



INSIDE MACINTOSH

More Macintosh Toolbox



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.
© 1993, Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

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

Apple, the Apple logo, APDA, AppleLink, AppleShare, AppleTalk, EtherTalk, ImageWriter, LaserWriter, LocalTalk, Macintosh, MPW, StyleWriter, and TokenTalk are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Apple Desktop Bus, Balloon Help, BalloonWriter, Chicago, Finder, Geneva, KanjiTalk, QuickDraw, QuickTime, ResEdit, System 7, and System 7.0 are trademarks of Apple Computer, Inc. Adobe Illustrator and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

AGFA is a trademark of Agfa-Gevaert.

America Online is a service mark of Quantum Computer Services, Inc.

CompuServe is a registered service mark of CompuServe, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company. Internet is a trademark of Digital Equipment Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

ISBN 0-201-63299-3
1 2 3 4 5 6 7 8 9-CRW-9796959493
First Printing, October 1993

Library of Congress Cataloging-in-Publication Data

Inside Macintosh : more Macintosh toolbox / [Apple Computer, Inc.].

p. cm.

Includes index.

ISBN 0-201-63299-3

1. Macintosh (Computer)—Programming. 2. Macintosh Toolbox.

I. Apple Computer, Inc.

QA76.8.M315613 1993

005.265—dc20

93-33260
CIP

Contents

Figures, Tables, and Listings xvii

Preface

About This Book xxv

Format of a Typical Chapter xxvi
Conventions Used in This Book xxvii
 Special Fonts xxvii
 Types of Notes xxvii
 Empty Strings xxviii
 Assembly-Language Information xxviii
The Development Environment xxviii

Chapter 1

Resource Manager 1-1

Introduction to Resources 1-3
 The Data Fork and the Resource Fork 1-4
 Resource Types and Resource IDs 1-6
 The Resource Map 1-8
 Search Path for Resources 1-10
About the Resource Manager 1-12
Using the Resource Manager 1-13
 Creating a Resource 1-15
 Getting a Resource 1-18
 Releasing and Detaching Resources 1-22
 Opening a Resource Fork 1-24
 Opening an Application's Resource Fork 1-24
 Creating and Opening a Resource Fork 1-25
 Specifying the Current Resource File 1-28
 Reading and Manipulating Resources 1-30
 Writing Resources 1-36
 Working With Partial Resources 1-40
Resource Manager Reference 1-42
 Data Structure, Resource Types, and Resource IDs 1-42
 The Resource Type 1-42
 Resource IDs 1-46
 Resource IDs of Owned Resources 1-47
 Resource Names 1-49
 Resource Manager Routines 1-49
 Initializing the Resource Manager 1-50
 Checking for Errors 1-51
 Creating an Empty Resource Fork 1-53

Opening Resource Forks	1-58
Getting and Setting the Current Resource File	1-68
Reading Resources Into Memory	1-71
Getting and Setting Resource Information	1-81
Modifying Resources	1-87
Writing to Resource Forks	1-92
Getting a Unique Resource ID	1-95
Counting and Listing Resource Types	1-97
Getting Resource Sizes	1-104
Disposing of Resources	1-106
Closing Resource Forks	1-110
Reading and Writing Partial Resources	1-111
Getting and Setting Resource Fork Attributes	1-116
Accessing Resource Entries in a Resource Map	1-119
Resource File Format	1-121
Resources in the System File	1-126
User Information Resources	1-127
Packages	1-128
Function Key Resources	1-129
Standard Icons	1-129
ROM Resources	1-134
Inserting the ROM Resource Map	1-134
Overriding ROM Resources	1-135
Summary of the Resource Manager	1-137
Pascal Summary	1-137
Constants	1-137
Data Type	1-139
Routines	1-139
C Summary	1-142
Constants	1-142
Data Type	1-143
Routines	1-144
Assembly-Language Summary	1-147
Trap Macros	1-147
Global Variables	1-147
Result Codes	1-148

Chapter 2

Scrap Manager 2-1

Introduction to the Scrap Manager	2-4
The Clipboard	2-10
Intelligent Cut and Paste	2-10
About the Scrap Manager	2-12
Location of the Scrap	2-12
Using the Scrap Manager	2-14
Getting Information About the Scrap	2-15

Putting Data in the Scrap	2-15
Handling the Cut Command	2-15
Handling the Copy Command	2-19
Handling Suspend Events	2-19
Getting Data From the Scrap	2-20
Handling the Paste Command	2-20
Handling Resume Events	2-25
Converting Data Between a Private Scrap and the Scrap	2-26
Converting Data Between the TextEdit Scrap and the Scrap	2-28
Handling Editing Operations in Dialog Boxes	2-31
Scrap Manager Reference	2-31
Data Structures	2-32
The Scrap Information Record	2-32
The Scrap Format Types	2-33
Routines	2-34
Getting Information About the Scrap	2-34
Writing Information to the Scrap	2-35
Reading Information From the Scrap	2-38
Transferring Data Between the Scrap in Memory and the Scrap on Disk	2-40
Summary of the Scrap Manager	2-42
Pascal Summary	2-42
Constants	2-42
Data Types	2-42
Routines	2-42
C Summary	2-43
Data Types	2-43
Routines	2-44
Assembly-Language Summary	2-45
Data Structures	2-45
Result Codes	2-45

Chapter 3

Help Manager 3-1

About the Help Manager	3-6
How the Help Manager Displays Balloons	3-8
Default Help Balloons for Menus, Windows, and Icons	3-13
About BalloonWriter	3-17
Using the Help Manager	3-18
Providing Text or Pictures for Help Balloons	3-18
Defining Help Messages	3-19
Using Clear, Concise Phrases	3-20
Using Active Constructions	3-22
Using Parallel Structure	3-22
Offering Hints	3-22
Using Consistent Terminology	3-23

Defining the Help Balloon Position	3-23
Specifying the Format for Help Messages	3-23
Specifying Options in Help Resources	3-25
Providing Help Balloons for Menus	3-27
Specifying Header Information for the 'hmnu' Resource	3-32
Specifying Help for Menu Items Missing From the Resource	3-33
Specifying Help for Menu Titles and for Items Dimmed by System Software	3-36
Specifying Help for Menu Items	3-39
Specifying Help for a Changing Menu Item	3-43
Specifying Resources by Item Name	3-45
Providing Help Balloons for Menus You Disable for Dialog Boxes	3-47
Providing Help Balloons for Items in Dialog Boxes and Alert Boxes	3-51
Specifying Header Information for the 'hdlg' Resource	3-54
Specifying Missing-Item Information	3-54
Specifying Help for Items in an Alert or Dialog Box	3-56
Adding a Help Item to an Item List Resource	3-62
Using a Help Item Versus Using an 'hwin' Resource	3-63
Providing Help Balloons for Window Content	3-63
Providing Help Balloons for Static Windows	3-65
Specifying Header Information for the 'hrct' Resource	3-67
Specifying Help for Rectangles in Windows	3-67
Associating Help Resources With Static Windows	3-68
Specifying Header Information for the 'hwin' Resource	3-69
Specifying 'hdlg' or 'hrct' Resources in the 'hwin' Resource	3-69
Providing Help Balloons for Dynamic Windows	3-74
Overriding Help Balloons for Non-Document Icons	3-84
Specifying Header Information for the 'hldr' Resource	3-85
Specifying Help for an Icon	3-85
Overriding Other Default Help Balloons	3-87
Specifying Header Information for the 'hovr' Resource	3-88
Overriding Default Help	3-88
Adding Menu Items to the Help Menu	3-90
Writing Your Own Balloon Definition Function	3-93
Help Manager Reference	3-95
Data Structures	3-95
The Help Message Record	3-95
The Help Manager String List Record	3-97
Help Manager Routines	3-97
Determining Balloon Help Status	3-98
Displaying and Removing Help Balloons	3-99
Enabling and Disabling Balloon Help Assistance	3-107
Adding Items to the Help Menu	3-108
Getting and Setting the Font Name and Size	3-110
Setting and Getting Information for Help Resources	3-114
Determining the Size of a Help Balloon	3-119
Getting the Message of a Help Balloon	3-122

Application-Defined Routines	3-128
Resources	3-132
The Menu Help Resource	3-132
The Dialog-Item Help Resource	3-140
The Rectangle Help Resource	3-148
The Window Help Resource	3-154
The Finder Icon Help Resource	3-156
The Default Help Override Resource	3-160
Summary of the Help Manager	3-166
Pascal Summary	3-166
Constants	3-166
Data Types	3-168
Help Manager Routines	3-169
Application-Defined Routines	3-170
C Summary	3-170
Constants	3-170
Data Types	3-173
Help Manager Routines	3-173
Application-Defined Routines	3-175
Assembly-Language Summary	3-176
Data Structures	3-176
Trap Macros	3-176
Result Codes	3-177

Chapter 4

List Manager 4-1

Introduction to Lists	4-4
Appearance of Lists	4-4
Selection of List Items	4-9
Keyboard Navigation of Lists	4-15
Movement of a Selection With Arrow Keys	4-15
Extension of a Selection With Arrow Keys	4-16
Type Selection in a Text-Only List	4-20
Multiple Lists in a Window	4-20
About the List Manager	4-22
Using the List Manager	4-26
Creating a List	4-27
Adding Rows and Columns to a List	4-30
Responding to Events Affecting a List	4-32
Working With List Selections	4-34
Customizing Cell Highlighting	4-38
Manipulating List Cells	4-40
Searching a List for a Particular Item	4-43
Supporting Keyboard Navigation of Lists	4-45
Supporting Type Selection of List Items	4-45
Supporting Arrow-Key Navigation of Lists	4-48

Supporting the Anchor Algorithm for Extending Lists With Arrow Keys	4-52	
Outlining the Current List	4-53	
Writing Your Own List Definition Procedure	4-58	
Responding to the Initialization Message	4-60	
Responding to the Draw Message	4-60	
Responding to the Highlighting Message	4-62	
Responding to the Close Message	4-62	
Using the Pictures List Definition Procedure	4-63	
List Manager Reference	4-65	
Data Structures	4-65	
The Cell Record	4-65	
The Data Handle	4-66	
The List Record	4-66	
List Manager Routines	4-70	
Creating and Disposing of Lists	4-70	
Adding and Deleting Columns and Rows To and From a List	4-73	
Determining or Changing the Selection	4-77	
Accessing and Manipulating Cell Data	4-79	
Responding to Events Affecting Lists	4-84	
Modifying a List's Appearance	4-87	
Searching a List for a Particular Item	4-90	
Changing the Size of Cells and Lists	4-91	
Getting Information About Cells	4-93	
Application-Defined Routines	4-96	
List Definition Procedures	4-96	
Match Functions	4-99	
Click-Loop Procedures	4-100	
Summary of the List Manager	4-102	
Pascal Summary	4-102	
Constants	4-102	
Data Types	4-102	
List Manager Routines	4-103	
Application-Defined Routines	4-105	
C Summary	4-106	
Constants	4-106	
Data Types	4-106	
List Manager Routines	4-107	
Application-Defined Routines	4-109	
Assembly-Language Summary	4-110	
Data Structures	4-110	
Trap Macros	4-111	

Introduction to the Icon Utilities	5-3
About the Icon Utilities	5-6
Using the Icon Utilities	5-7
Drawing Icons in an Icon Family	5-8
Drawing an Icon Directly From a Resource	5-10
Getting an Icon Suite and Drawing One of Its Icons	5-11
Drawing Specific Icons From an Icon Family	5-12
Manipulating Icons	5-13
Drawing Icons That Are Not Part of an Icon Family	5-13
Icon Utilities Reference	5-17
Data Structure	5-17
The Color Icon Record	5-17
Icon Utilities Routines	5-18
Drawing Icons From Resources	5-19
Getting Icons From Resources That Don't Belong to an Icon Family	5-28
Disposing of Icons	5-30
Creating an Icon Suite	5-30
Getting Icons From an Icon Suite	5-34
Drawing Icons From an Icon Suite	5-35
Performing Operations on Icons in an Icon Suite	5-38
Getting and Setting the Label for an Icon Suite	5-40
Getting Label Information	5-41
Disposing of Icon Suites	5-42
Converting an Icon Mask to a Region	5-43
Determining Whether a Point or Rectangle Is Within an Icon	5-46
Working With Icon Caches	5-53
Application-Defined Routines	5-57
Icon Action Functions	5-57
Icon Getter Functions	5-58
Summary of the Icon Utilities	5-60
Pascal Summary	5-60
Constants	5-60
Data Types	5-62
Icon Utilities Routines	5-62
Application-Defined Routines	5-65
C Summary	5-65
Constants	5-65
Data Types	5-67
Icon Utilities Routines	5-68
Application-Defined Routines	5-71
Assembly-Language Summary	5-71
Data Structure	5-71
Trap Macros	5-72
Result Codes	5-73

Introduction to Components	6-3	
About the Component Manager	6-4	
Using the Component Manager	6-6	
Opening Connections to Components	6-7	
Opening a Connection to a Default Component		6-7
Finding a Specific Component	6-8	
Opening a Connection to a Specific Component		6-9
Getting Information About a Component	6-10	
Using a Component	6-11	
Closing a Connection to a Component	6-12	
Creating Components	6-13	
The Structure of a Component	6-13	
Handling Requests for Service	6-18	
Responding to the Open Request	6-19	
Responding to the Close Request	6-21	
Responding to the Can Do Request	6-22	
Responding to the Version Request	6-22	
Responding to the Register Request	6-23	
Responding to the Unregister Request	6-24	
Responding to the Target Request	6-25	
Responding to Component-Specific Requests		6-26
Reporting an Error Code	6-28	
Defining a Component's Interfaces	6-28	
Managing Components	6-30	
Registering a Component	6-30	
Creating a Component Resource	6-32	
Establishing and Managing Connections		6-34
Component Manager Reference	6-37	
Data Structures for Applications	6-37	
The Component Description Record	6-37	
Component Identifiers and Component Instances		6-40
Routines for Applications	6-41	
Finding Components	6-42	
Opening and Closing Components	6-44	
Getting Information About Components	6-47	
Retrieving Component Errors	6-51	
Data Structures for Components	6-52	
The Component Description Record	6-52	
The Component Parameters Record	6-54	
Routines for Components	6-56	
Registering Components	6-57	
Dispatching to Component Routines	6-63	
Managing Component Connections	6-65	
Setting Component Errors	6-69	
Working With Component Reference Constants		6-70

Accessing a Component's Resource File	6-71
Calling Other Components	6-73
Capturing Components	6-75
Targeting a Component Instance	6-77
Changing the Default Search Order	6-78
Application-Defined Routines	6-79
Resources	6-80
The Component Resource	6-80
Summary of the Component Manager	6-86
Pascal Summary	6-86
Constants	6-86
Data Types	6-87
Routines for Applications	6-89
Routines for Components	6-90
Application-Defined Routines	6-92
C Summary	6-92
Constants	6-92
Data Structures	6-93
Routines for Applications	6-95
Routines for Components	6-96
Application-Defined Routines	6-97
Assembly-Language Summary	6-98
Trap Macros	6-98
Result Codes	6-99

Chapter 7

Translation Manager 7-1

About the Translation Manager	7-4
Opening Documents From the Finder	7-5
Opening Documents Within an Application	7-8
Translating Documents on the Desktop	7-9
Sharing Data Between Applications	7-10
Using the Translation Manager	7-10
Checking for the Translation Manager	7-12
Declaring the File Types Your Application Can Open	7-13
Declaring Custom Kind Strings	7-14
Using File-Opening Dialog Boxes	7-15
Translating Files Explicitly	7-17
Writing a Translation Extension	7-18
Creating a Translation Extension	7-19
Dispatching to Translation Extension-Defined Routines	7-24
Creating a Translation List	7-27
Identifying Files	7-32
Translating Files	7-33
Writing Application Translation Extensions	7-35

