documents" beginning on page 7-8.
`fdFlags`	Finder flags. There are only a few flags that your application might ever need to set; these are described in "Using Finder Information in the Catalog File" beginning on page 7-32. All of the Finder flags are listed here for completeness.

Flag name	Bit number	Description
`isAlias`	15	For a file, this bit indicates that the file is an alias file. For directories, this bit is reserved—in which case, set to 0.
`isInvisible`	14	The file or directory is invisible from the Finder and from the Standard File Package dialog boxes.
`hasBundle`	13	For a file, this bit indicates that the file contains a bundle resource. For directories, this bit is reserved—in which case, set to 0.
`nameLocked`	12	The file or directory can't be renamed from the Finder, and the icon cannot be changed.
`isStationery`	11	For a file, this bit indicates that the file is a stationery pad. For directories, this bit is reserved—in which case, set to 0.

Flag name	Bit number	Description
`hasCustomIcon`	10	The file or directory contains a customized icon.
`Reserved`	9	Reserved; set to 0.
`hasBeenInited`	8	The Finder has recorded information from the file's bundle resource into the desktop database and given the file or folder a position on the desktop.
`hasNoINITS`	7	The file contains no `'INIT'` resources; set to 0. Reserved for directories; set to 0.
`isShared`	6	The file is an application that can be executed by multiple users simultaneously. Defined only for applications; otherwise, set to 0.
`requiresSwitchLaunch`	5	Unused and reserved in System 7; set to 0.
`colorReserved`	4	Unused and reserved in System 7; set to 0.
`color`	1–3	Three bits of color coding.
`isOnDesk`	0	Unused and reserved in System 7; set to 0.

You can use these constants as masks for these flags:

```
CONST
fHasBundle  =  8192;    {set if file has a bundle }
                        { resource}
fInvisible  =  16384;   {set if icon is invisible}
kIsOnDesk   =  $1;      {unused and reserved in }
                        { System 7}
kColor      =  $E;      {three bits of color }
                        { coding}
kIsShared   =  $40;     {file can be executed by }
                        { multiple users }
                        { simultaneously}
kHasBeenInited
            =  $100;    {file info is in desktop }
                        { database}
kHasCustomIcon
            =  $400;    {file or directory has a }
                        { customized icon}
kIsStationery
             = $800;    {file is a stationery pad}
kNameLocked =  $1000;   {file or directory can't }
                        { be renamed from Finder, }
                        { and icon can't be }
                        { changed}
kHasBundle  =  $2000;   {file has bundle resource}
```

```
                kIsInvisible=  $4000;   {file or directory is }
                                        { invisible from Finder & }
                                        { from Standard File }
                                        { Package dialog boxes}
                kIsAlias    =  $8000;   {file is an alias file}
```

`fdLocation`	The location—specified in coordinates local to the window—of the file's icon within its window.
`fdFldr`	The window in which the file's icon appears; this information is meaningful only to the Finder.

Extended File Information Record

The Finder manipulates the fields in the extended file information records, which are data structures of type `FXInfo`; your application shouldn't have to check or set any of these fields directly.

```
TYPE  FXInfo =
      RECORD
         fdIconID:      Integer;    {icon ID}
         fdUnused:      ARRAY[1..3] OF Integer;
                                    {unused but reserved 6 bytes}
         fdScript:      SignedByte; {script flag and code}
         fdXFlags:      SignedByte; {reserved}
         fdComment:     Integer;    {comment ID}
         fdPutAway:     LongInt;    {home directory ID}
      END;
```

Field descriptions

`fdIconID`	An ID number for the file's icon; the numbers that identify icons are assigned by the Finder.
`fdUnused`	Reserved.
`fdScript`	The script system for displaying the file's name. Ordinarily, the Finder (and the Standard File Package) displays the names of all desktop objects in the system script, which depends on the region-specific configuration of the system. The high bit of the byte in the `fdScript` field is set by default to 0, which causes the Finder to display the filename in the current system script. If the high bit is set to 1, the Finder (and the Standard File Package) displays the filename and directory name in the script whose code is recorded in the remaining 7 bits.
`fdXFlags`	Reserved.
`fdComment`	An ID number for the comment that is displayed in the information window when the user selects a file and chooses the Get Info command from the File menu. The numbers that identify comments are assigned by the Finder.
`fdPutAway`	If the user moves the file onto the desktop, the directory ID of the folder from which the user moves the file.

Directory Information Record

The Finder manipulates the fields in the directory information record, which is a data structure of type `DInfo`. Your application shouldn't have to check or set any of these fields directly.

```
TYPE DInfo =
      RECORD
         frRect:     Rect;    {folder's window rectangle}
         frFlags:    Integer; {flags}
         frLocation: Point;   {folder's location in window}
         frView:     Integer; {folder's view}
      END;
```

Field descriptions

`frRect`	The rectangle for the window that the Finder displays when the user opens the folder.
`frFlags`	Reserved.
`frLocation`	Location of the folder in the parent window.
`frView`	The manner in which folders are displayed; this is set by the user with commands from the View menu of the Finder.

Extended Directory Information Record

The Finder manipulates the fields in the extended directory information records, which are data structures of type `DXInfo`; your application shouldn't have to check or set any of these fields directly.

```
TYPE  DXInfo =
      RECORD
         frScroll:      Point;      {scroll position}
         frOpenChain:   LongInt;    {directory ID chain of open }
                                    { folders}
         frScript:      SignedByte; {script flag and code}
         frXFlags:      SignedByte; {reserved}
         frComment:     Integer;    {comment ID}
         frPutAway:     LongInt;    {home directory ID}
      END;
```

Field descriptions

`frScroll`	Scroll position within the Finder window. The Finder does not necessarily save this position immediately upon user action.

`frOpenChain`	Chain of directory IDs for open folders. The Finder numbers directory IDs. The Finder does not necessarily save this information immediately upon user action.
`frScript`	The script system for displaying the folder's name. Ordinarily, the Finder (and the Standard File Package) displays the names of all desktop objects in the current system script, which depends on the region-specific configuration of the system. The high bit of the byte in the `fdScript` field is set by default to 0, which causes the Finder to display the folder's name in the current system script. If the high bit is set to 1, the Finder (and the Standard File Package) displays the filename and directory name in the script whose code is recorded in the remaining 7 bits. However, as of system software version 7.1, the Window Manager and Dialog Manager do not support multiple simultaneous scripts, so the system script is always used for displaying filenames and directory names in dialog boxes, window titles, and other user interface elements used by the Finder. Therefore, until the system software's script capability is fully implemented, you should treat this field as reserved.
`frXFlags`	Reserved.
`frComment`	An ID number for the comment that is displayed in the information window when the user selects a folder and chooses the Get Info command from the File menu. The numbers that identify comments are assigned by the Finder.
`frPutAway`	If the user moves the folder onto the desktop, the directory ID of the folder from which the user moves it.

Routines

This section describes the routines your application can use to resolve alias files if it bypasses the Finder when manipulating documents and to find system-related folders if your application needs to determine where they are located.

Resolving Alias Files

Ordinarily, when the user wants to open or print files, your application does not need to be concerned with whether they are aliases because the Finder resolves aliases before passing them to your application. If the user opens an alias that represents a document created by your application, the Finder passes your application the name and location of the document itself, not the alias. (Similarly, when the user opens an alias from within your application, the Standard File Package passes your application the name of the target document.) If your application bypasses the Finder when manipulating documents, it should check for and resolve aliases itself by using the Alias Manager function `ResolveAliasFile`, which is described here for completeness.

ResolveAliasFile

If your application bypasses the Finder when manipulating documents, it should check for and resolve aliases itself by using the `ResolveAliasFile` function.

```
FUNCTION ResolveAliasFile (VAR theSpec: FSSpec;
                           resolveAliasChains: Boolean;
                           VAR targetIsFolder: Boolean;
                           VAR wasAliased: Boolean): OSErr;
```

`theSpec` A file system specification record for the file or directory you plan to open.

`resolveAliasChains`
A Boolean value. Set this parameter to `TRUE` if you want `ResolveAliasFile` to resolve all aliases in a chain, stopping only when it reaches the target file. Set this parameter to `FALSE` if you want to resolve only one alias file, even if the target is another alias file.

`targetIsFolder`
A return parameter only. The `ResolveAliasFile` function returns `TRUE` in this parameter if the file specification record in the parameter `theSpec` points to a directory or a volume; otherwise, `ResolveAliasFile` returns `FALSE` in this parameter.

`wasAliased`
A return parameter only. The `ResolveAliasFile` function returns `TRUE` in this parameter if the file specification record in the parameter `theSpec` points to an alias; otherwise, `ResolveAliasFile` returns `FALSE` in this parameter.

DESCRIPTION

The `ResolveAliasFile` function returns in the parameter `theSpec` the name and location of the target file that you initially pass in the parameter `theSpec`.

The `ResolveAliasFile` function first checks the catalog file for the file or directory specified in the parameter `theSpec` to determine whether it is an alias and whether it is a file or a directory. If the object is not an alias, `ResolveAliasFile` leaves `theSpec` unchanged, sets the `targetIsFolder` parameter to `TRUE` for a directory or volume and `FALSE` for a file, sets `wasAliased` to `FALSE`, and returns `noErr`. If the object is an alias, `ResolveAliasFile` resolves it, places the target in the parameter `theSpec`, and sets the `wasAliased` flag to `TRUE`.

When `ResolveAliasFile` finds the specified volume and parent directory but fails to find the target file or directory in that location, `ResolveAliasFile` returns a result code of `fnfErr` and fills in the parameter `theSpec` with a complete file system specification record describing the target (that is, its volume reference number, parent directory ID, and filename or folder name). The file system specification record is valid, although the object it describes does not exist. This information is intended as a “hint” that lets you explore possible solutions to the resolution failure. You can, for example, use the file system specification record to create a replacement for a missing file with the File Manager function `FSpCreate`.

If `ResolveAliasFile` receives an error code while resolving an alias, it leaves the input parameters as they are and exits, returning an error code. In addition to any of these result codes, `ResolveAliasFile` can also return any Resource Manager or File Manager errors.

SPECIAL CONSIDERATIONS

Before calling the `ResolveAliasFile` function, you should make sure that it is available by using the `Gestalt` function with the `gestaltAliasMgrAttr` selector.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`fnfErr`	–43	Target not found, but volume and parent directory found, and `theSpec` parameter contains a valid file system specification record
`dirNFErr`	–120	Parent directory not found

SEE ALSO

Listing 7-13 on page 7-41 illustrates how to use `ResolveAliasFile` from an application’s own `MyOpen` function. The file system specification record is described in *Inside Macintosh: Files.* Aliases and other Alias Manager and File Manager routines are also described in greater detail in *Inside Macintosh: Files*. The `Gestalt` function is described in the chapter “Gestalt Manager” in *Inside Macintosh: Operating System Utilities.*

Finding Directories

You can use the `FindFolder` function to get the path information you need to gain access to the system-related directories described in “Using the System Folder and Its Related Directories” beginning on page 7-41. Those you’re most likely to want to access are Preferences, Temporary Items, and Trash. For example, you might wish to check for the existence of a user’s configuration file in Preferences, create a temporary file in Temporary Items, or—if your application runs out of disk storage when trying to save a file—check how much disk storage is taken by items in the Trash directory and report this to the user.

FindFolder

To get the path information to gain access to the system-related directories, use the `FindFolder` function.

```
FUNCTION FindFolder (vRefNum: Integer; folderType: OSType;
                     createFolder: Boolean;
                     VAR foundVRefNum: Integer;
                     VAR foundDirID: LongInt): OSErr;
```

`vRefNum` The volume reference number (or the constant `kOnSystemDisk` for the startup disk) of the volume on which you want to locate a directory.

`folderType`

A four-character folder type, or a constant that represents the type, for the directory you want to find. The constants and the four-character folder types they represent are listed here:

```
CONST
kAppleMenuFolderType
                     = 'amnu';   {Apple Menu Items}
kControlPanelFolderType
                     = 'ctrl';   {Control Panels}
kDesktopFolderType   = 'desk';   {Desktop Folder}
kExtensionFolderType
                     = 'extn';   {Extensions}
kFontsFolderType     = 'font';   {Fonts folder}
kPreferencesFolderType
                     = 'pref';   {Preferences}
kPrintMonitorDocsFolderType
                     = 'prnt';   {PrintMonitor  }
                                 { Documents}
kStartupFolderType   = 'strt';   {Startup Items}
kSystemFolderType    = 'macs';   {System Folder}
kTemporaryFolderType
                     = 'temp';   {Temporary Items}
kTrashFolderType     = 'trsh';   {single-user Trash}
kWhereToEmptyTrashFolderType
                     = 'empt';   {shared Trash on net}
```

`createFolder`

Pass the constant `kCreateFolder` in this parameter to create a directory if it does not already exist; otherwise, pass the constant `kDontCreateFolder`.

`foundVRefNum`
: The volume reference number, returned by `FindFolder`, for the volume containing the directory you specify in the `folderType` parameter.

`foundDirID`
: The directory ID number, returned by `FindFolder`, for the directory you specify in the `folderType` parameter.

DESCRIPTION

For the folder type on the particular volume (specified, respectively, in the `folderType` and `vRefNum` parameters), the `FindFolder` function returns the directory's volume reference number in the `foundVRefNum` parameter and its directory ID in the `foundDirID` parameter.

The specified folder used for a given volume might be located on a different volume in future versions of system software; therefore, do not assume the volume that you specify in `vRefNum` and the volume returned in `foundVRefNum` will be the same.

Specify a volume reference number (or the constant `kOnSystemDisk` for the startup disk) in the `vRefNum` parameter.

Specify a four-character folder type—or the constant that represents it—in the `folderType` parameter. Use the `kTrashFolderType` constant to locate the current user's Trash directory for a given volume—even one located on a file server. On a file server, you can use the `kWhereToEmptyTrashFolderType` constant to locate the parent directory of all logged-on users' Trash subdirectories.

Use the constant `kCreateFolder` in the `createFolder` parameter to tell `FindFolder` to create a directory if it does not already exist; otherwise, use the constant `kDontCreateFolder`. Directories inside the System Folder are created only if the System Folder directory exists. The `FindFolder` function will not create a System Folder directory even if you specify the `kCreateFolder` constant in the `createFolder` parameter.

The `FindFolder` function returns a nonzero result code if the folder isn't found, and it can also return other file system errors reported by the File Manager or Memory Manager.

SPECIAL CONSIDERATIONS

The Finder identifies the subdirectories of the System Folder, and their folder types, in a resource of type `'fld#'` located in the System file. Do not modify or rely on the contents of the `'fld#'` resource in the System file; use only the `FindFolder` function to find the appropriate directories.

To determine the availability of the `FindFolder` function, use the `Gestalt` function with the Gestalt selector `gestaltFindFolderAttr`. Test the bit field indicated by the `gestaltFindFolderPresent` constant in the response parameter. If the bit is set, then the `FindFolder` function is present.

```
CONST gestaltFindFolderPresent = 0;    {if this bit is set, }
                                       { FindFolder is present}
```

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	Type not found in `'fld#'` resource, or disk doesn’t have System Folder support or System Folder in volume header, or disk does not have desktop database support for Desktop Folder—in all cases, folder not found
`dupFNErr`	–48	File found instead of folder

SEE ALSO

The system-related directories located by the `FindFolder` function are described in “Using the System Folder and Its Related Directories” beginning on page 7-41.

Resources

This section describes the resources you supply for your files so that the Finder can use your files and relay information about them to your users. These resources are

- the signature resource—defined using a string (`'STR '`) resource—which the Finder uses to identify and start up your application when a user double-clicks documents created by your application
- the set of resources (icon list resource, small icon list resource, large 4-bit color icon resource, small 4-bit color icon resource, large 8-bit color icon resource, and small 8-bit color icon resource) that visually represent your application and any documents it creates, and two related resources, the icon (`'ICON'`) resource and the color icon (`'cicn'`) resource
- the file reference (`'FREF'`) resource, which links icons with the files types they represent and which allows users to launch your application by dragging document icons to your application icon
- a bundle (`'BNDL'`) resource, which groups together your application’s signature, icon list resource, and file reference resources
- a missing-application name string—that is, a string (`'STR '`) resource—for your application’s documents in order to display the name of your application if the user tries to open or print a document created by your application when your application is missing
- an application-missing message string—that is, a string (`'STR '`) resource—in your application’s documents in order to explain why the user can’t open or print certain documents used by your application
- the version (`'vers'`) resource, so that users can easily find out the version of a file and, if applicable, the version of the superset of files to which the single file belongs

For information about using the `'SIZE'` resource to support stationery pads, see “Supporting Stationery Pads” beginning on page 7-34.

This section describes the structures of these resources after they are compiled by the Rez resource compiler, available from APDA. If you are interested in creating the Rez input files for these resources, see instead “Using the Finder Interface” beginning on page 7-6 for detailed information.

The Signature Resource

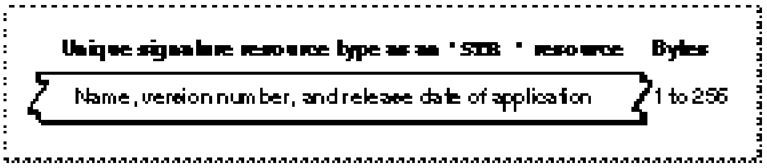
Every application that creates documents should define a signature resource, so that the Finder can identify and start up the application when a user double-clicks documents created by the application. A signature resource is typically defined to be a string resource (that is, a resource of type `'STR '`) that is given a unique four-character signature as its resource type. For example, an application with the signature of WAVE would use a string resource to define its signature resource as a resource of type `'WAVE'`. The signature resource should have a resource ID number of 0.

To ensure uniqueness, developers must register their applications' four-character signatures with Apple Computer, Inc., at Macintosh Developer Technical Support.

This section describes the structure of a signature resource defined to be of type `'STR '` after it's compiled by the Rez resource compiler. The format of a Rez input file for a signature resource differs from its compiled output form. If you are concerned only with creating a signature resource, see "Giving a Signature to Your Application and a Creator and a File Type to Your Documents" beginning on page 7-8.

If you examine a compiled version of a signature resource, as shown in Figure 7-16, you find that it contains a Pascal string that specifies the name, version number, and release date of the application.

Figure 7-16 Structure of a signature resource compiled as a string (`'STR '`) resource


Unique signature resource type as an 'STR ' resource
Bytes
Name, version number, and release date of application
1 to 256

If an application does not provide specific version information through a version resource (described in "Providing Version Resources" beginning on page 7-31), the Finder displays the string stored in the signature resource when the user selects the application and chooses Get Info from the File menu.

The Icon List Resource

An icon list resource is one of several icon resources that you create to represent visually for the user your application or one of the document types it creates. An icon list resource is a resource with the resource type `'ICN#'`. All icon list resources must be marked purgeable, and they must have resource IDs greater than 128.

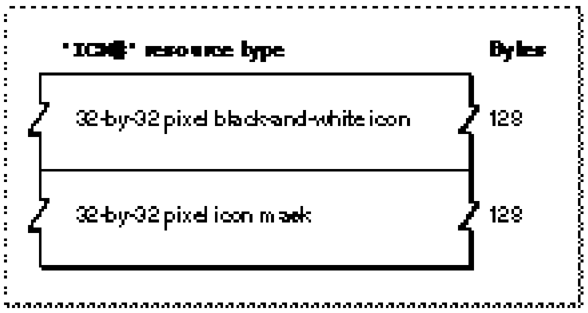
When the user chooses by Icon from the View menu, the Finder displays the black-and-white icon specified in this resource in windows if either the user has a black-and-white monitor or your application has not defined any resources for color icons; otherwise, the Finder displays a color version of the icon.

An icon list resource is defined to be an array of two items of type `String[128]`; each bit in the first array represents a pixel in the 32-by-32 pixel icon, and each bit in the second array represents a pixel in the 32-by-32 pixel mask. You can use a high-level tool such as the ResEdit application, which is available through APDA, to create icon list resources. You can then use the DeRez decompiler to convert your icon list resources into Rez input when necessary. See “Creating Icons for the Finder” beginning on page 7-11 for additional information about creating icon list resources and other resources for representing files to users.

An icon list resource defines one icon, which the Finder uses to display the file it represents. If you examine the compiled version of an icon list resource, as represented in Figure 7-17, you find that it contains the following elements:

- The 32-by-32 pixel black-and-white icon.
- The 32-by-32 pixel black icon mask, which shows the area covered by the black-and-white icon and any 32-by-32 pixel color versions of the icon. The Finder uses the mask to crop the icon’s outline into whatever background color or pattern is on the desktop. The Finder then draws the black-and-white icon specified in this resource—or the color icons specified in large 4-bit color icon resources or large 8-bit color icon resources—into this shape.

Figure 7-17 Structure of a compiled icon list (`'ICN#'`) resource



To create 16-by-16 pixel and color versions of the icon defined in an icon list resource (thereby supplying an entire icon family), your application must also create the following resources: a small icon list resource, a large 4-bit color icon resource, a small 4-bit color icon resource, a large 8-bit color icon resource, and a small 8-bit color icon resource. Their compiled formats are described in the next several sections; guidelines for creating them are provided in “Creating Icons for the Finder” beginning on page 7-11.

The Small Icon List Resource

A small icon list resource is one of several resources that you provide for an icon family. A small icon list resource is a resource with the resource type `'ics#'`. A small icon list resource must be marked purgeable, and it must have the same resource ID as the icon list resource that represents the file that the small icon list resource also represents.

When the user chooses by Small Icon from the View menu, the Finder displays the small black-and-white icon specified in this resource in windows if either the user has a black-and-white monitor or the application has not defined any resources for color icons; otherwise, a color version of the icon is displayed. Similarly, the small black-and-white icon or its color version appears in the Application menu after the user launches the application and in the Apple menu if the user places the application or an alias to it in the Apple Menu Items folder.

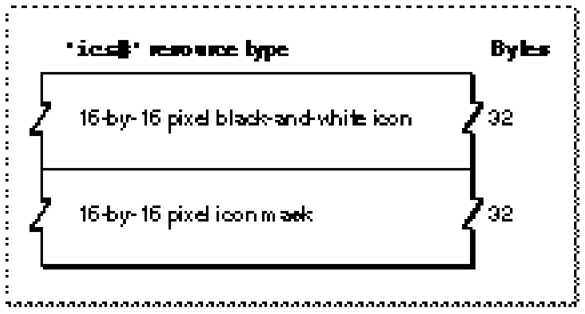
A small icon list resource is defined to be an array of two items of type `String[32]`; each bit in the first array represents a pixel in the 16-by-16 pixel icon, and each bit in the second array represents a pixel in the 16-by-16 pixel mask. You can use a high-level tool such as the ResEdit application to create small icon list resources. You can then use the DeRez decompiler to convert your small icon list resources into Rez input when necessary. See "Creating Icons for the Finder" beginning on page 7-11 for information about creating small icon list resources and other resources for representing files to users.