Translation Manager Reference	7-36	
Translation Manager Routines	7-36	
Getting Translation Information	7-37	
Translating Files	7-42	
Resources	7-43	
The Open Resource	7-44	
The Kind Resource	7-45	
Translation Extension Reference	7-46	
Translation Extension Data Structures	7-46	
File Type Specifications	7-46	
File Translation Lists	7-48	
Scrap Type Specifications	7-49	
Scrap Translation Lists	7-49	
Translation Extension Routines	7-50	
Managing Translation Progress Dialog Boxes	7-50	
Translation Extension-Defined Routines	7-54	
File Translation Extension Routines	7-54	
Scrap Translation Extension Routines	7-58	
Summary of the Translation Manager	7-63	
Pascal Summary	7-63	
Constants	7-63	
Data Types	7-63	
Translation Manager Routines	7-64	
C Summary	7-64	
Constants	7-64	
Data Types	7-65	
Translation Manager Routines	7-65	
Assembly-Language Summary	7-66	
Data Structures	7-66	
Trap Macros	7-66	
Result Codes	7-67	
Summary of Translation Extensions	7-68	
Pascal Summary	7-68	
Constants	7-68	
Data Types	7-68	
Translation Extension Routines	7-70	
Translation Extension-Defined Routines	7-70	
C Summary	7-71	
Constants	7-71	
Data Types	7-71	
Translation Extension Routines	7-73	
Translation Extension-Defined Routines	7-73	
Assembly-Language Summary	7-74	
Data Structures	7-74	
Trap Macros	7-75	
Result Codes	7-75	

About Control Panels	8-4
Control Panels	8-4
A Control Panel's Resources	8-6
The Finder's Interaction With Control Panels	8-7
Control Panels and System Extensions	8-8
About User Documentation for Control Panels	8-8
The Monitors Control Panel and Extensions to It	8-9
Creating Control Panel Files	8-12
Defining the User Interface for a Control Panel	8-12
Creating a Control Panel's Resources	8-14
Resource IDs for Control Panels	8-14
Defining the Control Panel Rectangles	8-15
Creating the Item List Resource	8-17
Defining the Icon for a Control Panel	8-20
Specifying the Machine Resource	8-20
Creating the File Reference, Bundle, and Signature Resources	8-21
Providing Additional Resources for a Control Panel	8-22
Specifying the Font of Text in a Control Panel	8-23
Creating a Font Information Resource	8-23
Defining Text in a Control Panel as User Items	8-24
Writing a Control Panel Function	8-25
Determining If a Control Panel Can Run on the Current System	8-29
Initializing the Control Panel Items and Allocating Storage	8-29
Responding to Activate Events	8-33
Responding to Keyboard Events	8-37
Responding to Mouse Events	8-39
Responding to Update Events	8-43
Handling Text Defined as User Items	8-43
Responding to Null Events	8-45
Responding to the User Closing the Control Panel	8-45
Handling Edit Menu Commands	8-46
Handling Errors	8-47
Creating an Extension for the Monitors Control Panel	8-48
Designing the User Interface for a Monitors Extension	8-49
Creating the Required Resources for a Monitors Extension	8-51
Creating a Card Resource for a Monitors Extension	8-51
Defining a Rectangle for a Monitors Extension	8-52
Creating an Item List Resource for a Monitors Extension	8-54
Creating the Monitor Code Resource	8-56
Supplying Optional Resources for a Monitors Extension	8-56
Specifying an Icon for the Options Dialog Box	8-57
Specifying Version Information	8-58
Providing an Alternative Name for a Video Card	8-58
Supplying Gamma Table Resources	8-59
Creating File Reference, Bundle, and Signature Resources	8-59

Including a System Extension Resource	8-61
Writing a Monitors Extension Function	8-61
Handling the Startup Message	8-66
Performing Initialization	8-68
Responding to a Click in the OK Button	8-70
Responding to a Cancel Request	8-71
Handling Mouse Events for a Monitors Extension	8-71
Handling Keyboard Events	8-73
Including Another Control Panel Definition in a Monitors Extension File	8-73
Control Panels Reference	8-74
Application-Defined Routines	8-74
Control Device Functions	8-74
Monitors Extension Functions	8-78
Resources	8-82
The Machine Resource	8-84
The Rectangle Positions Resource	8-85
The Font Information Resource	8-86
The Control Device Function Code Resource	8-87
The Card Resource	8-87
The Monitor Code Resource	8-88
The Rectangle Resource	8-88
Summary of Control Panels	8-89
Pascal Summary	8-89
Constants	8-89
Application-Defined Routines	8-90
C Summary	8-90
Constants	8-90
Application-Defined Routines	8-92

Chapter 9

Desktop Manager 9-1

About the Desktop Database	9-4
Using the Desktop Manager	9-4
Desktop Manager Reference	9-6
Data Structure	9-6
The Desktop Parameter Block	9-7
Routines	9-8
Locating, Opening, and Closing the Desktop Database	9-9
Reading the Desktop Database	9-12
Adding to the Desktop Database	9-17
Deleting Entries From the Desktop Database	9-20
Manipulating the Desktop Database Itself	9-23
Summary of the Desktop Manager	9-27
Pascal Summary	9-27
Constants	9-27

Data Types	9-27	
Routines	9-28	
C Summary	9-30	
Constants	9-30	
Data Types	9-31	
Routines	9-31	
Assembly-Language Summary		9-34
Data Structures	9-34	
Trap Macros	9-35	
Result Codes	9-35	

Glossary GL-1

Index IN-1

Figures, Tables, and Listings

Chapter 1

Resource Manager 1-1

Figure 1-1	The data fork and resource fork of a file	1-4
Figure 1-2	An application's and a document's data fork and resource fork	1-6
Figure 1-3	Resource attributes	1-8
Figure 1-4	A typical search order for a specific resource	1-11
Figure 1-5	The ResEdit window for the SurfWriter application	1-15
Figure 1-6	The menus of the SurfWriter application	1-16
Figure 1-7	Getting a handle to a resource	1-19
Figure 1-8	A handle to a purgeable resource after the resource has been purged	1-20
Figure 1-9	Detaching a resource	1-23
Figure 1-10	Resource ID of an owned resource	1-48
Figure 1-11	Format of a resource fork	1-121
Figure 1-12	Format of a resource header in a resource fork	1-122
Figure 1-13	Format of resource data for a single resource	1-122
Figure 1-14	Format of the resource map in a resource fork	1-123
Figure 1-15	Format of an item in a resource type list	1-123
Figure 1-16	Format of an entry in the reference list for a resource type	1-124
Figure 1-17	Format of an item in a resource name list	1-124
Figure 1-18	Offsets in a resource fork and an entry for a single resource in a reference list	1-125
Figure 1-19	Structure of a compiled ROM override ('ROv#') resource	1-136
Table 1-1	Typical locations of resources	1-12
Table 1-2	Standard resource types	1-43
Table 1-3	Resource types reserved for use by system software	1-46
Table 1-4	Document and application icons	1-130
Table 1-5	Folder icons	1-131
Table 1-6	System Folder icons	1-132
Table 1-7	Desktop icons	1-133
Table 1-8	Standard File Package icons	1-133
Listing 1-1	A menu in Rez input format	1-17
Listing 1-2	Safely changing a resource that is purgeable	1-21
Listing 1-3	Releasing a resource	1-22
Listing 1-4	Detaching a resource	1-24
Listing 1-5	Getting the file reference number for your application's resource fork	1-25
Listing 1-6	Creating an empty resource fork	1-26
Listing 1-7	Creating and opening a resource fork	1-27
Listing 1-8	Saving and restoring the current resource file	1-29
Listing 1-9	Getting a resource from a document file	1-32
Listing 1-10	Counting and indexing through resources	1-34

Listing 1-11	Saving a resource to a resource fork	1-38
Listing 1-12	Using partial resource routines	1-41

Chapter 2

Scrap Manager 2-1

Figure 2-1	Copying and pasting data between two applications using the scrap	2-5
Figure 2-2	Writing both standard formats to the scrap	2-8
Figure 2-3	Using a private scrap	2-9
Figure 2-4	Intelligent cut and paste	2-11
Figure 2-5	Non-intelligent cut and paste	2-11
Figure 2-6	Location of the scrap in memory	2-13
Table 2-1	Actions your application performs in response to editing commands	2-6
Listing 2-1	Writing data to the scrap	2-16
Listing 2-2	Writing data to a private scrap	2-18
Listing 2-3	Copying data from the scrap in response to suspend events	2-19
Listing 2-4	Handling the Paste command using the scrap	2-21
Listing 2-5	Handling the Paste command using a private scrap	2-24
Listing 2-6	Handling resume events	2-25
Listing 2-7	Converting data between the scrap and a private scrap	2-27
Listing 2-8	Using TextEdit to handle the Cut command	2-29
Listing 2-9	Using TextEdit to handle the Paste command	2-30

Chapter 3

Help Manager 3-1

Figure 3-1	The Help menu for the Finder	3-7
Figure 3-2	A help balloon drawn with the standard balloon definition function	3-8
Figure 3-3	The tip and hot rectangle for a help balloon	3-9
Figure 3-4	Standard balloon positions and their variation codes	3-10
Figure 3-5	Alternate positions of a help balloon	3-11
Figure 3-6	Default help balloons for the window frame	3-15
Figure 3-7	Default help balloons for the Apple and Help menus	3-16
Figure 3-8	Default help balloons for application and document icons	3-17
Figure 3-9	Help balloons for different states of the Cut command	3-29
Figure 3-10	A help balloon for an enabled menu title	3-37
Figure 3-11	A help balloon for a dimmed menu title	3-37
Figure 3-12	A help balloon for a menu title dimmed by the Dialog Manager	3-38
Figure 3-13	A help balloon for menu items dimmed by the Dialog Manager	3-38
Figure 3-14	A help balloon for a menu item	3-39
Figure 3-15	A help balloon for a dimmed menu item	3-40
Figure 3-16	Help balloons for a changing menu item	3-45
Figure 3-17	A help balloon in a modal dialog box	3-61
Figure 3-18	Static and dynamic windows	3-64

Figure 3-19	A tool palette with a help balloon	3-70
Figure 3-20	A help balloon for a dialog box with a title	3-72
Figure 3-21	Default and custom help balloons for an application icon	3-86
Figure 3-22	The Help menu with an appended menu item	3-90
Figure 3-23	Structure of a compiled menu help ('hmnu') resource	3-133
Figure 3-24	Structure of an 'hmnu' component compiled with the HMStringResItem identifier	3-135
Figure 3-25	Structure of an 'hmnu' component compiled with the HMStringResItem identifier	3-136
Figure 3-26	Structure of an 'hmnu' component compiled with the HMPictItem, HMTEResItem, or HMSTRResItem identifier	3-137
Figure 3-27	Structure of an 'hmnu' component compiled with the HMSkipItem identifier	3-138
Figure 3-28	Structure of a menu-item component compiled with the HMCompareItem identifier	3-139
Figure 3-29	Structure of a menu-item component compiled with the HMNamedResourceItem identifier	3-140
Figure 3-30	Structure of a compiled dialog-item help ('hdlg') resource	3-141
Figure 3-31	Structure of an 'hdlg' component compiled with the HMStringItem identifier	3-144
Figure 3-32	Structure of an 'hdlg' component compiled with the HMStringResItem identifier	3-145
Figure 3-33	Structure of an 'hdlg' component compiled with the HMPictItem, HMTEResItem, or HMSTRResItem identifier	3-146
Figure 3-34	Structure of an 'hdlg' component compiled with the HMSkipItem identifier	3-148
Figure 3-35	Structure of a compiled rectangle help ('hrct') resource	3-149
Figure 3-36	Structure of an 'hrct' component compiled with the HMStringItem identifier	3-150
Figure 3-37	Structure of an 'hrct' component compiled with the HMStringResItem identifier	3-151
Figure 3-38	Structure of an 'hrct' component compiled with the HMPictItem, HMTEResItem, or HMSTRResItem identifier	3-152
Figure 3-39	Structure of an 'hrct' component compiled with the HMSkipItem identifier	3-153
Figure 3-40	Structure of a compiled window help ('hwin') resource	3-155
Figure 3-41	Structure of a compiled Finder icon help ('hfdr') resource	3-157
Figure 3-42	Structure of an 'hfdr' component compiled with the HMStringItem identifier	3-158
Figure 3-43	Structure of an 'hfdr' component compiled with the HMStringResItem identifier	3-158
Figure 3-44	Structure of an 'hfdr' component compiled with the HMPictItem, HMTEResItem, or HMSTRResItem identifier	3-159
Figure 3-45	Structure of an 'hfdr' component compiled with the HMSkipItem identifier	3-160
Figure 3-46	Structure of a compiled default help override ('hovr') resource	3-161
Figure 3-47	Structure of an 'hovr' component compiled with the HMStringItem identifier	3-163
Figure 3-48	Structure of an 'hovr' component compiled with the HMStringResItem identifier	3-163
Figure 3-49	Structure of an 'hovr' component compiled with the HMPictItem, HMTEResItem, or HMSTRResItem identifier	3-164

Figure 3-50	Structure of an 'hovr' component compiled with the HMSkipItem identifier	3-165
Listing 3-1	Rez input for a partial 'hmnu' resource	3-31
Listing 3-2	Rez input for the missing-items component of an 'hmnu' resource	3-35
Listing 3-3	Rez input for corresponding 'hmnu' and 'STR#' resources	3-41
Listing 3-4	Rez input for an 'hmnu' resource that uses HMCompareItem for a changing menu item	3-44
Listing 3-5	Rez input for specifying help messages with named resources	3-46
Listing 3-6	Specifying an alternate 'hmnu' resource for a menu that your application disables when it displays movable modal dialog boxes	3-49
Listing 3-7	Reassigning 'hmnu' resources before displaying a movable modal dialog box	3-50
Listing 3-8	Rez input for an item list resource and an 'hdlg' resource	3-59
Listing 3-9	Rez input for corresponding 'hwin' and 'hrcr' resources	3-71
Listing 3-10	Rez input for specifying help for titled and untitled windows	3-72
Listing 3-11	Using a string resource as the help message for HMShowBalloon	3-77
Listing 3-12	Using a picture resource as the help message for HMShowBalloon	3-77
Listing 3-13	Using a handle to a picture resource as the help message for HMShowBalloon	3-78
Listing 3-14	Using a string list resource as the help message for HMShowBalloon	3-79
Listing 3-15	Using styled text resources as the help message for HMShowBalloon	3-80
Listing 3-16	Using HMShowBalloon to display help balloons	3-82
Listing 3-17	Rez input for creating an 'hfdr' resource for an application icon	3-86
Listing 3-18	Rez input for an 'hovr' resource	3-89
Listing 3-19	Rez input for specifying help balloons for items in the Help menu	3-91
Listing 3-20	Responding to the user's choice in a menu command	3-92
Listing 3-21	Using the HMExtractHelpMsg function	3-124
Listing 3-22	Using a tip function	3-131

Chapter 4

List Manager	4-1	
Figure 4-1	A one-column, text-only list without a scroll bar	4-4
Figure 4-2	A one-column, text-only list with a vertical scroll bar	4-5
Figure 4-3	A list whose scroll bar has been disabled	4-6
Figure 4-4	A deactivated list	4-6
Figure 4-5	A list containing multiple columns and graphical elements	4-7
Figure 4-6	A list of items whose cells display more than one type of information	4-8
Figure 4-7	A list with an item selected	4-9
Figure 4-8	Selection of a range of items in a list	4-10