A small icon list resource defines one icon, which the Finder uses to display the file it represents. If you examine the compiled version of a small icon list resource, as represented in Figure 7-18, you find that it contains the following elements:

- The 16-by-16 pixel black-and-white icon for display on the desktop.
- The 16-by-16 pixel black icon mask, which shows the area covered by the icon. The Finder uses the mask to crop the icon's outline into whatever background color or pattern is on the desktop. The Finder then draws the black-and-white icon specified in this resource—or the color icons specified in the small 4-bit color icon resource or the small 8-bit color icon resource—into this shape.

The format for the compiled icon list resource is described on page 7-57; the format for the compiled small 4-bit color icon resource is described on page 7-60; and the format for the compiled small 8-bit color icon resource is described on page 7-62.

Figure 7-18 Structure of a compiled small icon list (`'ics#'`) resource



The Large 4-Bit Color Icon Resource

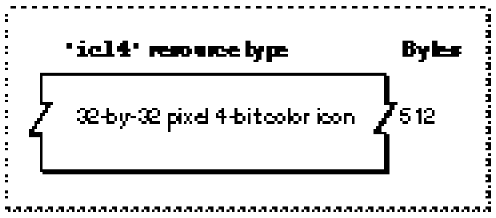
A large 4-bit color icon resource is one of several resources that you provide for an icon family. A large 4-bit color icon resource is a resource with the resource type `'icl4'`. A large 4-bit color icon resource must be marked purgeable, and it must have the same resource ID as the icon list resource that represents the file that the large 4-bit color icon resource also represents.

When the user chooses by Icon from the View menu, the Finder displays the large 4-bit color icon specified in this resource in windows if the user has a monitor displaying 4 bits of color data per pixel. Similarly, the large 4-bit color icon appears in the Application menu after the user launches the application and in the Apple menu if the user places the application or an alias to it in the Apple Menu Items folder.

A large 4-bit color icon resource is defined to be of type `String[512]`; every 4 bits in the string represent a pixel in the 32-by-32 pixel icon. You can use a high-level tool such as the ResEdit application to create large 4-bit color icon resources. You can then use the DeRez decompiler to convert your large 4-bit color icon resources into Rez input when necessary. See “Creating Icons for the Finder” beginning on page 7-11 for information about creating resources for visually representing files.

A large 4-bit color icon resource defines one icon, which the Finder uses to display the file it represents. If you examine the compiled version of a large 4-bit color icon resource, as represented in Figure 7-18, you find that it contains only the 32-by-32 pixel 4-bit color icon for display by the Finder. This resource does not specify a mask for the icon; instead, the Finder uses the mask specified for the icon list resource with the same resource ID number as this resource.

Figure 7-19 Structure of a compiled large 4-bit color icon (`'icl4'`) resource



The format for the compiled icon list resource is described on page 7-57.

The Small 4-Bit Color Icon Resource

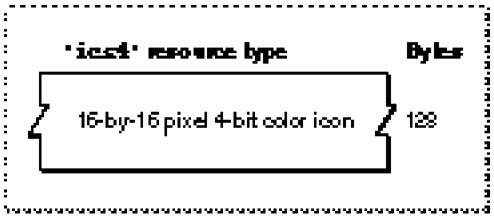
A small 4-bit color icon resource is one of several resources that you provide for an icon family. A small 4-bit color icon resource is a resource with the resource type `'ics4'`. A small 4-bit color icon resource must be marked purgeable, and it must have the same resource ID as the icon list resource that represents the file that the small 4-bit color icon resource also represents.

When the user chooses by Small Icon from the View menu, the Finder displays the small 4-bit color icon specified in this resource in windows if the user has a monitor displaying 4 bits of color data per pixel. Similarly, the small 4-bit color icon appears in the Application menu after the user launches the application and in the Apple menu if the user places the application or an alias to it in the Apple Menu Items folder.

A small 4-bit color icon resource is defined to be of type `String[128]`; every 4 bits in the string represent a pixel in the 16-by-16 pixel icon. You can use a high-level tool such as the ResEdit application to create small 4-bit color icon resources. You can then use the DeRez decompiler to convert your small 4-bit color icon resources into Rez input when necessary. See “Creating Icons for the Finder” beginning on page 7-11 for information about creating resources for representing files to users.

A small 4-bit color icon resource defines one icon, which the Finder uses to display the file it represents. If you examine the compiled version of a small 4-bit color icon resource, as represented in Figure 7-18, you find that it contains only the 16-by-16 pixel 4-bit color icon for display by the Finder. This resource does not specify a mask for the icon; instead, the Finder uses the mask specified for the small icon list resource with the same resource ID number as this resource.

Figure 7-20 Structure of a compiled small 4-bit color icon (`'ics4'`) resource



The format for the compiled icon list resource is described on page 7-57. The format for the compiled small icon list resource is described on page 7-58.

The Large 8-Bit Color Icon Resource

A large 8-bit color icon resource is one of several resources that you provide for an icon family. A large 8-bit color icon resource is a resource with the resource type `'icl8'`. A large 8-bit color icon resource must be marked purgeable, and it must have the same resource ID as the icon list resource that represents the file that the large 8-bit color icon resource also represents.

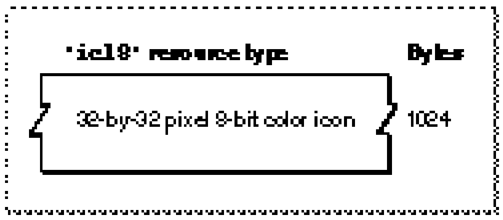
When the user chooses by Icon from the View menu, the Finder displays the large 8-bit color icon specified in this resource in windows if the user has a monitor displaying 8 bits of color data per pixel. Similarly, the large 8-bit color icon appears in the Application menu after the user launches the application and in the Apple menu if the user places the application or an alias to it in the Apple Menu Items folder.

A large 8-bit color icon resource is defined to be of type `String[1024]`; every byte in the string represents a pixel in the 32-by-32 pixel icon. You can use a high-level tool such as the ResEdit application to create large 8-bit color icon resources. You can then use the DeRez decompiler to convert your large 8-bit color icon resources into Rez input when necessary. See “Creating Icons for the Finder” beginning on page 7-11 for information about creating resources for visually representing files.

A large 8-bit color icon resource defines one icon, which the Finder uses to display the file it represents. If you examine the compiled version of a large 8-bit color icon resource, as represented in Figure 7-21, you find that it contains only the 32-by-32 pixel 8-bit color icon for display by the Finder. This resource does not specify a mask for the icon; instead, the Finder uses the mask specified for the icon list resource with the same resource ID number as this resource.

The format for the compiled icon list resource is described on page 7-57.

Figure 7-21 Structure of a compiled large 8-bit color icon (`'icl8'`) resource



The Small 8-Bit Color Icon Resource

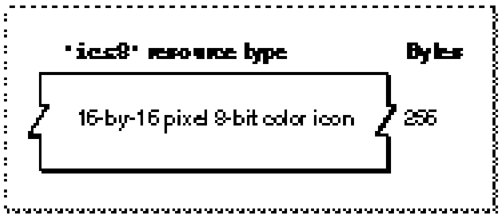
A small 8-bit color icon resource is one of several resources that you provide for an icon family. A small 8-bit color icon resource is a resource with the resource type `'ics8'`. A small 8-bit color icon resource must be marked purgeable, and it must have the same resource ID as the icon list resource that represents the file that the small 8-bit color icon resource also represents.

When the user chooses by Small Icon from the View menu, the Finder displays the small 8-bit color icon specified in this resource in windows if the user has a monitor displaying 8 bits of color data per pixel. Similarly, the small 8-bit color icon appears in the Application menu after the user launches the application and in the Apple menu if the user places the application or an alias to it in the Apple Menu Items folder.

A small 8-bit color icon resource is defined to be of type `String[256]`; every byte in the string represents a pixel in the 16-by-16 pixel icon. You can use a high-level tool such as the ResEdit application to create small 8-bit color icon resources. You can then use the DeRez decompiler to convert your small 8-bit color icon resources into Rez input when necessary. See “Creating Icons for the Finder” beginning on page 7-11 for information about creating resources for visually representing files.

A small 8-bit color icon resource defines one icon, which the Finder uses to display the file it represents. If you examine the compiled version of a small 8-bit color icon resource, as represented in Figure 7-22, you find that it contains only the 16-by-16 pixel 8-bit color icon for display by the Finder. This resource does not specify a mask for the icon; instead, the Finder uses the mask specified for the small icon list resource with the same resource ID number as this resource.

Figure 7-22 Structure of a compiled small 8-bit color icon (`'ics8'`) resource



The format for the compiled icon list resource is described on page 7-57. The format for the compiled small icon list resource is described on page 7-58.

The Icon Resource

When you want to display a 32-by-32 pixel black-and-white icon within some element of your application (such as within a menu, an alert box, or a dialog box), you can create an icon resource. An icon resource is a resource with the resource type `'ICON'`. All icon resources must be marked purgeable, and they must have resource IDs greater than 128.

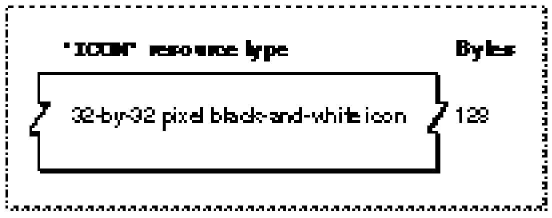
Using icon resources, you can create icons similar to the ones the Finder uses to display your application's files on the desktop; however, unlike the resource types previously described in this section, the Finder does *not* use or display any resources that you create of type `'ICON'`. Instead, your application uses icon resources of type `'ICON'` to display icons from within your application. Icon resources are described here for completeness and to mitigate the confusion that sometimes arises concerning icon (`'ICON'`) resources (which your application creates for its own use), icon list (`'ICN#'`) resources, and the other previously described resources necessary for defining an icon family (which your application creates for the Finder's use).

See "Creating Icons for the Finder" beginning on page 7-11 for additional information about creating icon list resources and other resources for representing files to users.

Generally, you use icon resources in menus and dialog boxes, as described in the chapters "Menu Manager" and "Dialog Manager" in this book. If you provide a color icon (`'cicn'`) resource with the same resource ID as the icon resource, the Menu Manager and the Dialog Manager display the color icons instead of the black-and-white icons for users with color monitors. (For example, the color alert box in Plate 2 specifies a resource of type `'cicn'` for the color icon in the upper-left corner of the alert box.)

An icon resource is defined to be of type `String[128]`; each bit represents a pixel in the 32-by-32 pixel icon. As illustrated in Figure 7-23 on the next page, an icon resource resembles an icon list resource without the array that specifies the icon's mask. You can use a high-level tool such as the ResEdit application to create icon resources. You can then use the DeRez decompiler to convert your icon resources into Rez input when necessary.

Figure 7-23 Structure of a compiled icon (`'ICON'`) resource



The Color Icon Resource

When you want to display a color icon within some element of your application (such as within a menu, an alert box, or a dialog box), you can create a color icon resource. A color icon resource is a resource with the resource type `'cicn'`. All color icon resources must be marked purgeable, and they must have resource IDs greater than 128.

Using color icon resources, you can create icons similar to the ones the Finder uses to display your application's files on the desktop; however, the Finder does *not* use or display any resources that you create of type `'cicn'`. Instead, your application uses icon resources of type `'cicn'` to display icons from within your application. Color icon resources (that is, those of resource type `'cicn'`) are mentioned here to mitigate the confusion that sometimes arises concerning color icon resources (which your application creates for its own use) and the small and large 4-bit and 8-bit color icon resources (types `'ics4'`, `'icl4'`, `'ics8'`, and `'icl8'`) necessary to define an icon family (which your application creates for the Finder's use).

See "Creating Icons for the Finder" beginning on page 7-11 for information about creating an icon family that includes color icons for representing files to users.

Generally, you use color icon resources in menus, alert boxes, and dialog boxes, as described in the chapters "Menu Manager" and "Dialog Manager" in this book. If you provide a color icon (`'cicn'`) resource with the same resource ID as an icon resource (described on page 7-63), the Menu Manager and the Dialog Manager display the color icon instead of the black-and-white icon for users with color monitors.You can use a high-level tool such as the ResEdit application to create color icon resources. You can then use the DeRez decompiler to convert your color icon resources into Rez input when necessary. (For example, the color alert box in Plate 2 specifies a resource of type `'cicn'` for the color icon in the upper-left corner of the alert box.)

See *Inside Macintosh: Imaging* for more information about color icon resources.

The File Reference Resource

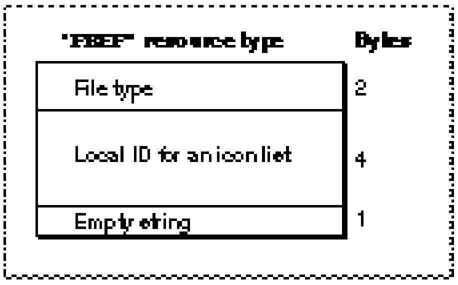
To link icons with the files types they represent and to allow users to launch your application by dragging document icons to your application icon, create a file reference resource for every icon list resource you create. A file reference resource is a resource with the resource type `'FREF'`. All file reference resources must have resource IDs greater than 128, and each must be marked purgeable.

This section describes the structure of a file reference resource after it is compiled by the Rez resource compiler. The format of a Rez input file for a file reference resource differs from its compiled output form. If you are concerned only with creating a file reference resource, see "Creating File Reference Resources" beginning on page 7-18.

If you examine a compiled version of a file reference resource, as illustrated in Figure 7-24, you find that it contains the following elements:

- File type. This is the four-character code that identifies the type of file represented by this resource. File types are described in "Giving a Signature to Your Application and a Creator and a File Type to Your Documents" beginning on page 7-8.
- Local ID. The Finder uses this number to map the file type specified in this resource to an icon list resource that is assigned the same local ID in the bundle resource. The icon list resource is described on page 7-57; the bundle resource is described in the next section.
- Empty string. This element should always contain an empty Pascal string.

Figure 7-24 Structure of a compiled file reference (`'FREF'`) resource

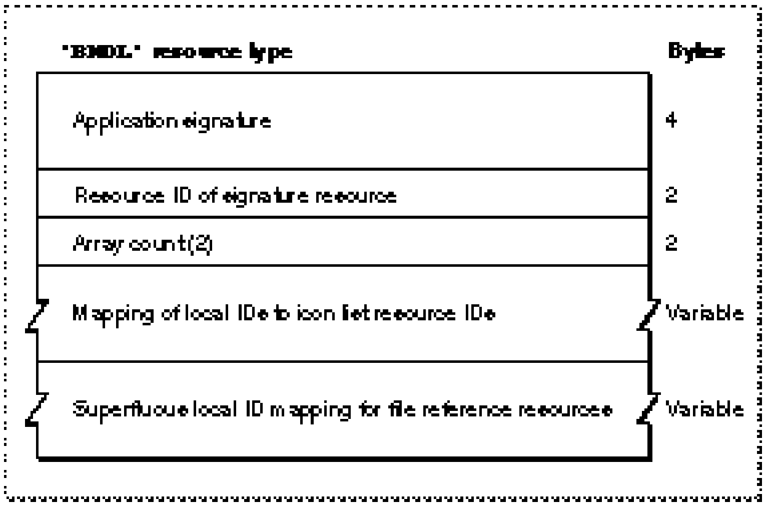


The Bundle Resource

To group together your application's signature, icon list resource, and file reference resources, create a bundle resource. A bundle resource is a resource with the resource type `'BNDL'`. All bundle resources must have resource ID numbers greater than 128, and all must be made purgeable.

This section describes the structure of the bundle resource after it is compiled by the Rez resource compiler. The format of a Rez input file for a bundle resource differs from its compiled output form. If you are concerned only with creating a bundle resource, see "Creating a Bundle Resource" beginning on page 7-20.

Figure 7-25 Structure of a compiled bundle (`'BNDL'`) resource


'BNDL' resource type
Bytes
Application signature
4
Resource ID of signature resource
2
Array count(2)
2
Mapping of local IDs to icon list resource IDs
Variable
Superfluous local ID mapping for file reference resources
Variable

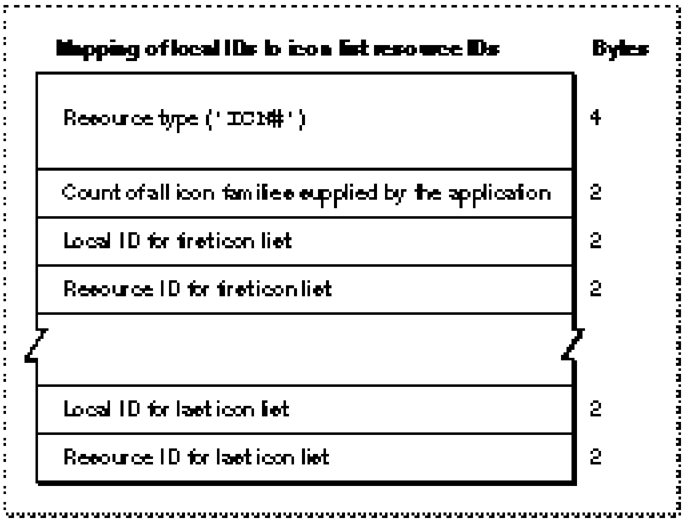
If you examine a compiled version of a file reference resource, as illustrated in Figure 7-25, you find that it contains the following elements:

- Application signature. This is the unique four-character code that identifies the application to the Finder. (Application signatures are described in “Giving a Signature to Your Application and a Creator and a File Type to Your Documents” beginning on page 7-8.)
- Resource ID of the signature resource. By convention, this should always be 0.
- Array count. This element should always contain the value 2.
- Mapping of local IDs to icon list resource IDs for all icons supplied by the application. This is illustrated in Figure 7-26.
- Superfluous local ID mapping for file reference resources. This is illustrated in Figure 7-27.

If you examine the compiled portion of a bundle resource that maps local IDs to icon list resource IDs, you find that it contains the following elements:

- Resource type. This element should always specify the resource type `'ICN#'` (that is, an icon list resource).
- Count of all the icon families supplied by the application. This is the number of local ID–to–icon list resource ID mapping pairs in the rest of this resource.
- Local ID for an icon list resource. This local ID must match the local ID assigned to the icon list resource within a file reference resource.
- Resource ID for the icon list resource assigned a local ID in the preceding element. To visually represent files of the type described in the file reference resource that contains the local ID in the preceding element, the Finder uses the black-and-white icon and mask described in this icon list resource. The Finder also uses the icons defined in the following resources with this same resource ID: small icon list resource, small 4-bit color icon resource, small 8-bit color icon resource, large 4-bit color icon resource, and large 8-bit color icon resource.

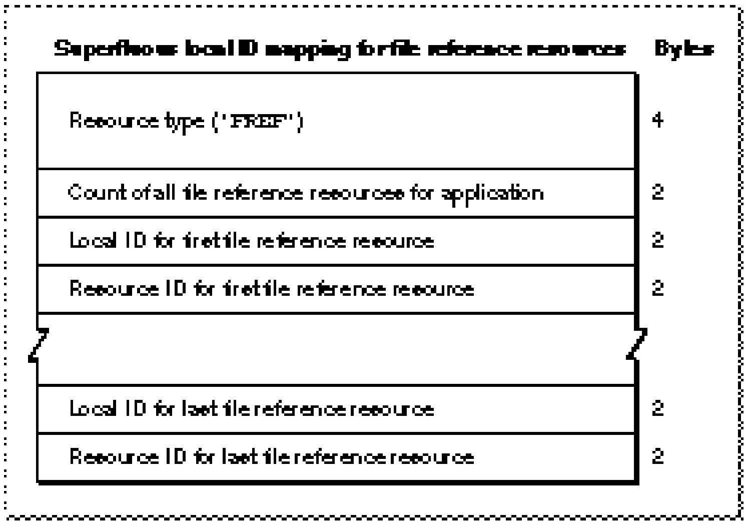
Figure 7-26 Mapping local IDs to icon list resource IDs in a bundle resource


Mapping of local IDs to icon list resource IDs
Bytes
Resource type ('ICN#')
4
Count of all icon families supplied by the application
2
Local ID for first icon list
2
Resource ID for first icon list
2
Local ID for last icon list
2
Resource ID for last icon list
2

- Local ID–to–icon list resource ID mapping pairs for the rest of the icons representing file types for an application.

Figure 7-27 illustrates the remainder of a bundle resource, which assigns local IDs to file reference resource IDs. This assignment is superfluous because the Finder doesn't map these local IDs to any other resources. This ID assignment was implemented for the earliest versions of Macintosh system software, and it remains this way today to maintain backward compatibility.

Figure 7-27 Structure of superfluous local ID mapping for file reference resources in a bundle resource


Superfluous local ID mapping for file reference resources
Bytes
Resource type ('FREF')
4
Count of all file reference resources for application
2
Local ID for first file reference resource
2
Resource ID for first file reference resource
2
Local ID for last file reference resource
2
Resource ID for last file reference resource
2

If you examine the compiled portion of the remainder of a bundle resource, you find that it contains the following elements:

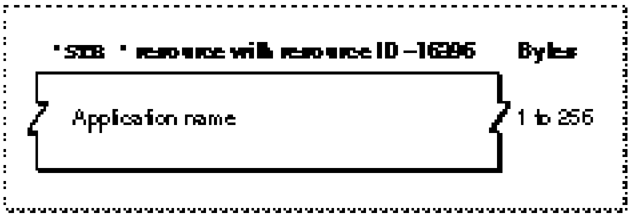
- Resource type. This element should always specify the resource type `'FREF'` (that is, a file reference resource).
- Count of all the file reference resources representing file types for an application. This is the number of local ID–to–file reference resource mapping pairs in the rest of this resource.
- Local ID for a file reference resource. The local ID can be any integer so long as no other file reference resource is given that same local ID within this resource.
- Resource ID for the file reference resource assigned a local ID in the preceding field.
- Local ID–to–file reference resource ID mapping pairs for the rest of the file reference resources that represent file types with application-supplied icons.

The Missing-Application Name String

When your application creates a document that the user can open, your application should include a missing-application name string in the resource file of the document. The missing-application name string is a resource with the resource type `'STR '`, it must have a resource ID number of –16396, and it must be made purgeable. The string resource should contain your application's name only. See "Displaying Messages When the Finder Can't Find Your Application" beginning on page 7-27 for additional information about copying this resource into the resource fork of your documents.

If you examine a compiled missing-application name string, as illustrated in Figure 7-28, you find that it consists entirely of a Pascal string that names the application that created the document. The Finder displays this string in an alert box if the user tries to open or print a document created by the application whenever the application is missing.