Figure 4-9	Effect of dragging after Shift-clicking	4-11
Figure 4-10	Selection of discontinuous items in a list	4-12
Figure 4-11	Effect of Shift-clicking in a list that contains discontinuous items	4-13
Figure 4-12	Notifying the user of nonstandard list behavior	4-14
Figure 4-13	Response to pressing the Command–Up Arrow keys	4-16
Figure 4-14	Response to user making a discontinuous selection, then pressing Shift–Right Arrow followed by Shift–Left Arrow using the extend algorithm	4-17
Figure 4-15	Response to Shift–Right Arrow using the anchor algorithm	4-19
Figure 4-16	An outlined list in a window with more than one list	4-21
Figure 4-17	Coordinates of cells	4-22
Figure 4-18	Selection flags	4-38
Figure 4-19	The Chooser’s use of a custom list definition procedure	4-58
Listing 4-1	Creating a list with a vertical scroll bar	4-27
Listing 4-2	Installing a list in a dialog box	4-29
Listing 4-3	Drawing a border around a list	4-30
Listing 4-4	Adding items from a string list to a one-column, text-only list	4-31
Listing 4-5	Responding to a mouse-down event in a list	4-33
Listing 4-6	Responding to an update event in a list	4-33
Listing 4-7	Finding the first selected cell in a list	4-34
Listing 4-8	Finding the last selected cell in a list	4-35
Listing 4-9	Selecting a cell and deselecting other cells	4-36
Listing 4-10	Scrolling so that a particular cell is visible	4-37
Listing 4-11	Clearing all cell data	4-40
Listing 4-12	Getting a copy of the data of a cell	4-41
Listing 4-13	Directly accessing a cell’s data	4-41
Listing 4-14	Adding an item to a one-column, alphabetical text list	4-42
Listing 4-15	A match function	4-43
Listing 4-16	Searching a list for a cell containing certain text or the next cell alphabetically	4-44
Listing 4-17	Resetting variables related to type selection	4-46
Listing 4-18	Selecting an item in response to a key-down event	4-47
Listing 4-19	Determining the location of a new cell in response to an arrow-key event	4-49
Listing 4-20	Moving the selection in response to an arrow-key event	4-50
Listing 4-21	Extending the selection in response to an arrow-key event	4-51
Listing 4-22	Processing an arrow-key event	4-52
Listing 4-23	Drawing an outline around a list	4-54
Listing 4-24	Adding a list to the ring	4-55
Listing 4-25	Updating the outline of all lists in a window	4-56
Listing 4-26	Moving the outline to the next list in a window	4-57
Listing 4-27	Moving the outline to the previous list in a window	4-57
Listing 4-28	Processing messages to a list definition procedure	4-59
Listing 4-29	Using the default initialization method	4-60
Listing 4-30	Responding to the <code>lDrawMsg</code> message	4-61
Listing 4-31	Responding to the <code>lHiliteMsg</code> message	4-62
Listing 4-32	Responding to the <code>lCloseMsg</code> message	4-63
Listing 4-33	Setting the cell size of a list	4-63
Listing 4-34	Adding an icon to a list of icons	4-64

Chapter 5

Icon Utilities 5-1

- Figure 5-1** The ResEdit view of an icon 5-4
- Figure 5-2** An icon family 5-5
- Listing 5-1** Drawing the icon from an icon family that is best suited to the user's display 5-10
- Listing 5-2** Drawing the icon from an icon suite that is best suited to the display device 5-11
- Listing 5-3** Drawing a specific icon from an icon family or icon suite 5-12
- Listing 5-4** Manipulating icon data in memory 5-13
- Listing 5-5** Drawing an icon of resource type 'ICON' 5-14
- Listing 5-6** Drawing an icon of resource type 'ICON' with a specific alignment and transform 5-15
- Listing 5-7** Drawing an icon of resource type 'icn' 5-15
- Listing 5-8** Drawing an icon of resource type 'icn' with a specific alignment and transform 5-16
- Listing 5-9** Drawing an icon of resource type 'SICN' with a specific alignment and transform 5-16

Chapter 6

Component Manager 6-1

- Figure 6-1** The relationship between an application, the Component Manager, and components 6-5
- Figure 6-2** Supporting multiple component connections 6-34
- Figure 6-3** Interaction between the `componentFlags` and `componentFlagsMask` fields 6-40
- Figure 6-4** Format of a component file 6-84
- Figure 6-5** Structure of a compiled component ('`thng`') resource 6-85
- Table 6-1** Request codes 6-14
- Listing 6-1** Finding a component 6-9
- Listing 6-2** Opening a specific component 6-10
- Listing 6-3** Getting information about a component 6-10
- Listing 6-4** Using a drawing component 6-11
- Listing 6-5** A drawing component for ovals 6-16
- Listing 6-6** Responding to an open request 6-20
- Listing 6-7** Responding to a close request 6-21
- Listing 6-8** Responding to the can do request 6-22
- Listing 6-9** Responding to the setup request 6-26
- Listing 6-10** Responding to the draw request 6-27
- Listing 6-11** Responding to the erase request 6-27
- Listing 6-12** Responding to the click request 6-27
- Listing 6-13** Responding to the move to request 6-28
- Listing 6-14** Registering a component 6-31
- Listing 6-15** Rez input for a component resource 6-33
- Listing 6-16** Delegating a request to another component 6-36

Figure 7-1	The Finder's application-unavailable alert box	7-5
Figure 7-2	The application-unavailable alert box for 'TEXT' and 'PICT' documents	7-5
Figure 7-3	The translation choices dialog box	7-6
Figure 7-4	A translation progress dialog box	7-7
Figure 7-5	The modified application-unavailable alert box	7-7
Figure 7-6	The enhanced file-opening dialog box	7-8
Figure 7-7	Document Converter configuration dialog box	7-9
Figure 7-8	A translation group with multiple source and destination types	7-29
Figure 7-9	A translation group with a single destination type	7-29
Figure 7-10	Point-to-point translation	7-30
Figure 7-11	Structure of a compiled open ('open') resource	7-44
Figure 7-12	Structure of a compiled kind ('kind') resource	7-45
Listing 7-1	Translation-specific selectors and response bit for Gestalt	7-12
Listing 7-2	A sample resource of type 'open'	7-13
Listing 7-3	A sample resource of type 'kind'	7-15
Listing 7-4	Sample resources for a translation extension	7-22
Listing 7-5	Handling Component Manager request codes	7-25
Listing 7-6	Creating a file translation list	7-30
Listing 7-7	Identifying file types	7-33
Listing 7-8	Translating a document	7-34

Figure 8-1	Two control panels, each with its own window	8-5
Figure 8-2	The General Controls control panel	8-6
Figure 8-3	Control panel icons in the Control Panels folder	8-9
Figure 8-4	The Monitors control panel	8-10
Figure 8-5	An Options dialog box for the SurfBoard video card	8-11
Figure 8-6	The River control panel interface	8-13
Figure 8-7	An icon for the River control panel file	8-14
Figure 8-8	The Color control panel	8-15
Figure 8-9	Coordinates defining the rectangles of the River control panel display area	8-16
Figure 8-10	Example of an inactive control panel	8-34
Figure 8-11	An Options dialog box with standard controls	8-49
Figure 8-12	An Options dialog box with superuser controls	8-50
Figure 8-13	The SurfBoard monitors extension icon	8-51
Figure 8-14	Display area defined by a rectangle resource	8-53
Figure 8-15	The SurfBoard Options dialog box with superuser controls	8-54
Figure 8-16	Structure of a compiled machine ('mach') resource	8-84
Figure 8-17	Structure of a compiled rectangle positions ('nrect') resource	8-85
Figure 8-18	Structure of a compiled font information ('finf') resource	8-86
Figure 8-19	Structure of a compiled card ('card') resource	8-87
Figure 8-20	Structure of a compiled rectangle ('RECT') resource	8-88

Table 8-1	Possible settings for the machine resource masks	8-21
Table 8-2	Error codes and their meaning	8-47
Table 8-3	Messages from the Finder	8-76
Table 8-4	Messages from the Monitors control panel	8-80
Table 8-5	Possible settings for the machine resource masks	8-85
Listing 8-1	Rez input for a rectangle positions list ('nrcct') resource	8-16
Listing 8-2	Rez input for an item list ('DITL') resource	8-18
Listing 8-3	Rez input for a machine ('mach') resource	8-21
Listing 8-4	Rez input for a file reference ('FREF') resource	8-21
Listing 8-5	Rez input for a signature resource	8-22
Listing 8-6	Rez input for a bundle ('BNDL') resource	8-22
Listing 8-7	A control panel's static text defined as user items	8-24
Listing 8-8	A control device function	8-27
Listing 8-9	Initializing a control panel: Allocating memory and setting controls	8-31
Listing 8-10	Responding to an activate event	8-35
Listing 8-11	Responding to a keyboard event	8-38
Listing 8-12	Responding to the user's interaction with controls	8-41
Listing 8-13	Responding to update events	8-43
Listing 8-14	Drawing text defined as user items	8-44
Listing 8-15	Terminating a control device function when the user closes the control panel	8-45
Listing 8-16	Responding to Edit menu commands	8-46
Listing 8-17	Rez input for a card ('card') resource	8-52
Listing 8-18	Rez input for a rectangle ('RECT') resource	8-53
Listing 8-19	Rez input for the SurfBoard monitors extension item list resource	8-55
Listing 8-20	Rez input for icon family resources for a monitors extension	8-57
Listing 8-21	Rez input for a version ('vers') resource	8-58
Listing 8-22	Rez input for the SurfBoard string list resource	8-59
Listing 8-23	Rez input for a file reference resource of a monitors extension	8-60
Listing 8-24	Rez input for a bundle resource of a monitors extension	8-60
Listing 8-25	A monitors extension function	8-64
Listing 8-26	Handling the startup message	8-66
Listing 8-27	Using a normal user rectangle or extending it to display superuser controls	8-67
Listing 8-28	Initializing a monitors extension	8-69
Listing 8-29	Drawing a line to separate superuser controls	8-70
Listing 8-30	Responding when a user clicks a control	8-72

About This Book

This book, *Inside Macintosh: More Macintosh Toolbox*, together with the book *Inside Macintosh: Macintosh Toolbox Essentials*, describes features you can build into your Macintosh application and documents the system software routines for implementing those features.

For information about events, windows, menus, controls, alert boxes, and dialog boxes and about how your application interacts with the Finder, see *Inside Macintosh: Macintosh Toolbox Essentials*.

This book, *More Macintosh Toolbox*, describes how you can enhance your application by supporting copy and paste and providing messages for help balloons. In addition, it describes other features you may want to use in your application, such as scrolling lists in dialog boxes and icons in windows. It also explains how to create resources, components, translation extensions, and control panels.

To read and write resources, see the chapter “Resource Manager.” This chapter describes how you can use resources to store the descriptions of user interface elements such as menus, windows, controls, dialog boxes, and icons. You can also use resources to store variable settings, such as the location of the window at the time the user closes it. When the user opens the document again, your application can read the information in the resource and restore the window to its previous location.

To support copy-and-paste operations in your application, see the chapter “Scrap Manager.” By using the Scrap Manager, you can allow users to copy and paste data between documents created by your application and documents created by other applications.

To provide messages for help balloons for elements of your application, see the chapter “Help Manager.” Help balloons are rounded-rectangle windows that contain explanatory information for the user. With tips pointing at the objects they annotate, help balloons look like the 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 the balloon’s hot rectangle.

To create lists in your application’s dialog boxes, including lists that contain scroll bars, see the chapter “List Manager.” You can use the List Manager to create one-column or multicolumn lists. Lists are useful for allowing the user to select one or more items from a group of items.

To display icons in a window or dialog box of your application, see the chapter “Icon Utilities.” By using Icon Utilities routines, you can automatically draw the icon from an icon family that is best suited for the current bit depth of the monitor.

To use or create components, see the chapter “Component Manager.” Components can provide your application with various services such as image compression or decompression services. You can also provide services to other applications by creating your own component.

To direct the translation of documents from one format to another, see the chapter “Translation Manager.” Macintosh Easy Open uses the Translation Manager to automatically provide some translation services for your application. Optionally, you can enhance your application’s interaction with Macintosh Easy Open or provide your own translation services.

To create a control panel or an extension to the Monitors control panel, see the chapter “Control Panels.” Control panels allow the user to set preferences for systemwide features, such as the speaker volume, desktop pattern, or picture displayed by a screen saver. Extensions to the Monitors control panel should be created only by the manufacturer of a video device.

To get information from the desktop database, see the chapter “Desktop Manager.” The desktop database contains information used by the Finder, such as icon definitions and their associated file types, as well as any comments that the user has added to the information window for desktop objects.

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

Some related topics can be found in other *Inside Macintosh* books. For information on how to read and write to the data fork of a file, see the chapter “Introduction to File Management” in *Inside Macintosh: Files*. For information about drawing into a window or other graphics port, see *Inside Macintosh: Imaging with QuickDraw*. 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 chapter “Resource Manager” contains these sections:

- n “Introduction to Resources.” This section presents a general introduction to resources, resource types, and resource forks.
- n “About the Resource Manager.” This section provides an overview of the features provided by the Resource Manager.
- n “Using the Resource Manager.” This section describes the tasks you can accomplish using the Resource Manager. It describes how to use the most common routines, gives related user interface information, provides code samples, and supplies additional information.

P R E F A C E

- n “Resource Manager Reference.” This section provides a complete reference to the Resource 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.
- n “Summary of the Resource Manager.” This section provides the Pascal and C interfaces for the constants, data structures, routines, and result codes associated with the Resource 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 not essential to an understanding of the main text. (An example appears on page 1-9.) u

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 1-5.) s

s WARNING

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 1-5.) 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

P R E F A C E

code samples in your application. You can find the location of code listings in the list of figures, tables, and listings beginning on page xvii. If you know the name of a particular routine (such as `DoPictBalloon` or `MyPlotAnICON`) 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.

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 referring to a sample application called *SurfWriter*; this book also refers to the sample applications *SurfPaint* and *SurfDB*. These applications are not actual products of Apple Computer, Inc. This book also refers to a River control panel and SurfBoard display card. These are not actual products of Apple Computer, Inc. In addition, the name River Change Systems is used to represent a fictitious company.

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
AppleLink:	APDA
America Online:	APDA
CompuServe:	76666,2405
Internet:	APDA@applelink.apple.com

P R E F A C E

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, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 303-2T
Cupertino, CA 95014-6299

Resource Manager

Contents

Introduction to Resources	1-3
The Data Fork and the Resource Fork	1-4
Resource Types and Resource IDs	1-6
The Resource Map	1-8
Search Path for Resources	1-10
About the Resource Manager	1-12
Using the Resource Manager	1-13
Creating a Resource	1-15
Getting a Resource	1-18
Releasing and Detaching Resources	1-22
Opening a Resource Fork	1-24
Opening an Application's Resource Fork	1-24
Creating and Opening a Resource Fork	1-25
Specifying the Current Resource File	1-28
Reading and Manipulating Resources	1-30
Writing Resources	1-36
Working With Partial Resources	1-40
Resource Manager Reference	1-42
Data Structure, Resource Types, and Resource IDs	1-42
The Resource Type	1-42
Resource IDs	1-46
Resource IDs of Owned Resources	1-47
Resource Names	1-49
Resource Manager Routines	1-49
Initializing the Resource Manager	1-50
Checking for Errors	1-51
Creating an Empty Resource Fork	1-53
Opening Resource Forks	1-58
Getting and Setting the Current Resource File	1-68
Reading Resources Into Memory	1-71

Getting and Setting Resource Information	1-81
Modifying Resources	1-87
Writing to Resource Forks	1-92
Getting a Unique Resource ID	1-95
Counting and Listing Resource Types	1-97
Getting Resource Sizes	1-104
Disposing of Resources	1-106
Closing Resource Forks	1-110
Reading and Writing Partial Resources	1-111
Getting and Setting Resource Fork Attributes	1-116
Accessing Resource Entries in a Resource Map	1-119
Resource File Format	1-121
Resources in the System File	1-126
User Information Resources	1-127
Packages	1-128
Function Key Resources	1-129
Standard Icons	1-129
ROM Resources	1-134
Inserting the ROM Resource Map	1-134
Overriding ROM Resources	1-135
Summary of the Resource Manager	1-137
Pascal Summary	1-137
Constants	1-137
Data Type	1-139
Routines	1-139
C Summary	1-142
Constants	1-142
Data Type	1-143
Routines	1-144
Assembly-Language Summary	1-147
Trap Macros	1-147
Global Variables	1-147
Result Codes	1-148

This chapter describes how to use the Resource Manager to read and write resources. You typically use resources to store the descriptions for user interface elements such as menus, windows, controls, dialog boxes, and icons. In addition, your application can store variable settings, such as the location of a window at the time the user closes the window, in a resource. When the user opens the document again, your application can read the information in the resource and restore the window to its previous location.

This chapter begins with an introduction to basic concepts you should understand before you begin to use Resource Manager routines. The rest of the chapter describes how to

- n create resources
- n get a handle to a resource
- n release and detach resources
- n create and open a resource fork
- n set the current resource file
- n read and manipulate resources
- n write resources
- n read and write partial resources

To use this chapter, you should be familiar with basic memory management on Macintosh computers and the Memory Manager. See the chapter “Introduction to Memory Management” in *Inside Macintosh: Memory* for details. You should also be familiar with the File Manager and the Standard File Package. See *Inside Macintosh: Files* for this information.

For information on how to create resources using a high-level resource editor like the ResEdit application or a resource compiler like Rez, see *ResEdit Reference* and *Macintosh Programmer’s Workshop Reference*. (Rez is provided with Apple’s Macintosh Programmer’s Workshop [MPW]; both MPW and ResEdit are available through APDA.)

To get information on the format of an individual resource type, see the documentation for the manager that interprets that resource. For example, to get the format of a 'MENU' resource, refer to the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Introduction to Resources

A **resource** is data of any kind stored in a defined format in a file’s resource fork. The Resource Manager keeps track of resources in memory and allows your application to read or write resources.

Resources are a basic element of every Macintosh application. Resources typically include data that describes menus, windows, controls, dialog boxes, sounds, fonts, and icons. Because such resources are separate from the application’s code, you can easily create and manage resources for menu titles, dialog boxes, and other parts of your

application without recompiling. Resources also simplify the process of translating interface elements containing text into other languages.

Applications and system software interpret the data for a resource 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 MPW). You can also use other resource tools, such as ResEdit, to create the resources for your application.

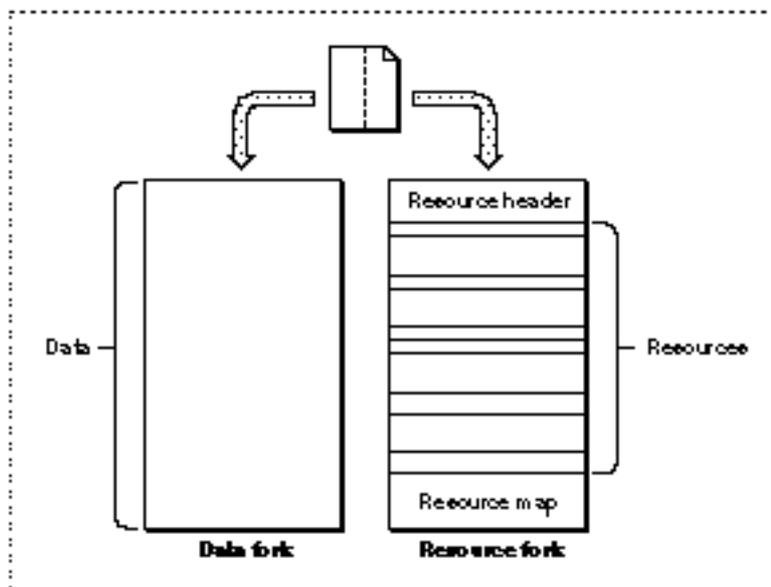
Inside Macintosh: Macintosh Toolbox Essentials describes how other managers, such as the Menu Manager, Window Manager, Dialog Manager, and Control Manager, use the Resource Manager to read resources for you. For example, you can use the Menu Manager, Window Manager, Dialog Manager, and Control Manager to read descriptions of your application's menus, windows, dialog boxes, and controls from resources. These managers all interpret a resource's data appropriately once it is read into memory. Although you'll typically use these managers to read resources for you, you can also use the Resource Manager directly to read and write resources.

The Data Fork and the Resource Fork

In Macintosh system software, a **file** is a named, ordered sequence of bytes stored on a volume and divided into two forks, the data fork and the resource fork. The **data fork** usually contains 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 header, the resources themselves, and a resource map.

Figure 1-1 shows the data fork and resource fork of a file.

Figure 1-1 The data fork and resource fork of a file



Resource Manager

The resource header includes offsets to the beginning of the resource data and to the resource map. The resource map includes information about the resources in the resource fork and offsets to the location of each resource.

A Macintosh file always contains both a resource fork and a data fork, although one or both of those forks can be empty. The data fork of a document file typically contains data created by the user, and the resource fork contains any document-specific resources, such as preference settings and the document's last window position. The resource fork of an application file (that is, any file with the file type 'APPL') typically includes resources that describe the application's menus, windows, controls, dialog boxes, and icons, as well as the application's 'CODE' resources. The resource fork of a file is also called a **resource file**, because in some respects you can treat it as if it were a separate file.

IMPORTANT

You should store all language-dependent data of your application, such as text used in help balloons and dialog boxes, as resources. If you do this, you can begin to localize your application by editing your application's resources without recompiling the application code. s

When your application writes data to a file, it writes to either the file's resource fork or its data fork. Typically, you use File Manager routines to read from and write to a file's data fork and Resource Manager routines to read from and write to a file's resource fork.

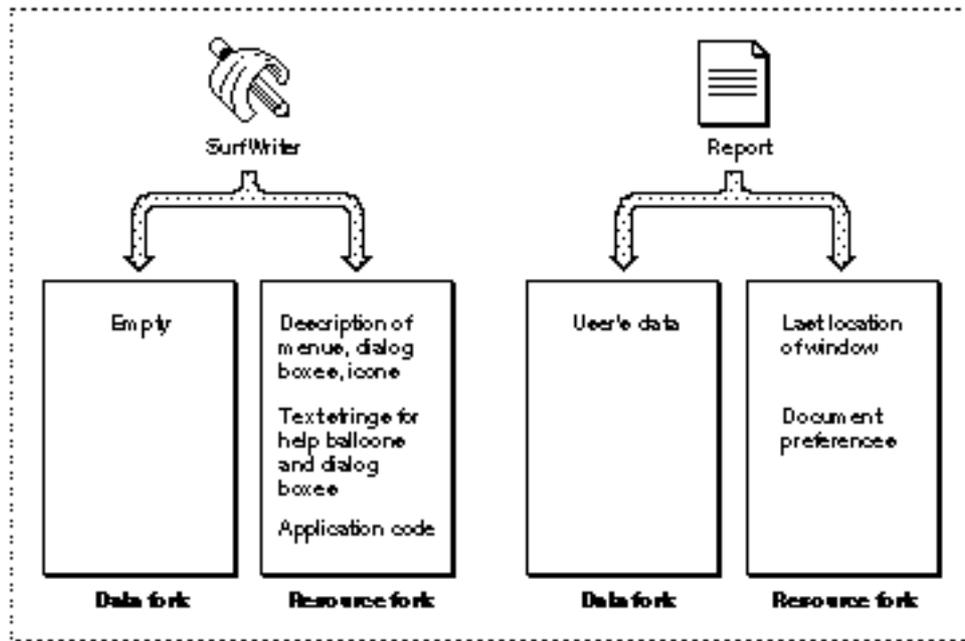
Whether you store data in the data fork or the resource fork of a document file depends largely on whether you can structure that data in a useful manner as a resource. For example, it's often convenient to store document-specific settings, such as the document's previous window size and location, as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

S WARNING

Don't use the resource fork of a file for data that is not in resource format. The Resource Manager assumes that any information in a resource fork can be interpreted according to the standard resource format described in this chapter. s

Figure 1-2 illustrates the typical contents of the data forks and resource forks of an application file and a document file.

Figure 1-2 An application's and a document's data fork and resource fork



A resource fork can contain at most 2727 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.

Resource Types and Resource IDs

You typically use resources to store structured data, 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 a specific resource of a given type by number. (You can also use a resource name instead of a resource ID to identify a resource of a given type. However, a resource ID is preferred because it's generally more convenient to generate unique numbers than unique names.)

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 differs from any other 'MENU' resources that you have defined. In general, resource numbers 128 through 32767 are available for your use, although the numbers you can use for some types of resources are more restricted. (See "Resource IDs" on page 1-46 for more information about restrictions on the resource IDs used with specific resource types.)

Resource Manager

System software defines a number of standard resource types. Here are some examples:

Resource type	Description
'ALRT'	Alert box
'CNTL'	Control
'CODE'	Application code segment
'DITL'	Item list in a dialog box or alert box
'DLOG'	Dialog box
'ICN#'	Large (32-by-32 pixel) black-and-white icon, with mask
'ICON'	Large (32-by-32 pixel) black-and-white icon, without mask
'MBAR'	Menu bar
'MENU'	Menu
'NFNT'	Bitmapped font
'STR'	String
'STR#'	String list
'WIND'	Window
'movv'	QuickTime movie
'snd'	Sound

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. See Table 1-2 on page 1-43 for a complete list of standard resource types.

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 for its own use in the System file's resource fork. Although many of these resources are used only by the system software, your application can use some of them if necessary. For example, the standard images for the I-beam and wristwatch cursors are stored as resources of type 'CURS' in the System file. Your application can use these resources to change the appearance of the cursor.

The Resource Map

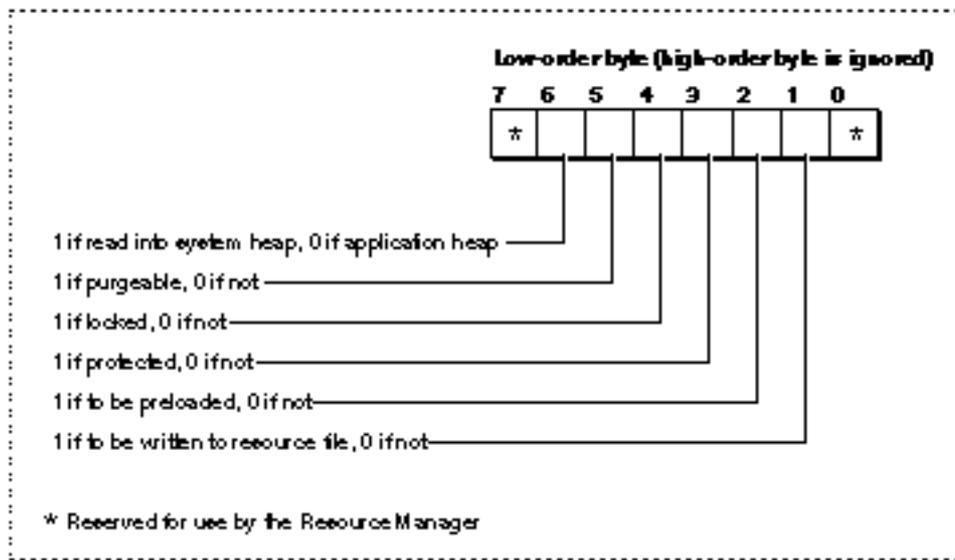
The **resource map** in the resource fork of a file contains entries for each resource in the resource fork. Each entry lists the resource's resource type, resource ID, name, attributes, and location. When the Resource Manager opens the resource fork of a file, it reads the resource map into memory. The resource map remains in memory until the file is closed.

The entries in the resource map on disk give the locations of resources as offsets to their locations in the resource fork. The entries in the resource map in memory specify the location of resources using handles—a handle whose value is `NIL`, if the resource is not currently in memory, or a handle to the resource's location in memory.

Resource attributes are flags that tell the Resource Manager how to handle the resource. For example, resource attributes specify whether the resource should be read into memory immediately when the Resource Manager opens the resource fork or read into memory only when needed; whether the resource should be read into the application or system heap; and whether the resource is purgeable.

The resource attributes for a resource are described by bits in the low-order byte of an integer value. Figure 1-3 shows which bits correspond to each resource attribute.

Figure 1-3 Resource attributes



When it first opens a resource fork, the Resource Manager examines the resource attributes for each resource listed in the resource map. If the preloaded attribute of the resource is set, the Resource Manager reads the resource into memory and specifies its location by setting the resource's resource map entry in memory to contain a handle to the resource data. If the preloaded attribute of the resource is not set, the Resource Manager does not read the resource into memory; instead, it specifies the resource's location in the resource map entry in memory with a handle whose value is `NIL`.

Resource Manager

When searching for a resource, the Resource Manager always looks in the resource map in memory, not the resource map of the resource fork on disk. If the resource map in memory specifies a handle for a particular resource, the Resource Manager uses the resource in memory; if the resource map in memory specifies a handle whose value is `NIL`, the Resource Manager reads the resource from the resource fork on disk into memory.

You can set the system heap attribute of a resource if you want to read a resource into the system heap. In most cases you should not set this attribute. If you do not set the system heap attribute, the Resource Manager reads the resource into relocatable blocks of your application's heap.

The purgeable attribute specifies whether the Resource Manager can purge a resource from memory 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 still in memory before referring to it through its resource handle.

Some resources must not be purgeable. For example, the Menu Manager expects menu resources to remain in memory, so you should not set the purgeable attribute of a menu resource. Other resources, such as windows, controls, and dialog boxes, do not have to remain in memory once the corresponding user interface element has been created. You should set the purgeable attribute for these kinds of resources.

You can set the locked attribute of a resource if you do not want the resource to be relocatable or purgeable. The locked attribute overrides the purgeable attribute; when the locked attribute is set, the resource is not purgeable, even if the purgeable attribute is set.

Note

If both the preloaded attribute and the locked attribute are set, the Resource Manager loads the resource as low in the heap as possible. [u](#)

You can set the protected attribute of a resource to ensure that your application doesn't accidentally change the resource ID or name of the resource, modify its contents, or remove the resource from its resource fork. In most cases you do not need to set this attribute. If you do set the protected attribute of a resource, you can still use a Resource Manager routine to change the protected attribute or to set other attributes of the resource.

The changed attribute applies only while the resource map is in memory. You should specify a value of 0 for the bit representing the changed attribute of a resource stored on disk. The Resource Manager sets the changed attribute of a resource's entry in the resource map in memory whenever your application changes a resource using the `ChangedResource` procedure, changes a resource map entry using the `SetResAttrs` or `SetResInfo` procedure, or adds a resource using the `AddResource` procedure.

Search Path for Resources

When your application uses a Resource Manager routine to read or perform an operation on a resource, the Resource Manager follows a defined search path to find the resource. The file whose resource fork the Resource Manager searches first is referred to as the **current resource file**. Whenever your application opens a resource fork of a file, that file becomes the current resource file. Thus, the current resource file usually corresponds to the file whose resource fork was opened most recently. However, your application can change the current resource file if needed by using the `UseResFile` procedure.

Most of the Resource Manager routines assume that the current resource file is the file on whose resource fork they should operate or, in the case of a search, the resource fork in which to begin the search. If the Resource Manager can't find the resource in the current resource file, it continues searching until it either finds the resource or has searched all files in the search path.

On startup, system software calls the `InitResources` function to initialize the Resource Manager. The Resource Manager creates a special heap zone within the system heap and builds a resource map that points to ROM-resident resources. It opens the resource fork of the System file and reads its resource map into memory.