Figure 7-28 Structure of a compiled missing-application name string resource



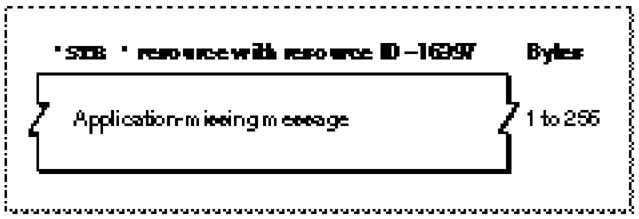
The Application-Missing Message String

When your application creates a document that your application uses but that the user cannot open (such as a preferences file), your application should set the creator of the document to a registered signature that is not the same as your or anyone else's application, and include an application-missing message string in the resource file of the document. The application-missing name string is a resource with the resource type

`'STR '`, it must have a resource ID number of –16397, and it must be made purgeable. The string resource should contain a message that explains why the user cannot open or print the document, as explained in "Displaying Messages When the Finder Can't Find Your Application" beginning on page 7-27.

If you examine a compiled application-missing message string, as illustrated in Figure 7-29, you find that it consists entirely of a Pascal string that explains why the user cannot open the document. The Finder displays this string in an alert box if the user tries to open or print a document that is given a special creator that is not used as a signature by any application file. (File creators and application signatures are explained in "Giving a Signature to Your Application and a Creator and a File Type to Your Documents" beginning on page 7-8.)

Figure 7-29 Structure of a compiled application-missing message string resource


'STR ' resource with resource ID –16397
Bytes
Application-missing message
1 to 256

The Version Resource

You can use a version resource in any file so that users can easily find out the version of the file and, if it is a part of a larger collection of files, of the entire superset of files. A version resource is a resource with the resource type `'vers'`. The version resource with a resource ID number of 1 specifies the version of an individual file; the version resource with a resource ID number of 2 specifies the superset of files to which the individual file belongs.

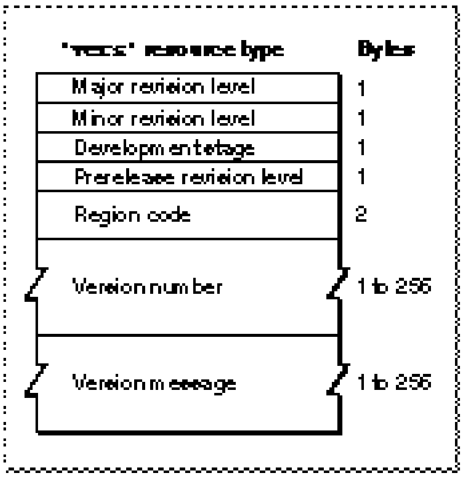
If your application does not contain a version resource with a resource ID number of 1, the Finder displays the string from your application's signature resource (described in "Giving a Signature to Your Application and a Creator and a File Type to Your Documents" beginning on page 7-8) in the information window when the user chooses the Get Info command from the File menu.

This section describes the structure of this resource after it is compiled by the Rez resource compiler. The format of a Rez input file for a version resource differs from its compiled output form. If you are concerned only with creating version resources, see "Providing Version Resources" beginning on page 7-31.

If you examine a compiled version of version resource, as illustrated in Figure 7-30 on page 7-70, you find that it contains the following elements:

- Major revision level in binary-coded decimal format.
- Minor revision level in binary-coded decimal format.

Figure 7-30 Format of a compiled version (`'vers'`) resource



- Development stage. The values that can appear in this field, as well as the constants that can be used to specify them in a Rez input file, are the following:

Value	Constant	Description
0x20	`development`	Prealpha file
0x40	`alpha`	Alpha file
0x60	`beta`	Beta file
0x80	`release`	Released file

- Prerelease revision level. This number specifies the version if the software is still prerelease.
- Region code. This identifies the script system for which this version of the software is intended. See the chapter “Script Manager” in *Inside Macintosh: Text* for information about the values represented by the various region codes that can be specified here.
- Version number. This Pascal string identifies the version number of the software. When the user opens the Views control panel, clicks the Show version box, and then chooses any command from the View menu other than by Icon or by Small Icon, the Finder window containing this application displays this string.
- Version message. This Pascal string identifies the version number and either a company copyright for a file or a product name for a superset of files. When the user selects this file and chooses the Get Info command, the Finder displays this string in the information window as follows:
  - For a version resource with a resource ID number of 1, this string is displayed in the version field of the information window.
  - For a version resource with a resource ID number of 2, this string is displayed beneath the file’s name next to the file’s icon at the top of the information window.

Summary of the Finder Interface

Pascal Summary

Constants

```
CONST {Gestalt selectors}
      gestaltFindFolderAttr         = 'fold';   {selector for FindFolder}

      {interpreting Gestalt selector responses}
      gestaltFindFolderPresent      = 0;        {if this bit is set, }
                                                { FindFolder is present}
      {for custom icons}
      kCustomIconResource           = –16455;   {resource ID for }
                                                { custom icon}
      {for Finder flags}
      fHasBundle                    = 8192;     {set if file has 'BNDL'}
      fInvisible                    = 16384;    {set if icon is invisible}
      kIsOnDesk                     = $1;       {unused and reserved in }
                                                { System 7}
      kColor                        = $E;       {three bits of color coding}
      kIsShared                     = $40;      {file can be executed by }
                                                { multiple users }
                                                { simultaneously}
      kHasBeenInited                 = $100;    {file info is in desktop }
                                                { database}
      kHasCustomIcon                 = $400;    {file or directory has a }
                                                { customized icon}
      kIsStationery                  = $800;    {file is a stationery pad}
      kNameLocked                    = $1000;   {file or directory can't }
                                                { be renamed from Finder, }
                                                { and icon can't be changed}
      kHasBundle                     = $2000;   {file has a bundle resource}
      kIsInvisible                   = $4000;   {file or directory is }
                                                { invisible from Finder & }
                                                { from Standard File }
                                                { Package dialog boxes}
      kIsAlias                       = $8000;   {file is an alias file}
```

```
{for FindFolder}
kOnSystemDisk                 = $8000;    {use vRefNum for the }
                                          { boot disk}
kCreateFolder                 = TRUE;     {create folder if it }
                                          { doesn't exist}
kDontCreateFolder             = FALSE;    {don't create folder}

{for special folder types}
kSystemFolderType             = 'macs';   {System Folder}
kDesktopFolderType            = 'desk';   {Desktop Folder}
kTrashFolderType              = 'trsh';   {single-user Trash}
kWhereToEmptyTrashFolderType  = 'empt';   {shared Trash on network}
kPrintMonitorDocsFolderType   = 'prnt';   {PrintMonitor Documents}
kStartupFolderType            = 'strt';   {Startup Items}
kFontsFolderType              = 'font';   {Fonts}
kAppleMenuFolderType          = 'amnu';   {Apple Menu Items}
kControlPanelFolderType       = 'ctrl';   {Control Panels}
kExtensionFolderType          = 'extn';   {Extensions}
kPreferencesFolderType        = 'pref';   {Preferences}
kTemporaryFolderType          = 'temp';   {Temporary Items}
{alias types}
kContainerFolderAliasType     = 'fdrp';   {folder alias}
kContainerTrashAliasType      = 'trsh';   {Trash alias}
kContainerHardDiskAliasType   = 'hdsk';   {hard disk alias}
kContainerFloppyAliasType     = 'flpy';   {floppy disk alias}
kContainerServerAliasType     = 'srvr';   {server alias}
kApplicationAliasType         = 'adrp';   {application alias}
kContainerAliasType           = 'drop';   {all other containers}
kSystemFolderAliasType        = 'fasy';   {System Folder alias}
kAppleMenuFolderAliasType     = 'faam';   {Apple Menu Items folder }
                                          { alias}
kStartupFolderAliasType       = 'fast';   {Startup Items folder alias}
kPrintMonitorDocsFolderAliasType
                              = 'fapn';   {PrintMonitor Documents }
                                          { folder alias}
kPreferencesFolderAliasType   = 'fapf';   {Preferences folder alias}
kControlPanelFolderAliasType  = 'fact';   {Control Panels folder alias}
kExtensionFolderAliasType     = 'faex';   {Extensions folder alias}
kExportedFolderAliasType      = 'faet';   {export folder alias}
kDropFolderAliasType          = 'fadr';   {drop folder alias}
kSharedFolderAliasType        = 'fash';   {shared folder alias}
kMountedFolderAliasType       = 'famn';   {mounted folder alias}
```

Data Types

```
TYPE  {Finder information records in the volume catalog file}
      FInfo =
      RECORD
         fdType:           OSType;        {file type}
         fdCreator:        OSType;        {file creator}
         fdFlags:          Integer;       {Finder flags}
         fdLocation:       Point;         {file's location in window}
         fdFldr:           Integer;       {directory that contains file}
      END;

      FXInfo =
      RECORD
         fdIconID:         Integer;       {icon ID}
         fdUnused:         ARRAY[1..3] OF Integer;
                                          {unused but reserved 6 bytes}
         fdScript:         SignedByte;    {script flag and code}
         fdXFlags:         SignedByte;    {reserved}
         fdComment:        Integer;       {comment ID}
         fdPutAway:        LongInt;       {home directory ID}
      END;

      DInfo =
      RECORD
         frRect:        Rect;             {folder's window rectangle}
         frFlags:       Integer;          {flags}
         frLocation:    Point;            {folder's location in window}
         frView:        Integer;          {folder's view}
      END;

      DXInfo =
      RECORD
         frScroll:      Point;            {scroll position}
         frOpenChain:   LongInt;          {dir ID chain of open folders}
         frScript:      SignedByte;       {script flag and code}
         frXFlags:      SignedByte;       {reserved}
         frComment:     Integer;          {comment ID}
         frPutAway:     LongInt;          {directory ID}
      END;
```

Routines

Resolving Alias Files

```
FUNCTION ResolveAliasFile    (VAR theSpec: FSSpec;
                              resolveAliasChains: Boolean;
                              VAR targetIsFolder: Boolean;
                              VAR wasAliased: Boolean): OSErr;
```

Finding Directories

```
FUNCTION FindFolder          (vRefNum: Integer; folderType: OSType;
                              createFolder: Boolean;
                              VAR foundVRefNum: Integer;
                              VAR foundDirID: LongInt): OSErr;
```