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 the Resource Manager searches for a resource, it normally looks first in the resource map in memory of the last resource fork that your application opened. So, if your application has a single file open, the Resource Manager looks first in the resource map for that file's resource fork. If the Resource Manager doesn't find the resource there, it continues to search the resource maps of each resource fork open to your application in reverse order of opening (that is, the most recently opened is searched first). After looking in the resource maps of the resource files your application has opened, the Resource Manager searches your application's resource map. If it doesn't find the resource there, it searches the System file's resource map.

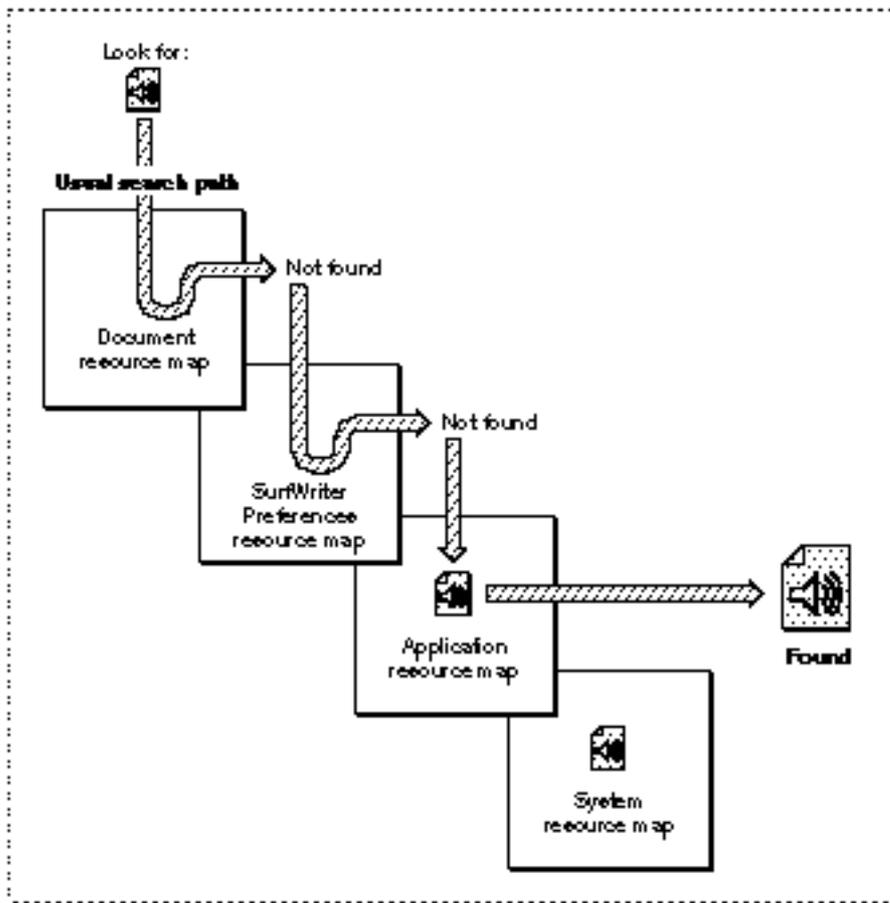
This default search order allows your application to use resources defined in the System file, to override resources defined in the System file, to share a single resource among several files by storing it in your application's resource fork, and to override application-defined resources with document-specific resources.

When the Resource Manager opens a resource fork, the File Manager assigns that resource fork a **file reference number**, which is a unique number identifying an access path to the resource fork. Your application needs to keep track of the file reference number of its own resource fork, so that it can refer specifically to that resource fork when necessary. Your application may also need to keep track of the file reference numbers for other resource forks that it opens.

For example, the SurfWriter application stores in its own resource fork the first few bars of Beethoven's Fifth Symphony as a resource of type `'snd'`. The SurfWriter application plays this sound whenever the user writes more than one page of text per hour. The user can change this sound for all documents created by SurfWriter by using SurfWriter's Preferences command to specify or record a new sound.

SurfWriter also allows the user to associate a sound with a specific document by using SurfWriter's Set Reward Sound command to specify or record a new sound. When SurfWriter wants to play the sound, it uses the Resource Manager to read the resource of type 'snd' with the resource ID kProductiveWriter. Figure 1-4 shows the search path the Resource Manager takes to find this sound resource.

Figure 1-4 A typical search order for a specific resource



System software opens SurfWriter's resource fork when the user opens the SurfWriter application. On startup, SurfWriter opens its preferences file (SurfWriter Preferences). When the user opens a SurfWriter document, SurfWriter opens the document's data fork and resource fork. When SurfWriter attempts to read an 'snd' resource, the Resource Manager looks first in the resource map in memory of the current resource file (in the example illustrated in Figure 1-4, the SurfWriter document) for the requested resource. If the Resource Manager doesn't find the resource, it searches the resource map of the next most recently opened file (in this example, SurfWriter Preferences). It continues searching the resource forks in memory of any resource forks open to the SurfWriter

application until it either finds the resource or has searched the last resource map in its search path. Typically the last resource map searched by the Resource Manager is the resource map of the System file. This allows your application to use resources in the System file as a default.

Table 1-1 summarizes the typical locations of resources used by an application.

Table 1-1 Typical locations of resources

Resource fork	Resources contained in resource fork
Resource fork of System file	Sounds, icons, cursors, and other elements available for use by all applications, and code resources that manage user interface elements such as menus, controls, and windows
Resource fork of application	Static data (such as text used in dialog boxes or help balloons) and descriptions of menus, windows, controls, icons, and other elements
Resource fork of application's preferences file	Data that encodes the user's global preferences for the application
Resource fork of document	Data that defines characteristics specific only to this document, such as its last size and location

Although you can take advantage of the Resource Manager's search order to find a particular resource, in general your application should set the current resource file to the file whose resource fork contains the desired resource before reading and writing resource data. In addition, you can restrict the Resource Manager search path by using Resource Manager routines that look only in the current resource file's resource map when searching for a specific resource.

About the Resource Manager

The Resource Manager provides routines that allow your application (and system software) to create, delete, open, read, modify, and write resources; get information about them; and alter the Resource Manager's search path.

Most Macintosh applications commonly read data from resources either indirectly, by calling other system software routines (such as Menu Manager routines) that in turn call the Resource Manager, or directly, by calling Resource Manager routines. At any time during your application's execution, at least two resource forks from which it can read information are likely to be open: the System file's resource fork and your application's resource fork.

As previously described, system software opens the System file's resource fork at startup and your application's resource fork at application launch. Your application is likely to open the resource forks of several other files at various times while it is running. For example, if your application saves the last position and size of a window (as determined by the user), you can use Resource Manager routines to write this information to an application-defined resource in the document file's resource fork. The next time the user opens the document, your application can use the Resource Manager to read the information saved in this resource and position the document accordingly.

You can store the user's general preferences, such as the default font or paper size, in your application's preferences file. You store a preferences file in the Preferences folder of the System Folder. The name of an application's preferences file typically consists of the name of the application followed by the word "Preferences." If your application can be shared by multiple users, you can use the Resource Manager to create a separate preferences file for each user.

Using the Resource Manager

You use the Resource Manager to perform operations on resources. To determine whether certain features of the Resource Manager are available (support for `FSSpec` records and partial resources), use the `Gestalt` function.

Two commonly used Resource Manager routines use a file system specification (`FSSpec`) record: the `FSpCreateResFile` procedure and the `FSpOpenResFile` function. These routines are available only in System 7 or later. Call the `Gestalt` function with the `gestaltFSAttr` selector to determine whether the Resource Manager routines that use `FSSpec` records exist. If the bit indicated by the constant `gestaltHasFSSpecCalls` is set, then the routines are available.

```
CONST
    gestaltFSAttr          = 'fs  ';    {Gestalt selector for }
                                { File Mgr attributes}
    gestaltHasFSSpecCalls = 1;        {check this bit in the }
                                { response parameter}
```

In addition, the Resource Manager routines for reading and writing partial resources are available only in System 7 or later versions of system software. Use the `Gestalt` function to determine whether these features are available. Call the `Gestalt` function with the `gestaltResourceMgrAttr` selector to determine whether the routines for handling partial resources exist. If the bit indicated by the constant `gestaltPartialRsrcs` is set, then the Resource Manager routines for handling partial resources are available. For more information about the `Gestalt` function, see *Inside Macintosh: Operating System Utilities*.

Resource Manager

```

CONST
    gestaltResourceMgrAttr = 'rsrc';    {Gestalt selector for }
                                     { Resource Mgr attributes}
    gestaltPartialRsrcs    = 0;        {check this bit in the }
                                     { response parameter}

```

You can use the `ResError` function to retrieve errors that may result from calling Resource Manager routines. Resource Manager procedures do not report error information directly. Instead, after calling a Resource Manager procedure your application should call the `ResError` function to determine whether an error occurred.

Resource Manager functions usually return `NIL` or `-1` as the function result when there's an error. For Resource Manager functions that return `-1`, your application can call the `ResError` function to determine the specific error that occurred. For Resource Manager functions that return handles, your application should always check whether the value of the returned handle is `NIL`. If it is, your application can use `ResError` to obtain specific information about the nature of the error. Note, however, that in some cases `ResError` returns `noErr` even though the value of the returned handle is `NIL`.

The rest of this section describes how to create a resource using `ResEdit` or the `Rez` resource compiler. It then describes how to use Resource Manager routines to

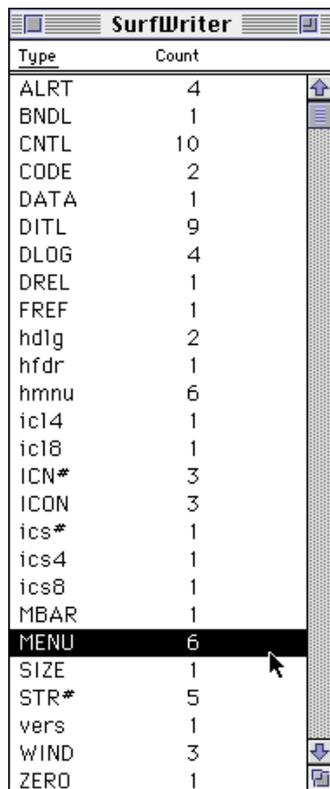
- n get a handle to a resource and modify a purgeable resource safely
- n release and detach resources
- n create and open a resource fork
- n set the current resource file (the file whose resource fork the Resource Manager searches first)
- n read and manipulate resources
- n write resources
- n read and write partial resources

For detailed descriptions of all Resource Manager routines, see “Resource Manager Reference” beginning on page 1-42. For information on writing data to a file's data fork, see *Inside Macintosh: Files*.

Creating a Resource

You typically define the user interface elements of your application, such as menus, windows, dialog boxes, and controls, by specifying descriptions of these elements in resources. You can then use Menu Manager, Window Manager, Dialog Manager, or Control Manager routines to create these elements—based on their resource descriptions—as needed. You can create resource descriptions using a resource editor, such as ResEdit, which lets you create the resources in a visual manner; or you can provide a textual, formal description of resources in a file and then use a resource compiler, such as Rez, to compile the description into a resource. Figure 1-5 shows the window ResEdit displays for the SurfWriter application. This window lists all of the resources in the resource fork of the SurfWriter application.

Figure 1-5 The ResEdit window for the SurfWriter application

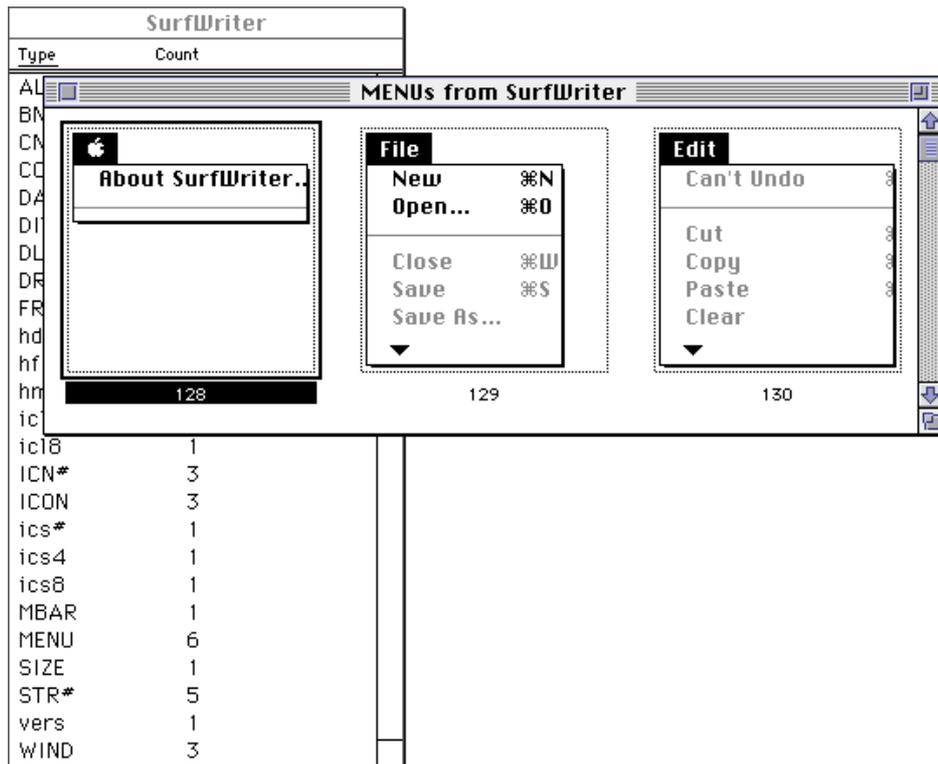


Type	Count
ALRT	4
BNDL	1
CNTL	10
CODE	2
DATA	1
DITL	9
DLOG	4
DREL	1
FREF	1
hdlg	2
hfdrr	1
hmnu	6
icl4	1
icl8	1
ICN*	3
ICON	3
ics*	1
ics4	1
ics8	1
MBAR	1
MENU	6
SIZE	1
STR*	5
vers	1
WIND	3
ZERO	1

Resource Manager

You can use ResEdit to examine any of your application's resources. For example, to view your application's 'MENU' resources, double-click that resource in the ResEdit window. Figure 1-6 shows how ResEdit displays the menus of the SurfWriter application.

Figure 1-6 The menus of the SurfWriter application



Listing 1-1 shows the definition of SurfWriter's Apple menu in Rez input format.

Listing 1-1 A menu in Rez input format

```
#define mApple 128

resource 'MENU' (mApple, preload) { /*resource ID, preload resource*/
    mApple,                          /*menu ID*/
    textMenuProc,                    /*uses standard menu definition */
                                      /* procedure*/
    0b11111111111111111111111111111101, /*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*/
        noicon, nokey, nomark, plain; /*item characteristics*/
                                      /*second menu item*/
        "-",                            /*item text (divider)*/
        noicon, nokey, nomark, plain /*item characteristics*/
    }
};
```

Your application can also create, modify, and save resources as needed using various Resource Manager routines.

You can store your application-specific resources in the application file itself. You need not add resources to your application after it is created. Instead, store any document-specific resources in the relevant document and store user preferences in a preferences file in the Preferences folder of the System Folder.

To retrieve resources from your application's resource fork, you usually use other managers (such as the Menu Manager or Window Manager). To retrieve resources other than menus, windows, dialog boxes, or controls, you usually use Resource Manager routines.

To retrieve a resource from a document file or a preferences file, your application needs to open the resource fork of the file and then use Resource Manager routines to retrieve any resources in the file. The section that follows, “Getting a Resource,” describes how the Resource Manager returns a handle to a resource at your application’s request and how to modify a purgeable resource safely. The sections “Opening a Resource Fork” and “Reading and Manipulating Resources” beginning on page 1-24 and page 1-30, respectively, describe in detail how to use Resource Manager routines to open and read resources.

Getting a Resource

You usually use the `GetResource` function to read data from resources other than menus, windows, dialog boxes, and controls. You supply the resource type and resource ID of the desired resource, and the `GetResource` function searches the resource maps of open resource forks (according to the search path described in “Search Path for Resources” beginning on page 1-10) for that resource’s entry.