C Summary

Constants

```
enum {
      /*Gestalt selectors*/
      #define gestaltFindFolderAttr  'fold'     /*selector for FindFolder*/

      /*interpreting Gestalt selector responses*/
      gestaltFindFolderPresent      = 0         /*if this bit is set, */
                                                /* FindFolder is present*/
};
      /*for custom icons*/
#define kCustomIconResource          –16455     /*resource ID for */
                                                /* custom icon*/

      /*Finder flags*/
#define  kIsOnDesk                  0x1         /*unused and reserved in */
                                                /* System 7*/
#define  kColor                     0xE         /*3 bits of color coding*/
#define  kIsShared                  0x40        /*file can be executed by */
                                                /* multiple users */
                                                /* simultaneously*/
#define  kHasBeenInited             0x100       /*file info is in desktop */
                                                /* database*/
#define  kHasCustomIcon             0x400       /*file or directory has a */
                                                /* customized icon*/
```

```
#define  kIsStationary              0x800       /*file is a stationery pad*/
#define  kNameLocked                0x1000      /*file or directory can't */
                                                /* be renamed from the */
                                                /* Finder, and icon can't */
                                                /* be changed*/
#define  kHasBundle                 0x2000      /*file has a bundle */
                                                /* resource*/
#define  kIsInvisible               0x4000      /*file or directory is */
                                                /* invisible from Finder */
                                                /* & from Standard File */
                                                /* Package dialog boxes*/
#define  kIsAlias                   0x8000      /*file is an alias file*/

enum {
      /*for Finder flags*/
      fHasBundle                    = 8192,     /*set if file has 'BNDL'*/
      fInvisible                    = 16384     /*set if icon is invisible*/
};
enum {
      /*for FindFolder*/
      kOnSystemDisk                 = 0x8000    /*use vRefNum for the */
                                                /* boot disk*/
      #define kCreateFolder            true     /*create folder if it */
                                                /* doesn't exist*/
      #define kDontCreateFolder        false    /*don't create folder*/

      /*for special folder types*/
      #define kSystemFolderType      'macs'     /*System Folder*/
      #define kDesktopFolderType     'desk'     /*Desktop Folder*/
      #define kTrashFolderType       'trsh'     /*single-user Trash*/
      #define kWhereToEmptyTrashFolderType
                                     'empt'     /*shared Trash*/
      #define kPrintMonitorDocsFolderType
                                     'prnt'     /*PrintMonitor Documents*/
      #define kStartupFolderType     'strt'     /*Startup Items*/
      #define kFontsFolderType       'font'     /*Fonts*/
      #define kAppleMenuFolderType   'amnu'     /*Apple Menu Items*/
      #define kControlPanelFolderType 'ctrl'    /*Control Panels*/
      #define kExtensionFolderType   'extn'     /*Extensions*/
      #define kPreferencesFolderType 'pref'     /*Preferences*/
      #define kTemporaryFolderType   'temp'     /*Temporary Items*/
};
      /*for alias types*/
#define kContainerFolderAliasType    'fdrp'     /*folder alias*/
#define kContainerTrashAliasType     'trsh'     /*Trash alias*/
```

```
#define kContainerHardDiskAliasType  'hdsk'     /*hard disk alias*/
#define kContainerFloppyAliasType    'flpy'     /*floppy disk alias*/
#define kContainerServerAliasType    'srvr'     /*server alias*/
#define kApplicationAliasType        'adrp'     /*application alias*/
#define kContainerAliasType          'drop'     /*all other containers*/
#define kSystemFolderAliasType       'fasy'     /*System Folder alias*/
#define kAppleMenuFolderAliasType    'faam'     /*Apple Menu Items folder */
                                                /* alias*/
#define kStartupFolderAliasType      'fast'     /*Startup Items folder */
                                                /* alias*/
#define kPrintMonitorDocsFolderAliasType
                                     'fapn'     /*PrintMonitor Documents */
                                                /* folder alias*/
#define kPreferencesFolderAliasType  'fapf'     /*Preferences folder alias*/
#define kControlPanelFolderAliasType 'fact'     /*Control Panels fldr alias*/
#define kExtensionFolderAliasType    'faex'     /*Extensions folder alias*/
#define kExportedFolderAliasType     'faet'     /*export folder alias*/
#define kDropFolderAliasType         'fadr'     /*drop folder alias*/
#define kSharedFolderAliasType       'fash'     /*shared folder alias*/
#define kMountedFolderAliasType      'famn'     /*mounted folder alias*/
```

Data Types

```
struct FInfo {       /*Finder information records in the catalog file*/
      OSType            fdType;        /*file type*/
      OSType            fdCreator;     /*file creator*/
      unsigned short    fdFlags;       /*Finder flags*/
      Point             fdLocation;    /*file's location in window*/
      short             fdFldr;        /*directory that contains file*/
};

struct FXInfo {
      short             fdIconID;      /*icon ID*/
      short             fdUnused[3];   /*unused but reserved 6 bytes*/
      char              fdScript;      /*script flag and code*/
      char              fdXFlags;      /*reserved*/
      short             fdComment;     /*comment ID*/
      long              fdPutAway;     /*home directory ID*/
};
```

```
struct DInfo {
      Rect              frRect;        /*folder's window rectangle*/
      unsigned short    frFlags;       /*flags*/
      Point             frLocation;    /*folder's location in window*/
      short             frView;        /*folder's view*/
};

struct DXInfo {
      Point          frScroll;         /*scroll position*/
      long           frOpenChain;      /*directory ID chain of open folders*/
      char           frScript;         /*script flag and code*/
      char           frXFlags;         /*reserved*/
      short          frComment;        /*comment ID*/
      long           frPutAway;        /*directory ID*/
};
```

Routines

Resolving Alias Files

```
pascal OSErr ResolveAliasFile
                            (FSSpec *theSpec, Boolean resolveAliasChains,
                             Boolean *targetIsFolder, Boolean *wasAliased);
```

Finding Directories

```
pascal OSErr FindFolder     (short vRefNum, OSType folderType,
                             Boolean createFolder, short *foundVRefNum,
                             long *foundDirID);
```

Assembly-Language Summary

Data Structures

FInfo Data Structure

0	`fdType`	long	file type
4	`fdCreator`	long	file creator
8	`fdFlags`	word	Finder flags
10	`fdLocation`	long	file’s location in window
14	`fdFldr`	word	directory that contains file

FXInfo Data Structure

0	`fdIconID`	word	icon ID
2	`fdUnused`	6 bytes	reserved
8	`fdScript`	1 byte	script flag and code
9	`fdXFlags`	1 byte	reserved
10	`fdComment`	word	comment ID
12	`fdPutAway`	long	home directory ID

DInfo Data Structure

0	`frRect`	8 bytes	folder's window rectangle
8	`frFlags`	word	flags
10	`frLocation`	long	folder's location in window
14	`frView`	word	folder's view

DXInfo Data Structure

0	`frScroll`	long	scroll position
4	`frOpenChain`	long	directory ID chain of open folders
8	`frScript`	1 byte	script flag and code
9	`frXFlags`	1 byte	reserved
10	`frComment`	word	comment ID
12	`frPutAway`	long	directory ID

Result Codes

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`fnfErr`	–43	For `FindFolder`: Type not found in `'fld#'` resource, or disk doesn't have System Folder support or System Folder in volume header, or disk does not have desktop database support for Desktop Folder—in all cases, folder not found For `ResolveAliasFile`: Target not found, but volume and parent directory found and `theSpec` parameter contains a valid file system specification record
`dupFNErr`	–48	File found instead of folder
`dirNFErr`	–120	Parent directory not found

Glossary

action procedure A procedure that performs an action in response to the user holding the mouse button down while the cursor is in a control.

activate event A type of event that indicates that a window is becoming active or inactive. Each activate event specifies the window to be changed and the direction of the change (that is, whether it's becoming active or becoming inactive).

active control A control in which the Control Manager responds to a user's mouse actions by providing visual feedback.

active window The frontmost window on the desktop, the one in which the user is currently working. The active window is designated by racing stripes in the title bar, active controls, and highlighted selections.

A5 world An area of memory in an application's partition that contains the QuickDraw gloabl variables, the application global variables, the application parameters, and the jump table—all of which are accessed through the A5 register.

alert An alert sound, an alert box, or both. Alerts warn the user of an unusual or a potentially undesirable situation occurring within an application. See also **alert box** and **alert sound.**

alert box A window that an application displays on the screen to warn the user or to report an error to the user. An alert box typically consists of text describing the situation and buttons that require the user to acknowledge or rectify the problem. An alert box may or may not be accompanied by an alert sound. See also **caution alert, note alert,** and **stop alert.**

alert color table resource A resource (of type `'actb'`) that lets an application display an alert box using colors other than the system's default window colors.

alert resource A resource (of type `'ALRT'`) that specifies alert sounds, a display rectangle, and an item list for an alert box.

alert sound An audible signal from the Macintosh speaker that warns the user of an unusual or a potentially undesirable situation occurring within an application. An alert sound may or may not be accompanied by an alert box.

alias An object that represents another file, directory, or volume.

alias file A file that contains a record that points to another file, directory, or volume. An alias file is displayed by the Finder as an alias.

alias record A data structure created by the Alias Manager to identify a file, directory, or volume.

alias target The file, directory, or volume described by the alias record.

Apple event A high-level event whose structure and interpretation are determined by the Apple Event Interprocess Messaging Protocol.

Apple Menu Items folder A directory located in the System Folder for storing desk accessories, applications, folders, and aliases that the user wants to display in and access from the Apple menu.

application heap An area of memory in the application heap zone in which memory is dynamically allocated and released on demand. The heap contains the application's `'CODE'` segment 1, data structures, resource map, and other code segments as needed.

application partition A partition of memory reserved for use by an application. The application partition consists of free space, the application heap, the application's stack, and the application's A5 world.

auto-key event An event indicating that a key is still down after a certain amount of time has elapsed.

auxiliary window record A data structure that the Window Manager uses to tie together a list of windows and their corresponding window color information tables.

background process A process that isn't currently interacting with the user. Compare **foreground process.**

bundle bit A flag in a file's Finder information record that informs the Finder that a bundle (`'BNDL'`) resource exists for the file. A file's Finder information record is stored in a volume's catalog file. The Finder uses the information in the bundle resource to associate icons with the file.

button A control that appears on the screen as a rounded rectangle with a title centered inside. When the user clicks a button, the application performs the action described by the button's title. Button actions are usually performed instantaneously. Examples include completing operations defined by a dialog box and acknowledging an error message in an alert box.

catalog file A special file, located on a volume, that contains information about the hierarchical organization of files and folders on that volume.

caution alert An alert box that warns the user of an operation that may have undesirable results if it's allowed to continue. A caution alert gives the user the choice of continuing the action (by clicking the OK button) or stopping the action (by clicking the Cancel button). A caution alert is identified by an icon bearing an exclamation point in the upper-left corner of the alert box. See also **note alert** and **stop alert.**

character code A value that represents a particular character. The character code that is generated depends on the virtual key code and the state of the modifier keys. In the Roman script system, character codes are specified in the extended version of ASCII (the American Standard Code for Information Interchange).

checkbox A control that appears onscreen as a small square with an accompanying title. A checkbox displays one of two settings: on (indicated by an X inside the box) or off. When the user clicks a checkbox, the application reverses its setting. See also **radio button.**

close box The small white box on the left side of the title bar of an active window. Clicking it closes the window.

close region The area occupied by a window's close box. See also **close box.**

Command-key equivalent Refers specifically to a keyboard equivalent that the user invokes by holding down the Command key and pressing another key (other than a modifier key) at the same time.

content region The part of a window in which the contents of a document, the size box, and the window controls (including the scroll bars) are displayed.

context The information about a process maintained by the Process Manager. This information includes the current state of the process, the address and size of its partition, its type, its creator, a copy of its low-memory global variables, information about its `'SIZE'` resource, and a process serial number.

control An onscreen object that the user can manipulate with the mouse. By manipulating a control, the user can take an immediate action or change a setting to modify a future action.

control color table In an item color table resource, a specification for the colors used to draw the various parts of a control.

control definition function A function that defines the appearance and behavior of a control. A control definition function, for example, draws the control. See also **standard control definition functions.**

control definition ID A number passed to control-creation routines to indicate the type of control. It consists of the control definition function's resource ID and a variation code.

control list A series of entries pointing to the descriptions of the controls associated with the window.

Control Manager A collection of routines that applications use to create and manipulate controls, especially those in windows.

Control Panels folder A directory located in the System Folder for storing control panels, which allow users to modify the work environment of their Macintosh computer.

control record A data structure of type `ControlRecord`, which the Control Manager uses to store all the information it needs for its operations on a control.

current menu list A data structure that contains handles to the menu records of all menus in the current menu bar and the menu records of any submenus or pop-up menus that an application inserts into the list.

current process The process that is currently executing and whose A5 world is valid; this process can be in the background or the foreground.

cursor Any 256-bit image, defined by a 16-by-16 bit square. The mouse driver displays the current cursor and maps the movement of the mouse to relative locations on the screen as the user moves the mouse.

custom alert box An alert box whose upper-left corner contains blank space or displays an icon other than those used by caution alerts, stop alerts, or note alerts.

customized icon An icon created by the user or by an application and stored with a resource ID of –16455 in the resource fork of a file. A file with a customized icon has the `hasCustomIcon` bit set in its Finder flags field.

data fork The part of a file that contains data accessed using the File Manager. The data usually corresponds to data entered by the user; the application creating a file can store and interpret the data in the data fork in whatever manner is appropriate.

default button In an alert box or a dialog box, the button whose action is invoked when the user presses the Return key or the Enter key. The Dialog Manager automatically draws a bold outline around the default button in alert boxes; applications should draw a bold outline around the default button in dialog boxes. The default button should invoke the preferred action, which, whenever possible, should be a “safe” action—that is, one that doesn’t cause loss of data.

desktop The working environment displayed on the Macintosh computer: the gray background area on the screen.

desktop database A Finder-maintained database of icons, file types, applications, version data, and comments for all volumes over 2 MB. Compare **Desktop file.**

Desktop file A resource file in which the Finder stores icons, file types, applications, version data, and comments for all volumes less than 2 MB. Compare **desktop database.**

Desktop Folder A directory, located at the root level of each volume, used by the Finder for storing information about the icons that appear on the desktop area of the screen. The Desktop Folder is invisible to the user. What the user sees onscreen is the union of the contents of Desktop Folders for all mounted volumes.

dial A control, similar to a scroll bar, that graphically represents the ranges of values that a user can set or that simply displays the value, magnitude, or position of something, typically in some pseudo-analog form.

dialog box A window that an application displays on the screen to solicit information from the user before the application carries out the user’s command. See also **modal dialog box, modeless dialog box,** and **movable modal dialog box.**

dialog color table resource A resource (of type `'dctb'`) that lets an application display a dialog box using colors other than the system’s default window colors.

Dialog Manager A collection of routines that applications use to implement alerts and dialog boxes.

dialog record A data structure of type `DialogRecord` that the Dialog Manager uses to create dialog boxes and alerts.

dialog resource A resource (of type `'DLOG'`) that specifies the window type, display rectangle, and item list for a dialog box.

disabled item In an alert box or a dialog box, an item for which the Dialog Manager does not report user events. An example of a disabled item is static text, which typically does not respond to clicks.

disk-inserted event An event indicating that a disk has been inserted into a disk drive.

display rectangle A rectangle that defines the size and location of an item in an alert box or a dialog box. The display rectangle is specified in an item list and uses coordinates local to the alert box or dialog box.

divider A gray line used in menus to separate groups of menu items.

document window A window in which the user enters text, draws graphics, or otherwise enters or manipulates data.

drag region The area occupied by a window's title bar, except for the close box and zoom box. The user can move a window on the desktop by dragging the drag region.

edition The data written to an edition container by a publisher. A publisher writes data to an edition whenever a user saves a document that contains a publisher, and subscribers in other documents may read the data from the edition whenever it is updated.

enabled item In an alert box or a dialog box, an item for which the Dialog Manager reports user events. For example, the Dialog Manager reports clicks in an enabled OK button.

event The means by which the Event Manager communicates information about user actions, changes in the processing status of the application, and other occurrences that require a response from the application.

event filter function An application-defined routine that supplements the Dialog Manager's ability to handle events—for example, an event filter function can test for disk-inserted events and can allow background applications to receive update events.

Event Manager The collection of routines that an application can use to receive information about actions performed by the user, to receive notice of changes in the processing status of the application, and to communicate with other applications.

event mask An integer with one bit position for each event type. You specify an event mask as a parameter to Event Manager routines to specify the event types you want your application to receive, thereby disabling (or "masking out") the events you are not interested in receiving.

event record A data structure of type `EventRecord` that your application uses when retrieving information about an event. The Event Manager returns, in an event record, information about what type of event occurred (a mouse click or keypress, for example) and additional information associated with the event.

Extensions folder A directory located in the System Folder for storing system extension files such as printer and network drivers and files of types `'INIT'`, `'scri'`, and `'appe'`.

file A named, ordered sequence of bytes stored on a Macintosh volume, divided into a data fork and a resource fork.

Finder An application that works with the system software to keep track of files and manage the user's desktop display.

Fonts folder A directory located in the System Folder for storing fonts.

foreground process The process currently interacting with the user; it appears to the user as the active application. The foreground process displays its menu bar, and its windows are in front of the windows of other applications. Compare **background process.**

frame The part of a window drawn automatically by the Window Manager, namely, the title bar, including the close box and zoom box, and the window's outline.

global coordinate system The coordinate system that represents all potential QuickDraw drawing space. The origin of the global coordinate system—that is, the point (0,0)— is at the upper-left corner of the main screen. Compare **local coordinate system.**

graphics port A complete, individual drawing environment with an independent coordinate system. Each window is drawn in a graphics port.

gray area The area within a scroll bar, excluding the scroll arrows and the scroll box. When the user clicks the gray area of a scroll bar, the application moves the displayed area of the document by an entire window less one line (or column, row, or character).

gray region A region that represents all available desktop area—that is, a collection of rounded-corner rectangles representing the display areas of all monitors available to a computer.

grow image An outline of a window's new frame, drawn on the screen while the user is resizing the window with the size box.

help balloon A rounded-rectangle window that contains explanatory information for the user. With tips pointing at the objects they annotate, help balloons look like bubbles used for dialog in comic strips. Help balloons are turned on by the user from the Help menu; when Balloon Help assistance is on, a help balloon appears whenever the user moves the cursor over an area that is associated with it.

hierarchical menu A menu to which a submenu is attached.

high-level event An event sent from one application to another requesting transfer of information or performance of some action.

high-level event queue A separate queue that the Event Manager maintains to store high-level events transmitted to an application. The Event Manager maintains a high-level event queue for each open application capable of receiving high-level events.

hot spot A point that the mouse driver uses to align the cursor with the mouse location.

icon An image that represents an object, a concept, or a message.

icon family The set of icons that represent an object—such as an application or a document—displayed by the Finder. An entire icon family consists of large (32-by-32 pixel) and small (16-by-16 pixel) icons, each with a mask, and each available in three different versions of color: black and white, 4 bits of color data per pixel, and 8 bits of color data per pixel.

inactive control A control that has no meaning or effect in the current context—for example, the scroll bars in an empty window. The Control Manager dims inactive controls or otherwise visually indicates their inactive state.

inactive window A window in which the user is not working.

indicator A moving part in a dial or slider control. A user moves an indicator to set a value, and an application moves it to indicate the current setting of the control. In a scroll bar, the scroll box is the indicator.

item color table resource A resource (of type `'ictb'`) that an application can use to display an alert box or a dialog box with items using a typeface, font style, font size, or colors other than the system's default font and colors. (For an application to use a nonstandard typeface, font style, or font size, the user must have a color monitor.)

item list A resource (of type `'DITL'`) that specifies the items—such as buttons and static text—to display in an alert box or a dialog box.

item number An integer that identifies an item in either a menu or a dialog box. Menu items are assigned item numbers starting with 1 for the first menu item in the menu, 2 for the second menu item in the menu, and so on, up to the number of the last menu item in the menu. Dialog items are assigned numbers that correspond to the item's position in its item list. For example, the first item listed in a dialog item list is item number 1.

keyboard equivalent A keyboard combination of one or more modifier keys and another key that invokes a corresponding menu command when pressed by the user.

key-down event An event indicating that the user pressed a key on the keyboard.

key-up event An event indicating that the user released a key on the keyboard.

local coordinate system The coordinate system defined by the port rectangle of a graphics port. When the Window Manager creates a window, it places the origin of the local coordinate system at the upper-left corner of the window's port rectangle. Compare **global coordinate system.**

location name An identifier for the network location of the computer on which a PPC port resides. A location name consists of an object string, a type string, and a zone.

low-level event The type of event returned by the Event Manager to report very low level hardware and software occurrences. Low-level events report actions by the user, changes in windows on the screen, and that the Event Manager has no other events to report. Compare **high-level event, operating-system event.**

major switch A change of the foreground process. The Process Manager switches the context of the foreground process with the context of a background process (including the A5 worlds and low-memory global variables) and brings the background process to the front, sending the previous foreground process to the background. See also **context.**

menu A user interface element you can use in your application to allow the user to view or choose an item from a list of choices and commands that your application provides. See also **hierarchical menu, pop-up menu, pull-down menu,** and **submenu.**

menu bar A white rectangle that is tall enough to display menu titles in the height of the system font and system font size, and with a black lower border that is one pixel tall. The menu bar extends across the top of the startup screen and contains the title of each available pull-down menu.

menu bar definition function A function that draws the menu bar and performs most of the drawing activities related to the display of menus when the user moves the cursor between menus. This function, in conjunction with the menu definition procedure, defines the general appearance and behavior of menus.

menu bar entry A menu color entry record that contains 0 in both the `mctID` and `mctItem` fields. A menu bar entry defines the color for an application's menu bar and defines default colors for its menu titles, menu items, and background color of menus.

menu bar resource A resource (of type `'MBAR'`) that specifies the order and resource ID of each menu in a menu bar.

menu color entry record A data structure of type `MCEntry` that defines the colors for an application's menu bar, menus, or menu items. The first two fields of a menu color entry record, `mctID` and `mctItem`, define whether the entry is a menu bar entry, a menu title entry, or a menu item entry.

menu color information table An array of menu color entry records, maintained by the Menu Manager, that define the standard color for the menu bar, titles of menus, text and characteristics of menu items, and background color of a displayed menu. If you do not add any entries to this table, the Menu Manager draws your menus using the default colors, black on white.

menu color information table resource A resource (of type `'mctb'`) that specifies the colors for an application's menu bar, menus, and menu items.

menu definition procedure A procedure that performs all the drawing of menu items within a specific menu. This procedure, in conjunction with the menu bar definition function, defines the general appearance and behavior of menus.

menu ID A number that you assign to a menu in your application. Each menu in your application must have a unique menu ID.

menu item In a menu, a rectangle with text and other characteristics identifying a command that the user can choose.

menu item entry A menu color entry record that contains nonzero values in both the `mctID` and `mctItem` fields. A menu item entry defines colors for the mark, text, and keyboard equivalent of items in a specific menu. It also defines the default background color of a menu.

menu list A data structure that contains handles to the menu records of one or more menus (although a menu list can be empty). Compare **current menu list.**

Menu Manager The collection of routines that an application can use to create, display, and manage its menus.

menu record A data structure of type `MenuInfo` that the Menu Manager uses to maintain information about a menu.

menu resource A resource (of type `'MENU'`) that specifies the menu title and the individual characteristics of items in a menu.

menu title entry A menu color entry record that contains a nonzero value in the `mctID` field and contains 0 in the `mctItem` field. A menu title entry defines colors for the title, items, and background color of a specific menu. It also defines the default menu bar color.

minimum partition size The actual partition size limit below which an application cannot run.

minor switch A change in the context of a process. The Process Manager switches the context of a process to give time to a background process without bringing the background process to the front.

modal dialog box A dialog box that puts the user in the state or "mode" of being able to work only inside the dialog box. A modal dialog box resembles an alert box. The user cannot move a modal dialog box and can dismiss it only by clicking its buttons. See also **modeless dialog box** and **movable modal dialog box.**

modeless dialog box A dialog box that looks like a document window without a size box or scroll bars. The user can move a modeless dialog box, make it inactive and active again, and close it like any document window. See also **modal dialog box** and **movable modal dialog box.**

modifier keys The Shift, Option, Command, Control, and Caps Lock keys.

mouse-down event An event indicating that the user pressed the mouse button.

mouse location The location of the cursor at the time the event occurred.

mouse-moved event An event indicating that the cursor is outside of a specified region.

mouse-up event An event indicating that the user released the mouse button.

movable modal dialog box A modal dialog box that has a title bar (with no close box) by which the user can drag the dialog box. See also **dialog box, modal dialog box,** and **modeless dialog box.**

note alert An alert box that informs users of a minor mistake that won't have any disastrous consequences if left as is. Usually a note alert simply offers information, and the user responds by clicking the OK button. A note alert is identified by an icon bearing a face and a cartoonlike dialog balloon in the upper-left corner of the alert box. See also **caution alert** and **stop alert.**

null event An event indicating that no events of the requested types exist in the application's event stream.

offset point The point in a region whose horizontal and vertical offsets from the upper-left corner of the region's enclosing rectangle are the same as the offsets of a specified point. The `DrayGrayRgn` function uses an offset point to limit the motion of a region and to calculate the distance a region has moved.

operating-system event An event returned by the Event Manager to communicate information about changes in the operating status of applications (suspend and resume events) and to report that the user has moved the cursor outside of an area specified by the application (mouse-moved events). Compare **low-level event, high-level event.**

Operating System Event Manager The collection of low-level routines that manage the Operating System event queue.