If the `GetResource` function finds an entry for the requested resource in the resource map and the resource is in memory (that is, if the resource map in memory does not specify the resource’s location with a handle whose value is `NIL`), `GetResource` returns a handle to the resource. If the resource is listed in the resource map but is not in memory (the resource map in memory specifies the resource’s location with a handle whose value is `NIL`), `GetResource` reads the resource data from disk into memory, replaces the entry for the resource’s location with a handle to the resource, and returns to your application a handle to the resource. For a resource that cannot be purged (that is, whose purgeable attribute is not set) you can use the returned handle to refer to the resource in other Resource Manager routines. (Handles to purgeable resources are discussed later in this section.)

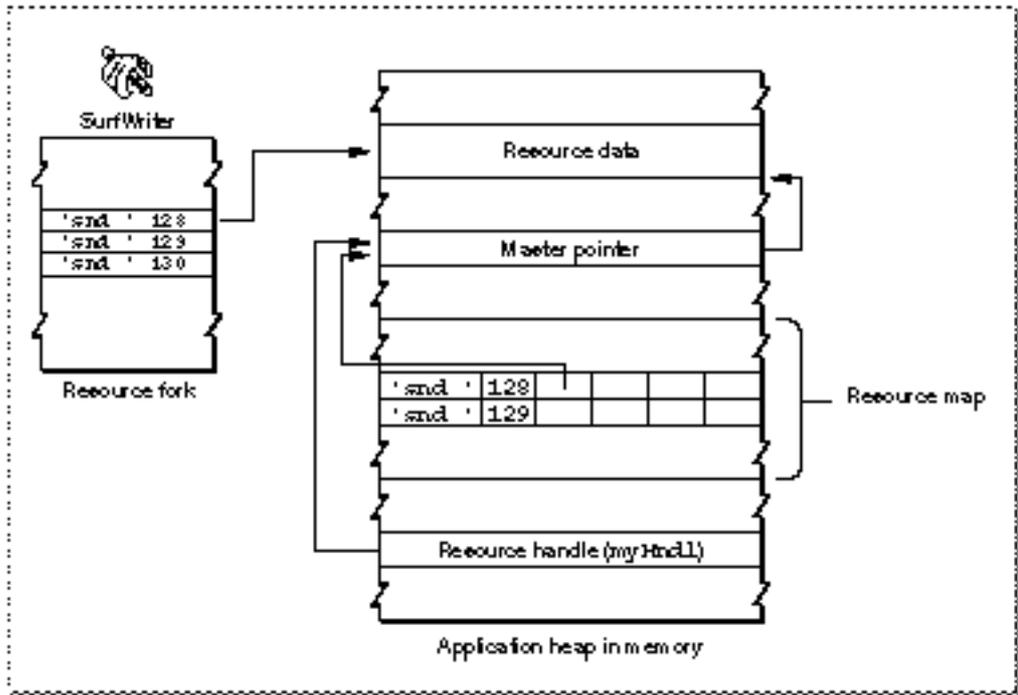
For example, this code uses `GetResource` to get a handle to an ‘snd’ resource with resource ID 128.

```
VAR
    resourceType: ResType;
    resourceID:   Integer;
    myHndl:      Handle;

resourceType := 'snd';
resourceID   := 128;
myHndl := GetResource(resourceType, resourceID);
```

Figure 1-7 shows how `GetResource` returns a handle to a resource at your application's request.

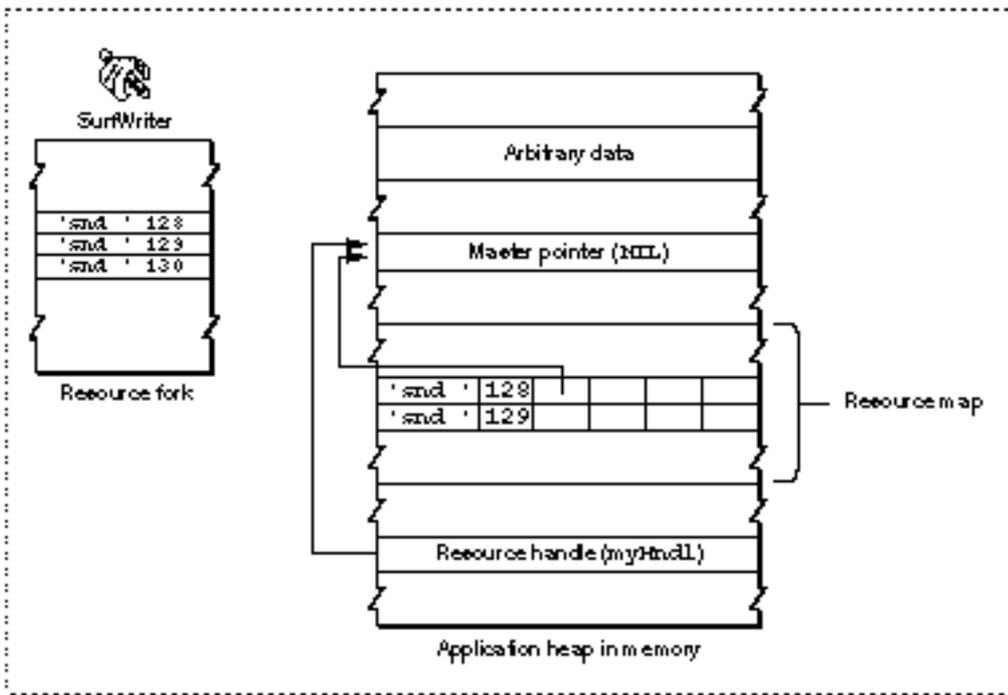
Figure 1-7 Getting a handle to a resource



Note that the handle returned to your application is a copy of the handle in the resource map. The resource map contains a handle to the resource data, and the Resource Manager returns a handle to the same block of memory for use by your application. If you use `GetResource` to get a handle to a resource that has the purgeable attribute set or if you intend to modify such a resource, keep the following discussion in mind.

If a resource is marked purgeable and the Memory Manager determines that it must purge a resource to make more room in your application's heap, it releases the memory occupied by the resource. In this case, the handle to the resource data is no longer valid, because the handle's master pointer is set to `NIL`. If your application attempts to use the handle previously returned by the Resource Manager, the handle no longer refers to the resource. Figure 1-8 shows a handle to a resource that is no longer valid, because the Memory Manager has purged the resource. To avoid this situation, you should call the `LoadResource` procedure to make sure that the resource is in memory before attempting to refer to it.

Figure 1-8 A handle to a purgeable resource after the resource has been purged



If you need to make changes to a purgeable resource using routines that may cause the Memory Manager to purge the resource, you should make the resource temporarily not purgeable. You can use the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState` for this purpose. After calling `HGetState` and `HNoPurge`, change the resource as necessary. To make the changes permanent, use the `ChangedResource` and `WriteResource` procedures; then call `HSetState` when you're finished. Listing 1-2 illustrates the use of these routines.

Listing 1-2 Safely changing a resource that is purgeable

```

VAR
    resourceType: ResType;
    resourceID: Integer;
    myHndl: Handle;
    state: SignedByte;

resourceType := 'snd ';
resourceID := 128;
{read the resource into memory}
myHndl := GetResource(resourceType, resourceID);
state := HGetState(myHndl); {get the state of the handle}
HNoPurge(myHndl);          {mark the handle as not purgeable}
{modify the resource as needed}
{...}
ChangedResource(myHndl);   {mark the resource as changed}
WriteResource(myHndl);     {write the resource to disk}
HSetState(myHndl, state);  {restore the handle's state}

```

Although you'll usually want to use `WriteResource` to write a resource's data to disk immediately (as shown in Listing 1-2), you can instead use the `SetResPurge` procedure and specify `TRUE` in the `install` parameter. If you do this, the Memory Manager calls the Resource Manager before purging data specified by a handle. The Resource Manager determines whether the passed handle is that of a resource in your application's heap, and, if so, calls `WriteResource` to write the resource to disk if its changed attribute is set. You can call the `SetResPurge` procedure and specify `FALSE` in the `install` parameter to restore the normal state, so that the Memory Manager purges resource data in memory without checking with the Resource Manager.

Releasing and Detaching Resources

When you've finished using a resource, you can call `ReleaseResource` to release the memory associated with that resource. For a given resource, the `ReleaseResource` procedure releases the memory associated with the resource, setting the handle's master pointer to `NIL`, thus making your application's handle to the resource invalid. (This is similar to the situation shown in Figure 1-8.) After releasing a resource, use another Resource Manager routine if you need to use the resource again. For example, the code in Listing 1-3 first uses `GetResource` to get a handle to a resource, manipulates the resource, then uses `ReleaseResource` when the application has finished using the resource. If the application needs the resource later, it must get a valid handle to the resource by reading the resource into memory again (using `GetResource`, for example).

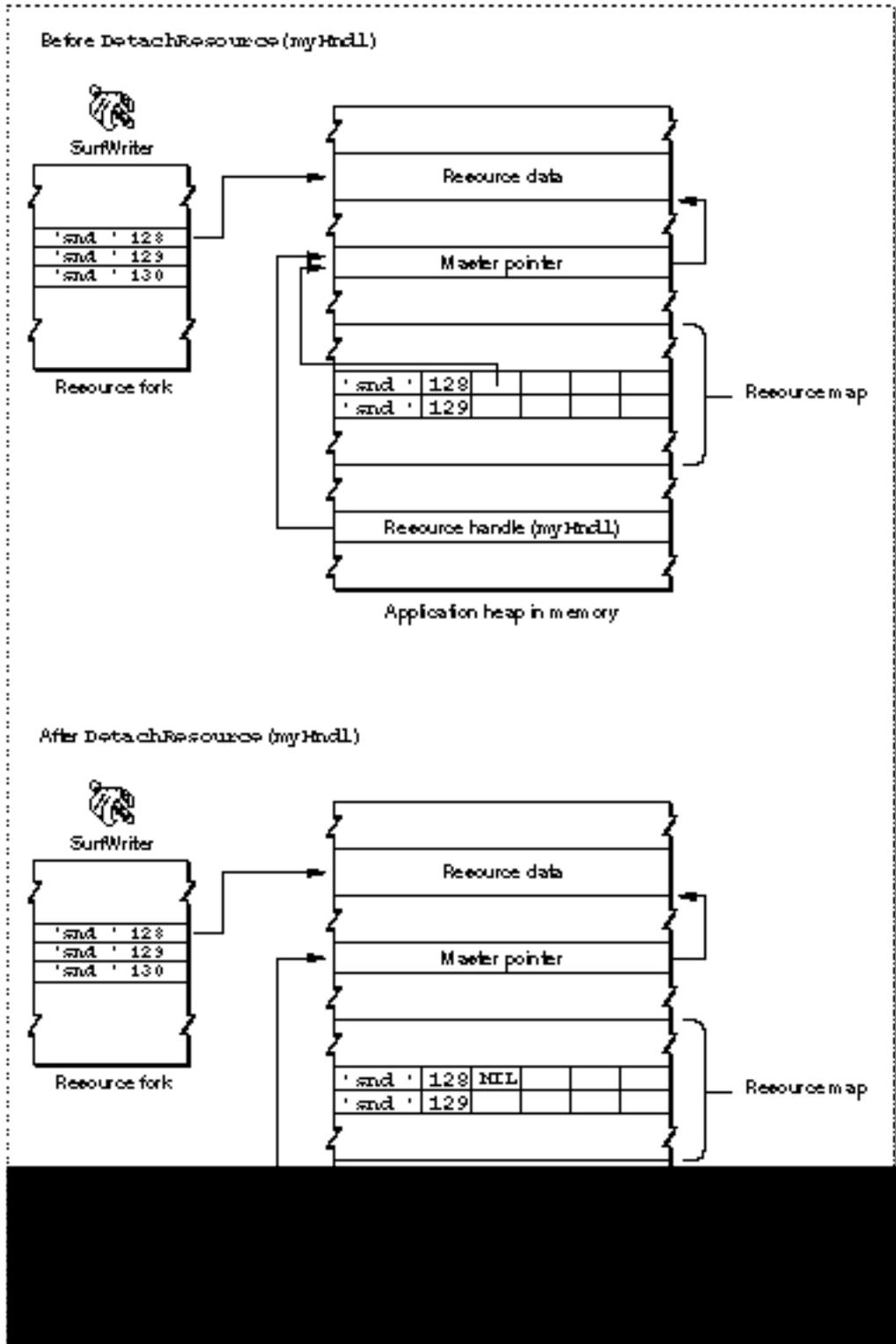
Listing 1-3 Releasing a resource

```
PROCEDURE MyGetAndPlaySoundResource(resourceID: Integer);
VAR
    myHndl: Handle;
BEGIN
    myHndl := GetResource('snd ', resourceID);
    {use the resource}
    {when done, release the resource}
    ReleaseResource(myHndl);
END;
```

Your application can also use the `DetachResource` procedure to replace a resource's handle in the resource map with a handle whose value is `NIL`. However, the `DetachResource` procedure does not release the memory associated with the resource. You can use `DetachResource` when you want your application to access the resource's data directly, without the aid of the Resource Manager, or when you need to pass the handle to a routine that does not accept a resource handle. (For example, the `AddResource` routine used in Listing 1-4 on page 1-24 takes a handle to data, not a handle to a resource.) Once you detach a resource, the Resource Manager does not recognize the resource's handle in the resource map in memory as a valid handle to a resource, but your application can still manipulate the resource's data through its own handle to the data.

Figure 1-9 shows how both your application and the Resource Manager have a handle to a resource after your application calls `GetResource`. The figure also shows how the Resource Manager replaces the handle in the resource map in memory with a handle whose value is `NIL` when your application calls `DetachResource`.

Figure 1-9 Detaching a resource



You can also easily copy a resource by first reading in the resource using `GetResource`, detaching the resource using `DetachResource`, then copying the resource by using `AddResource` (and specifying a new resource ID). Listing 1-4 uses this technique to copy a resource within the current resource file.

Listing 1-4 Detaching a resource

```
PROCEDURE MyCopyAResource(resourceType: ResType;
                          resourceID: Integer;
                          VAR myHndl: Handle);
VAR
  newResourceID: Integer;
BEGIN
  myHndl := GetResource(resourceType, resourceID);
  DetachResource(myHndl);           {detach the resource}
  newResourceID := UniqueID(resourceType);
  AddResource(myHndl, resourceType, newResourceID, '');
END;
```

Opening a Resource Fork

When your application opens a file's resource fork or data fork, the File Manager returns a file reference number. You use a file reference number in File Manager routines (and in a few Resource Manager routines) to identify a unique access path to an open fork of a specific file. Even though the file reference number of the data fork and the resource fork usually match, you should use the file reference number of a file's resource fork in Resource Manager routines; don't assume that it has the same value as the file reference number for the same file's data fork.

Opening an Application's Resource Fork

Because system software automatically opens your application's resource fork when the user opens your application, you do not need to open it explicitly. However, you should save your application's file reference number. You can do this by calling the `CurResFile` function early in your initialization procedure. (The `CurResFile` function returns the file reference number of the current resource file.) Listing 1-5 shows the part of SurfWriter's initialization procedure that gets the file reference number of the application's resource fork.

Listing 1-5 Getting the file reference number for your application's resource fork

```

PROCEDURE MyInitialize;
BEGIN
    MaxApplZone;           {extend heap zone to limit}
    MoreMasters;          {get 64 more master pointers}
    MoreMasters;          {get 64 more master pointers}
    InitGraf(@thePort);   {initialize QuickDraw}
    InitFonts;            {initialize Font Manager}
    InitWindows;         {initialize Window Manager}
    TEInit;               {initialize TextEdit}
    InitDialogs(nil);     {initialize Dialog Manager}
    InitCursor;           {set cursor to arrow}
    {get the file ref num of this app's resource file }
    { and save it in a global variable}
    gAppsResourceFork := CurResFile;
    {do other initialization}
END;

```

SurfWriter uses an application-defined global variable (`gAppsResourceFork`) to refer to its resource fork in subsequent calls to Resource Manager routines.