Operating System event queue A queue that the Operating System Event Manager creates and maintains. The Operating System Event Manager detects and reports low-level hardware-related events such as mouse clicks, keypresses, and disk insertions and places these events in the Operating System event queue.

part code An integer from 1 through 253 that stands for a particular part of a control. The `FindControl` and `TrackControl` functions return a part code to indicate the location of the cursor when the user presses the mouse button.

pop-up menu A menu that appears elsewhere than the menu bar. The Control Manager provides a control definition function for applications to use when implementing pop-up menus.

port name A unique identifier for a particular application on a computer, used for the purposes of communication between applications. A port name consists of a name string, a type string, and a script code.

port rectangle An entry in the graphics port data structure, described in *Inside Macintosh: Imaging.* Ordinarily, the port rectangle represents the area of a graphics port available for drawing—that is, the content region of a window.

Preferences folder A directory located in the System Folder for holding files that record users' configuration settings for applications on a particular Macintosh computer.

preferred partition size The partition size at which an application can run most effectively. The Operating System attempts to secure this partition size upon launch of the application.

PrintMonitor Documents folder A directory located in the System Folder for storing spooled documents waiting to be printed.

process An open application or, in some cases, an open desk accessory. (Only desk accessories that are not opened in the context of another application are considered processes.)

process serial number A number assigned by the Process Manager to identify a particular instance of an application during a single boot of the local machine.

pull-down menu A menu that is identified by a menu title (a word or an icon) in the menu bar.

query document A file of file type `'qery'` containing commands and data in a format appropriate for a database or other data source. An application uses high-level Data Access Manager routines to open a query document.

radio button A control that appears onscreen as a small circle. A radio button displays one of two settings: on (indicated by a black dot inside the circle) or off. A radio button is always a part of a group of related radio buttons in which only one button can be on at a time. When the user clicks an unmarked radio button, the application turns that button on and turns the other buttons in its group off.

Rescued Items from *volume name* folder A directory located in the Trash directory and created by the Finder at system startup, restart, or shutdown only when it finds items in the Temporary Items folder, usually after a system crash. The Rescued Items from *volume name* folder is named for the volume on which the Temporary Items folder exists. When a user empties the Trash, all Rescued Items folders disappear.

resource Any data stored according to a defined structure in a resource fork of a file; the data in a resource is interpreted according to its resource type.

resource fork The part of a file that contains the files' resources. A resource fork consists of a resource map and resources.

resource ID A number that identifies a specific resource of a given resource type.

resource type A sequence of four characters that uniquely identifies a specific type of resource.

resume event An event indicating that an application has been switched back into the foreground and can resume interacting with the user.

return receipt A high-level event that indicates whether the other application accepted the high-level event sent to it by your application.

scroll arrow An arrow at either end of a scroll bar. When the user clicks a scroll arrow, the application moves a document or list one line (or some similar measure) in the direction of the arrow. When the user holds the mouse button down while the cursor is over a scroll arrow, the application moves the document or list continuously in the direction of the arrow.

scroll bar A control with which the user can change the portion of a document displayed within a window. A scroll bar is a light gray rectangle with scroll arrows at each end. Windows can have a horizontal scroll bar, a vertical scroll bar, or both. A vertical scroll bar lies along the right side of a window. A horizontal scroll bar runs along the bottom of a window. Inside the scroll bar is a rectangle called the scroll box. The rest of the scroll bar is called the gray area. The user can move through a document by manipulating the parts of the scroll bar.

scroll box A box that slides up and down or back and forth across a scroll bar. The position of the scroll box in a scroll bar indicates the position of the window contents relative to the entire document. When the user drags the scroll box, the application displays a different portion of the document.

signature A resource whose type is defined by a four-character sequence that uniquely identifies an application to the Finder. A signature is located in an application's resource fork.

size box A box in the lower-right corner of windows that can be resized. Dragging the size box resizes the window.

size region The area occupied by a window's size box. See also **size box.**

size resource A resource (of type `'SIZE'`) that specifies the operating characteristics, minimum partition size, and preferred partition size of an application.

slider A control, such as a scroll bar, that graphically represent the ranges of values that a user can set or that simply displays the value, magnitude, or position of something, typically in some pseudo-analog form.

standard control definition functions Three control definition functions, stored as `'CDEF'` resources in the System file. The `'CDEF'` resource with resource ID 0 defines the look and behavior of buttons, checkboxes, and radio buttons; the `'CDEF'` resource with resource ID 1 defines the look and behavior of scroll bars; and the `'CDEF'` resource with resource ID 63 defines the look and behavior of pop-up menus.

standard state The size and location that an application deems the most convenient for a window.

Startup Items folder A directory located in the System Folder for storing applications and desk accessories that the user wants started up every time the Finder starts up.

stationery pad A document that a user creates to serve as a template for other documents. The Finder tags a document as a stationery pad by setting the `isStationery` bit in the Finder flags field of the file's file information record. An application that is asked to open a stationery pad should copy the template's contents into a new document and open the document in an untitled window.

stop alert An alert box that informs the user of a problem or situation so serious that the user's desired action cannot be completed. Stop alerts typically have only a single button (OK), because all the user can do is acknowledge that the action cannot be completed. A stop alert is identified by an icon of an upraised hand in the upper-left corner of the alert box. See also **caution alert** and **note alert.**

structure region The entire screen area occupied by a window, including both the window frame and the content region.

submenu A menu that is attached to another menu.

suspend event An event indicating that the execution of your application is about to be suspended as the result of a major switch. The application is suspended at the application's next call to `WaitNextEvent` or `EventAvail`.

system alert sound A sound resource that is stored in the System file and played whenever system software or an application uses the Sound Manager procedure `SysBeep`. With the Sound control panel, the user can select which sound to use.

System file A file, located in the System Folder, that contains the basic system software plus some system resources, such as sound and keyboard resources.The System file behaves like a folder in this regard: although it looks like a suitcase icon, double-clicking it opens a window that reveals movable resource files (such as sounds, keyboard layouts, and script system resource collections) stored in the System file.

System Folder A directory containing the software that Macintosh computers use to start up. The System Folder includes a set of folders for storing related files, such as preferences files that an application might need when starting up.

Temporary Items folder A directory located at the root level of a volume for storing temporary buffer files created by applications. The Temporary Items folder is invisible to the user.

text style table In an item color table resource, a specification for the typeface, font style, font size, and color of text in an editable text item or a static text item.

title bar The bar at the top of a window that displays the window name, contains the close and zoom boxes, and indicates whether the window is active.

Toolbox Event Manager See **Event Manager.**

Trash folder A directory at the root level of a volume for storing files that the user has moved to the Trash icon. After opening the Trash icon, the user sees the collection of all items that the user has moved to the Trash icon—that is, the union of appropriate Trash directories from all mounted volumes. A Macintosh computer set up to share files among users in a network environment maintains separate Trash subdirectories for remote users within its shared Trash directory. The Finder empties a Trash directory (or, in the case of a file server, a Trash subdirectory) only when the user of that directory chooses the Empty Trash command.

update event An event indicating that the contents of a window need updating.

update region A region maintained by the Window Manager that includes the parts of a window's content region that need updating. The Event Manager generates update events as necessary, based on the contents of the update region, telling your application to update a window.

user state The size and location that the user has established for a window.

variation code A number that selects among variations supported by a single window defintion function or control definition function. The variation code is stored in the low-order 4 bits of the window definition ID or control definition ID. See also **control definition function, control definition ID, window definition function,** and **window definition ID.**

virtual key code A value that represents the key pressed or released by the user; this value is always the same for a specific physical key on a keyboard. Compare **character code.**

visible region The part of a window's graphics port that's actually visible on the screen—that is, the part that's not covered by other windows.

window An area on the screen that displays information, including user documents as well as communications such as alert boxes and dialog boxes. The user can open or close a window; move it around on the desktop; and sometimes change its size, scroll through it, and edit its contents.

window color table The data structure in which the Window Manager stores the colors to be used for drawing a window's frame and for highlighting selected text.

window definition function A function that defines the general appearance and behavior of a window. The Window Manager calls the window definition function to draw the window's frame, determine what region of the window the cursor is in, draw the window's size box, draw the window's zoom box, move and resize the window, and calculate the window's structure and content regions.

window definition ID An integer that specifies the resource ID of a window definition function in the upper 12 bits and an optional variation code in the lower 4 bits. When creating a new window, your application supplies a window definition ID either as a field in the `'WIND'` resource or as a parameter to the `NewWindow` or `NewCWindow` function.

window list A list maintained by the Window Manager of all windows on the desktop. The frontmost window is first in the window list, and the remaining windows appear in the order in which they are layered on the desktop.

Window Manager port A graphics port that represents the desktop area on the main monitor—that is, a rounded-corner rectangle that occupies all of the main monitor except for the area occupied by the title bar.

window origin The upper-left corner of a window. Usually specified as (0,0), the window origin is expressed in coordinates local to the window.

window record A data structure of type `WindowRecord` (or `CWindowRecord`) in which the Window Manager stores a window's characteristics, including the window's graphics port, title, visibility status, and control list.

window region Special-purpose region of a window. See also **close region, content region, drag region, size region,** and **zoom region.**

window type A collection of characteristics—such as the shape of the window's frame and the features of its title bar—that describe a window.

zoom box A box in the right side of a window's title bar that the user can click to alternate between two different window sizes (the user state and the standard state).

zoom region The area occupied by a window's zoom box. See also **zoom box.**

Index

T H E   A P P L E   P U B L I S H I N G   S Y S T E M

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter IINTX printer. Final page negatives were output directly from text files on an AGFA ProSet 9800 imagesetter. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

LEAD WRITER
Sharon Everson

WRITERS
Sharon Everson, Tony Francis, Sue Luttner

DEVELOPMENTAL EDITORS
Sue Factor, Antonio Padial, Anne Szabla, Beverly Zegarski

INDEX SPECIALIST
Sanborn Hodgkins

ILLUSTRATORS
Bruce Lee, Ruth Anderson, Pat Rogondino, Martine Trelaun

COVER DESIGNER
Barb Smyth

PRODUCTION EDITOR
Rex Wolf

PROJECT MANAGER
Patricia Eastman

Special thanks to Ray Chiang, Kevin MacDonell, Dave Radcliffe, Jim Reekes

Acknowledgments to Eric Anderson, Randy Carr, John Glenn, Sandee Karr, Peggy Kunz, Tim Monroe, Dave Owens, Dean Yu, and the entire *Inside Macintosh* team.