Creating and Opening a Resource Fork

To save resources in the resource fork of a file, you must first create the resource fork (if it doesn't already exist in a form that can be used by the Resource Manager) and obtain a file reference number for it. After creating a new resource fork, you must open it before writing any resources to it. You'll usually want to save the file reference number of any resource fork that your application opens.

To create a resource fork, use the `FSpCreateResFile` procedure. This procedure requires four parameters: a file-system specification record (identifying the name and location of the file), the signature of the application creating the file, the file type of the file, and the script code for the file.

A file system specification record is a standard format for identifying a file or directory. The file system specification record for files and directories is available in System 7 and later versions of system software and is defined by the `FSSpec` data type.

```

TYPE FSSpec = {file system specification}
RECORD
    vRefNum: Integer;      {volume reference number}
    parID:   LongInt;     {directory ID of parent directory}
    name:    Str63;       {filename or directory name}
END;

```

Certain File Manager routines—those that open a file’s data fork—also take a file system specification record as a parameter. You can use the same `FSSpec` record in Resource Manager routines that create or open the file’s resource fork.

If the file specified by the `FSSpec` record doesn’t already exist (that is, if the file has neither a data fork nor a resource fork), the `FSpCreateResFile` procedure creates a resource file—that is, a resource fork, including a resource map. In this case, the file has a zero-length data fork. The `FSpCreateResFile` procedure also sets the creator, type, and script code fields of the file’s catalog information record to the specified values.

If the file specified by the `FSSpec` record already exists and includes a resource fork with a resource map, `FSpCreateResFile` does nothing, and the `ResError` function returns an appropriate result code. If the data fork of the file specified by the `FSSpec` record already exists but the file has a zero-length resource fork, `FSpCreateResFile` creates an empty resource fork and resource map for the file; it also changes the creator, type, and script code fields of the catalog information record of the file to the specified values.

Listing 1-6 shows a function that creates a new resource fork, including a resource map.

Listing 1-6 Creating an empty resource fork

```
FUNCTION MyCreateResourceFork (myFSSpec: FSSpec): OSErr;
BEGIN
    FSpCreateResFile(myFSSpec, gAppSignature, 'TEXT',
                    smSystemScript);
    MyCreateResourceFork := ResError;
END;
```

After creating a resource fork, you can open it using the `FSpOpenResFile` function. The `FSpOpenResFile` function returns a file reference number that you can use to change or limit the Resource Manager’s search order or to close a resource fork.

After opening a resource fork, you can write resources to it using the routines described in “Writing Resources” beginning on page 1-36. (You can also write to a resource fork using File Manager routines; in general, you should use the Resource Manager.) When you are finished using a resource fork that your application has specifically opened, you should close it using the `CloseResFile` procedure. The Resource Manager automatically closes any resource forks opened by your application that are still open when your application calls `ExitToShell`.

Listing 1-7 shows a routine that uses the application-defined function `MyCreateResourceFork` (shown in Listing 1-6) to create a new resource fork, opens the resource fork, writes resources to it, then closes the resource fork when it is finished.

Listing 1-7 Creating and opening a resource fork

```

FUNCTION MyCreateAndOpenResourceFork (myFSSpec: FSSpec): OSErr;
VAR
    myErr:      OSErr;
    myRefNum:   Integer;
BEGIN
    {create a resource file}
    myErr := MyCreateResourceFork(myFSSpec);
    IF myErr = noErr THEN {open the resource file}
        myRefNum := FSpOpenResFile(myFSSpec, fsRdWrPerm);
    IF ResError = noErr THEN {write to the resource file}
        myErr := MyWriteResourcesToFile(myRefNum);
    CloseResFile(myRefNum); {close the resource file}
    MyCreateAndOpenResourceFork := myErr;
END;

```

Note that when you open a resource fork, the Resource Manager resets the search path so that the file whose resource fork you just opened becomes the current resource file. For example, suppose the SurfWriter application file is open, and the user opens document A, then document B. SurfWriter opens the resource forks of both documents. In this case, the search order is

1. document B (the current resource file)
2. document A
3. the SurfWriter application
4. the System file

If the user is working with document A and SurfWriter uses the `UseResFile` procedure to set the current resource file to document A, the new search order is

1. document A (the current resource file)
2. the SurfWriter application
3. the System file

If the user opens another document, document C, and SurfWriter opens its resource fork, the new search order becomes

1. document C (the current resource file)
2. document B
3. document A
4. the SurfWriter application
5. the System file

Specifying the Current Resource File

When you request a resource, the Resource Manager follows the search order described in “Search Path for Resources” on page 1-10. To change the starting point of the search or to restrict the search to the resource fork of a specific file, you can use `CurResFile` and `UseResFile`. To get the file reference number for the current resource file, use the `CurResFile` function. You can then use the `UseResFile` procedure to set the current resource file to the desired file, use other Resource Manager routines to retrieve any desired resources, and then use `UseResFile` again to restore the current resource file to its previous setting.

For example, the SurfWriter application allows users to specify or record either a special reward sound that applies only to a specific document or a general reward sound that can apply to any document. SurfWriter stores a document-specific reward sound resource in the document and the general reward sound resource in either the SurfWriter Preferences file (if the reward sound is user-defined) or in the application file. If several documents are open and SurfWriter needs to play a document-specific reward sound, SurfWriter attempts to get the sound from that document without searching the resource forks of any other documents that might be open. If the document doesn't have the specified reward sound, SurfWriter searches for the sound in the resource fork of the preferences file and, if necessary, of the application file and System file.

Listing 1-8 shows how the SurfWriter application uses `CurResFile` and `UseResFile` to get and play the appropriate reward sound for a given document. All reward sounds share the same resource ID, `kProductiveWriter`. The application-defined procedure `MyGetAndPlayRewardSoundResource` first checks whether the reward sound setting for the document specifies a sound stored in that document or a general reward sound stored in the preferences file or elsewhere. If the document has a reward sound, the procedure sets the current resource file to that document, searches just that document's resource fork for the sound, and plays the sound. If the document doesn't have a reward sound, the `MyGetAndPlayRewardSoundResource` procedure sets the current resource file to SurfWriter Preferences, searches the entire resource chain from that point on for the sound, and plays the sound. This scheme ensures that SurfWriter always plays the correct reward sound, even if different reward sound resources in different documents share the same resource ID.

Listing 1-8 Saving and restoring the current resource file

```

PROCEDURE MyGetAndPlayRewardSoundResource (refNum: Integer);
VAR
    myHndl:      Handle;
    prevResFile: Integer;
BEGIN
    prevResFile := CurResFile; {save the current resource file}
    IF MyHasDocumentRewardSound(refNum) THEN
        BEGIN
            {first set the current resource file to a specific document}
            UseResFile(refNum);
            {get reward sound from the document using Get1Resource }
            { to limit search to current resource file and avoid }
            { searching the resource forks of any other open documents}
            myHndl := Get1Resource('snd ', kProductiveWriter);
        END
    ELSE
        BEGIN
            {set current resource file to SurfWriter Preferences}
            UseResFile(gSurfPrefsResourceFork);
            {get reward sound resource using GetResource to search }
            { entire resource chain starting with preferences file}
            myHndl := GetResource('snd ', kProductiveWriter);
        END;
    IF myHndl <> NIL THEN
        BEGIN
            MyPlayThisSound(myHndl);
            ReleaseResource(myHndl);
        END;
    UseResFile(prevResFile);{restore the current resource }
                                { file to its previous setting}
END;

```

The `MyGetAndPlayRewardSoundResource` procedure saves the reference number of the current resource file and then calls the application-defined routine `MyHasDocumentRewardSound` to check whether the document has a reward sound associated with it. If so, `MyGetAndPlayRewardSoundResource` sets the current resource file to the value specified by the `refNum` parameter. The procedure then uses the `Get1Resource` function to get, from the current resource file, a handle to the resource of type `'snd '` with the ID specified by `kProductiveWriter`.

If the document doesn't have a specified reward sound, `MyGetAndPlayRewardSoundResource` uses `UseResFile` to set the current resource file to the SurfWriter Preferences file's resource fork and `GetResource` to search the entire resource chain from that point. If `GetResource` locates a resource with the specified resource ID in the SurfWriter Preferences file, it returns a handle to that resource; if not, it continues to search until it finds the specified resource or reaches the end of the resource chain. This ensures that the procedure won't get a user-defined resource with the same resource ID in some other SurfWriter document that is currently open instead of the general reward sound saved in SurfWriter Preferences or in SurfWriter itself.

If the call to `Get1Resource` or `GetResource` is successful (that is, if it does not return a handle whose value is `NIL`), `MyGetAndPlayRewardSoundResource` plays the appropriate reward sound, then uses `ReleaseResource` to release the memory occupied by the sound resource. Finally, the procedure uses `UseResFile` to restore the current resource file to its previous setting.

Reading and Manipulating Resources

The Resource Manager provides a number of routines that read resources from a resource fork. When you request a resource, the Resource Manager follows the search path described in "Search Path for Resources" on page 1-10. That is, the Resource Manager searches each resource fork open to your application, beginning with the current resource file, and continues until it either finds the resource or reaches the end of the chain.

You can change where the Resource Manager starts its search using the `UseResFile` procedure. (See the previous section, "Specifying the Current Resource File," for details.) You can limit the search to only the current resource file by using the Resource Manager routines that contain a "1" in their names, such as `Get1Resource`, `Get1NamedResource`, `Get1IndResource`, `Unique1ID`, and `Count1Resources`.

To get a resource, you can specify it by its resource type and resource ID or by its resource type and resource name. By convention, most applications refer to a resource by its resource type and resource ID, rather than by its resource type and resource name.

Resource Manager

You can use the `SetResLoad` procedure to enable and disable automatic loading of resource data into memory for routines that return handles to resources. Such routines normally read the resource data into memory if it's not already there. This is the default setting and the effect of calling `SetResLoad` with the `load` parameter set to `TRUE`. If you call `SetResLoad` with the `load` parameter set to `FALSE`, subsequent calls to routines that return handles to resources will not load the resource data into memory. Instead, such routines return a handle whose master pointer is set to `NIL` unless the resource is already in memory. This setting is useful when you want to read from the resource map without reading the resource data into memory. To read the resource data into memory after a call to `SetResLoad` with the `load` parameter set to `FALSE`, call `LoadResource`.

S WARNING

If you call `SetResLoad` with the `load` parameter set to `FALSE`, be sure to call `SetResLoad` with the `load` parameter set to `TRUE` as soon as possible. Other parts of system software that call the Resource Manager rely on the default setting (the `load` parameter set to `TRUE`), and some routines won't work if resources are not loaded automatically. **s**

In addition to the `SetResLoad` procedure, you can use the `preloaded` attribute of an individual resource to control loading of that resource's data into memory. The Resource Manager loads a resource into memory when it first opens a resource fork if the resource's `preloaded` attribute is set.

Note

If both the `preloaded` attribute and the `locked` attribute are set, the Resource Manager loads the resource as low in the heap as possible. **u**

Here's an example of a situation in which an application might need to read a resource. The SurfWriter application always saves the last position of a document window when the user saves the document, storing this information in a resource that it has defined for this purpose. SurfWriter defines a resource with resource type `rWindowState` and resource ID `kLastWindowStateID` to store information about the window (its position and its state—that is, either the user state or the standard state). SurfWriter's window state resource has this format, defined by a record of type `MyWindowState`:

```
TYPE MyWindowState =
  RECORD
    userStateRect: Rect;      {user state rectangle}
    zoomState: Boolean; {window state: TRUE = standard; }
                          { FALSE = user}
  END;

MyWindowStatePtr = ^MyWindowState;
MyWindowStateHnd = ^MyWindowStatePtr;
```

Listing 1-9 shows a procedure called `MySetWindowPosition` that the `SurfWriter` application uses in the process of opening a document. The `SurfWriter` application stores the last location of a document in its window state resource. When `SurfWriter` opens the document again, it uses `MySetWindowPosition` to read the document's window state resource and uses the resource data to set the window's location.

Listing 1-9 Getting a resource from a document file

```
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;
END;
```

Resource Manager

```

{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));
SetPort(savePort);
END;

```

The `MySetWindowPosition` procedure uses the `FSpOpenResFile` function to open the document's resource fork, then uses `Get1Resource` to get a handle to the resource that contains information about the window's last position. The procedure can then verify that the saved position is practical and move the window to that position.

Note that when a Resource Manager routine returns a handle to a resource, the routine returns the resource using the `Handle` data type. You usually define a data type (such as `MyWindowState`) to access the resource's data. If you also define a handle to your defined data type (such as `MyWindowStateHnd`), you need to coerce the returned handle to the appropriate type, as shown in this line from Listing 1-9:

```
myStateHandle := MyWindowStateHnd(Get1Resource(rWinState, kLastWinStateID));
```

If you use this method, you also need to coerce your defined handle back to a handle of type `Handle` when you use other Resource Manager routines. For example, after it has finished moving the window, `MySetWindowPosition` uses `ReleaseResource` to release the memory allocated to the resource's data (which also sets the master pointer of the resource's handle in the resource map in memory to `NIL`). As shown in this line from Listing 1-9, `SurfWriter` coerces the defined handle back to a handle:

```
ReleaseResource(Handle(myStateHandle));
```

After releasing the resource data's memory, `MySetWindowPosition` uses the `CloseResFile` procedure to close the resource fork.

Note

Listing 1-9 assumes the window state resource is not purgeable. If it were, `MySetWindowPosition` would need to call `LoadResource` before accessing the data in the resource. \cup

The Resource Manager also provides routines that let you index through all resources of a given type (for example, using `CountResources` and `GetIndResource`). This can be useful whenever you want to read all the resources of a given type.

Listing 1-10 shows an application-defined procedure that allows a user to open a file that contains sound resources. The `SurfWriter` application opens the specified file, counts the number of 'snd' resources in the file, then performs an operation on each 'snd' resource (adding the name of each resource to its Sounds menu).

Listing 1-10 Counting and indexing through resources

```
PROCEDURE MyDoOpenSoundResources;
VAR
    mySFReply:      StandardFileReply; {reply record}
    myNumTypes:    Integer;           {number of types to display}
    myTypeList:    SFTypeList;       {file type of files}
    myRefNum:      Integer;           {resource file reference no}
    mySndHandle:   Handle;            {handle to sound resource}
    numberOfSnds:  Integer;           {# of sounds in resource file}
    index:         Integer;           {index of sound resource}
    resName:       Str255;            {name of sound resource}
    curRes:        Integer;           {saved current resource file}
    myType:        ResType;           {resource type}
    myResID:       Integer;           {resource ID of snd resource}
    myWindow:     WindowPtr;         {window pointer}
    menu:          MenuHandle;        {handle to Sounds menu}
    myErr:         OSErr;             {error information}
```

Resource Manager

```

BEGIN
  curRes := CurResFile;
  myWindow := FrontWindow;
  MyDoActivate(myWindow, FALSE);    {deactivate front window}
  myTypeList[0] := 'SFSD';          {show files of this type}
  myNumTypes := 1;
  {let user choose a file that contains sound resources}
  StandardGetFile(NIL, myNumTypes, myTypeList, mySFReply);
  IF mySFReply.sfGood = TRUE THEN
  BEGIN
    myRefNum := FSpOpenResFile(mySFReply.sfFile, fsRdWrPerm);
    IF myRefNum = -1 THEN
      DoError;
    menu := GetMenuHandle(mSounds);
    numberOfSnds := Count1Resources('snd ');
    FOR index := 1 TO numberOfSnds DO
    BEGIN {the loop}
      mySndHandle := Get1IndResource('snd ', index);
      IF mySndHandle = NIL THEN
        DoError
      ELSE
        BEGIN
          GetResInfo(mySndHandle, myResID, myType, resName);
          AppendMenu(menu, resName);
          ReleaseResource(mySndHandle);
        END; {of mySndHandle <> NIL}
      END; {of the loop}
    UseResFile(curRes);
    gSoundResFileRefNum := myRefNum;
  END; {of sfReply.good}
END;

```

After the user selects a file that contains SurfWriter sound resources (that is, a file of type 'SFSD'), the `MyDoOpenSoundResources` procedure calls `FSpOpenResFile` to open the file's resource fork and obtain its file reference number. (If `FSpOpenResFile` fails to open the resource fork, it returns `-1` instead of a file reference number.) The `MyDoOpenSoundResources` procedure then uses the `Count1Resources` function to count the number of 'snd ' resources in the resource fork. It can then index through the resources one at a time, using `Get1IndResource` to open each resource, `GetResInfo` to get the resource's name, and `AppendMenu` to append each name to SurfWriter's Sounds menu.

Note

In most situations, you can use the Menu Manager procedure `AppendResMenu` to add names of resources to a menu. See *Inside Macintosh: Macintosh Toolbox Essentials* for details. u

Writing Resources

After opening a resource fork (as described in “Creating and Opening a Resource Fork” beginning on page 1-25), you can write resources to it. You can write resources only to the current resource file. To ensure that the current resource file is set to the appropriate resource fork, you can use `CurResFile` to save the file reference number of the current resource file, then `UseResFile` to set the current resource file to the desired resource fork.

To specify data for a new resource, you usually use the `AddResource` procedure, which creates a new entry for the resource in the resource map in memory and sets the entry’s location to refer to the resource’s data. Note that `AddResource` changes only the resource map in memory; it doesn’t change anything on disk. Use the `UpdateResFile` or `WriteResource` procedure to write the resource to disk. The `AddResource` procedure always adds the resource to the resource map in memory that corresponds to the current resource file. For this reason, you usually need to set the current resource file to the desired file before calling `AddResource`.

If you change a resource that is referenced through the resource map in memory, you use the `ChangedResource` procedure to set the `resChanged` attribute of that resource’s entry. You should then immediately call the `UpdateResFile` or `WriteResource` procedure to write the changed resource data to disk. Note that although the `UpdateResFile` procedure writes only those resources that have been added or changed to disk, it also writes the entire resource map to disk (overwriting its previous contents). The `WriteResource` procedure writes only the resource data of a single resource to disk; it does not update the resource’s entry in the resource map on disk.

The `ChangedResource` procedure reserves enough disk space to contain the changed resource. It does this every time it’s called, but the actual writing of the resource does not take place until a call to `WriteResource` or `UpdateResFile`. Thus, if you call `ChangedResource` several times on a large resource before the resource is actually written, you may unexpectedly run out of disk space, because many times the amount of space actually needed is reserved. When the resource is actually written, the file’s end-of-file (EOF) is set correctly, and the next call to `ChangedResource` will work as expected.

Resource Manager

IMPORTANT

If your application frequently changes the contents of resources (especially large resources), you should call `WriteResource` or `UpdateResFile` immediately after calling `ChangedResource`.

To ensure that the Resource Manager does not purge a purgeable resource while your application is in the process of changing it, use the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState`. First call `HGetState` and `HNoPurge`, then change the resource as necessary. To make a change to a resource permanent, use the `ChangedResource` and `WriteResource` (or `UpdateResFile`) procedures; then call `HSetState` when you're finished. (See Listing 1-2 on page 1-21 for an example of this technique.) However, most applications do not make resources purgeable and therefore don't need to take this precaution.

Here's an example of a situation in which an application might need to write a resource. As previously described, the SurfWriter application always saves the last position of a document window when the user saves the document, storing this information in a resource that it has defined for this purpose. SurfWriter defines a resource with resource type `rWinState` and resource ID `kLastWinStateID` to store the window state (its position and state, that is, either the user or the standard state). SurfWriter's window state resource has this format, defined by a record of type `MyWindowState`:

```

TYPE MyWindowState =
    RECORD
        userStateRect: Rect;    {user state rectangle}
        zoomState:    Boolean; {window state: TRUE = standard; }
                                {                FALSE = user}
    END;

MyWindowStatePtr = ^MyWindowState;
MyWindowStateHnd = ^MyWindowStatePtr;

```

Listing 1-11 shows SurfWriter's application-defined routine for saving the last position of a window in a window state resource in a document's resource fork.

Listing 1-11 Saving a resource to a resource fork

```

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 userStateRect 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));
        IF ResError <> noErr THEN
            DoError;
    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;
END;

```

Resource Manager

```

IF myStateHandle <> NIL THEN
BEGIN
    UpdateResFile(myResFileRefNum);
    ReleaseResource(Handle(myStateHandle));
END;
UseResFile(curResRefNum);
END;

```

The `MySaveWindowPosition` procedure first sets the `userStateRect` field of the window state record to the bounds of the current content region of the window. It also sets the `zoomState` field of the record to a Boolean value that indicates whether the window is currently in the user state or standard state. If the window is in the standard state, the procedure sets the `userStateRect` field of the window state record to the user state of the window. (SurfWriter always saves the user state and the last state of the window. When it opens a document, it sets the user state to its previous state, verifies that this position is still valid, then calculates the window's standard state.)

The `MySaveWindowPosition` procedure then saves the file reference number of the current resource file and sets the current resource file to the document displayed in the current window. The procedure then uses the `Get1Resource` function to determine whether the resource file of the document already contains a window state resource. If so, the procedure changes the resource data, then calls `ChangedResource` to set the `resChanged` attribute of the resource's entry of the resource map in memory. If the resource doesn't yet exist, the procedure simply adds the new resource using the `AddResource` procedure.

Note that when a Resource Manager routine returns a handle to a resource, it returns the resource using the `Handle` data type. You usually define a data type (such as `MyWindowState`) to access the resource's data. If you also define a handle to your defined data type (such as `MyWindowStateHnd`), you need to coerce the returned handle to the appropriate type, as shown in this line from Listing 1-11:

```
myStateHandle := MyWindowStateHnd(Get1Resource(rWinState, kLastWinStateID));
```

If you use this method, you also need to coerce your defined handle back to a handle of type `Handle` when you use other Resource Manager routines, as shown in this line from Listing 1-11:

```
AddResource(Handle(myStateHandle), rWinState, kLastWinStateID,
             'last window state');
```

After `MySaveWindowPosition` changes or adds the resource (affecting only the resource map and resource data in memory), the `MySaveWindowPosition` procedure makes the change permanent by calling `UpdateResFile` and specifying the file reference number of the resource fork to update on disk. The `UpdateResFile` procedure writes the entire resource map in memory to disk and updates the resource data of any resource whose `resChanged` attribute is set in the resource map in memory.

(If you want to update only the resource that was just changed or added, you can use `WriteResource` instead of `UpdateResFile`.)

Note

Listing 1-11 assumes the window state resource is not purgeable. If it were, `MySaveWindowPosition` would need to call `HGetState` and `HNoPurge` before changing the resource. u

When done with the resource, `MySaveWindowPosition` uses `ReleaseResource`, which releases the memory allocated to the resource's data (and at the same time sets the master pointer of the resource's handle in the resource map in memory to `NIL`). Then `MySaveWindowPosition` restores the current resource file to its previous setting.

Working With Partial Resources

Some resources, such as the 'snd ' and 'sfnt' resources, can be quite large—sometimes too large to fit in the available memory. The `ReadPartialResource` and `WritePartialResource` procedures, which are available in System 7 and later versions of system software, allow you to read a portion of the resource into memory or alter a section of the resource while it is still on disk. You can also use the `SetResourceSize` procedure to enlarge or reduce the size of a resource on disk. When you use `ReadPartialResource` and `WritePartialResource`, you specify how far into the resource you want to begin reading or writing and how many bytes you actually want to read or write at that spot, so you must be sure of the location of the data.

S WARNING

Be aware that having a copy of a resource in memory when you are using the partial resource routines may cause problems. For example, if you read the resource into memory using `GetResource`, modify the resource in memory, and then access the resource on disk using either the `ReadPartialResource` or `WritePartialResource` procedure, note that these procedures work with the data in the buffer you specify, not the data referenced through the resource's handle. s

To read or write any part of a resource, call the `SetResLoad` procedure specifying `FALSE` for its load parameter, then use the `GetResource` function to get an empty handle (that is, a handle whose master pointer is set to `NIL`) to the resource. (Because of the call to the `SetResLoad` procedure, the `GetResource` function does not load the entire resource into memory.) Then call `SetResLoad` specifying `TRUE` for its load parameter and use the partial resource routines to access portions of the resource.

Listing 1-12 illustrates one way to deal with partial resources. The application-defined procedure `MyReadAPartial` begins by calling `SetResLoad` (with the load parameter set to `FALSE`) to ensure that the Resource Manager will not attempt to read the entire resource into memory in the subsequent call to `GetResource`. After calling `GetResource` and checking for errors, `MyReadAPartial` calls `SetResLoad` (with the load parameter set to `TRUE`) to restore normal loading of resource data into memory. The procedure then calls `ReadPartialResource`, specifying as parameters the handle returned by `GetResource`, an offset to the beginning of the resource subsection to be read, a buffer into which to read the subsection, and the length of the subsection. The `ReadPartialResource` procedure reads the specified partial resource into the specified buffer.

Listing 1-12 Using partial resource routines

```

PROCEDURE MyReadAPartial(myRsrcType: ResType; myRsrcID: Integer;
                        start: LongInt; count: LongInt;
                        VAR putItHere: Ptr);
VAR
    myResHdl:          Handle;
    myErr:             OSErr;
BEGIN
    SetResLoad(FALSE);      {don't load resource}
    myResHdl := GetResource(myRsrcType, myRsrcID);
    myErr := ResError;
    SetResLoad(TRUE);      {reset to always load}
    IF myErr = noErr THEN
    BEGIN
        ReadPartialResource(myResHdl, start, putItHere, count);
        myErr := ResError;
        {check and report error}
        IF myErr <> noErr THEN DoError(myErr);
    END
    ELSE {handle error from GetResource}
        DoError(myErr);
END;

```

Resource Manager Reference

This section begins by describing the data type, standard resource types, and ranges of resource IDs used for various kinds of resources. “Resource Manager Routines” beginning on page 1-49 describes the routines provided by the Resource Manager for manipulating resources.

“Resource File Format” beginning on page 1-121 describes the format of a resource fork. “Resources in the System File” beginning on page 1-126 describes System file resources such as packages and icons. “ROM Resources” beginning on page 1-134 describes how to access ROM resources directly and how to override them.

Data Structure, Resource Types, and Resource IDs

This section describes the data type for the resource type, lists the standard resource types, and describes the ranges of resource IDs available to your application for different kinds of resources. The Resource Manager and your application use a resource type and a resource ID to identify a specific resource.

The Resource Type

The Resource Manager uses the resource type along with the resource ID to identify a resource uniquely. A resource type is defined by the `ResType` data type.

```
TYPE ResType = PACKED ARRAY[1..4] OF Char;
```

A resource type can be any sequence of four alphanumeric characters, including the space character. You can define your own resource types, but they must consist of all uppercase letters and must not conflict with any of the standard resource types.

IMPORTANT

When identifying resource types, the Resource Manager distinguishes between uppercase letters and their lowercase counterparts. In addition, Apple reserves for its own use all resource types that consist of all lowercase letters, all spaces, or all international characters (characters greater than \$7F). s

Table 1-2 lists the standard resource types.

Table 1-2 Standard resource types

Resource type	Description
'ADBS'	Apple Desktop Bus service routine
'ALRT'	Alert box
'BNDL'	Bundle
'CDEF'	Control definition function
'CDEV'	Control device function for a control panel
'CNTL'	Control
'CODE'	Application code segment
'CURS'	Cursor
'DITL'	Item list in a dialog or alert box
'DLOG'	Dialog box
'DRVR'	Desk accessory or other device driver
'FKEY'	Command-Shift-number combination
'FOND'	Font family record
'FONT'	Bitmapped font
'FREF'	File reference
'ICN#'	Large (32-by-32 pixel) black-and-white icon, with mask
'ICON'	Large (32-by-32 pixel) black-and-white icon, without mask
'INIT'	System extension
'KCAP'	Physical keyboard description (used by Key Caps desk accessory)
'KCHR'	Keyboard layout (software); maps virtual key codes to character codes
'LDEF'	List definition procedure
'MBAR'	Menu bar
'MDEF'	Menu definition procedure
'MENU'	Menu
'NFNT'	Bitmapped font
'PACK'	Package
'PAT'	Pattern
'PAT#'	Pattern list

continued

Table 1-2 Standard resource types (continued)

Resource type	Description
'PICT'	QuickDraw picture
'POST'	PostScript [®] resource
'PREC'	Print record
'SICN'	Small (16-by-16 pixel) icon (mask optional)
'SIZE'	Size of application's partition and other information
'STR'	String
'STR#'	String list
'WDEF'	Window definition function
'WIND'	Window
'actb'	Alert color table
'alis'	Alias record
'card'	Video card name
'cctb'	Control color table
'cicn'	Color icon
'clut'	Color look-up table
'crsr'	Color cursor
'dctb'	Dialog color table
'ddev'	Database extension
'eadr'	Ethernet hardware address
'fctb'	Font color table
'hdlg'	Help for dialog box or alert box items
'hfdr'	Help for application icons
'hmnv'	Help for application menus
'hovr'	Help that overrides Finder help
'hrct'	Help for areas in windows
'hwin'	Association of 'hrct' and 'hdlg' resources to specific windows
'icl4'	Large (32-by-32 pixel) color icon with 4 bits of color data per pixel
'icl8'	Large (32-by-32 pixel) color icon with 8 bits of color data per pixel
'ics#'	Small (16-by-16 pixel) black-and-white icon, with mask
'ics4'	Small (16-by-16 pixel) color icon with 4 bits of color data per pixel
'ics8'	Small (16-by-16 pixel) color icon with 8 bits of color data per pixel

Table 1-2 Standard resource types (continued)

Resource type	Description
'ictb'	Item color table
'itl0'	Date and time formats
'itl1'	Names of days and months
'itl2'	Text Utilities sort hooks
'itl4'	Localizable tables and code
'itlk'	Remappings of certain key combinations before the <code>KeyTrans</code> function is called for the corresponding 'KCHR' resource
'kcs#'	List of small black-and-white icons, with mask, for a corresponding 'KCHR' resource
'kcs4'	Small (16-by-16 pixel) color icon with 4 bits of color data per pixel for a corresponding 'KCHR' resource
'kcs8'	Small (16-by-16 pixel) color icon with 8 bits of color data per pixel for a corresponding 'KCHR' resource
'mctb'	Menu color information table
'mntr'	Monitors extension code resource
'movv'	QuickTime movie
'pltt'	Color palette
'ppat'	Pixel pattern
'qdef'	Query definition function
'qrsc'	Query resource
'sect'	Section record
'sfnt'	Outline font
'snd'	Sound
'snth'	Synthesizer
'styl'	TextEdit style record
'sysz'	System heap space required by a system extension
'vers'	Version number
'wctb'	Window color table
'wstr'	String (uses word for length byte)

Table 1-3 lists resource types that are reserved for use by system software. These resource types consist entirely of uppercase letters or combinations of uppercase and lowercase letters and the number sign (#). Other resource types specific to system software that consist entirely of lowercase letters or other characters are not included in Table 1-3. This list is provided for your information; you should not use these resource types in your application.

Table 1-3 Resource types reserved for use by system software

Resource type	Description
' CACH '	RAM cache code
' DSAT '	System startup alert table
' FCMT '	“Get Info” comments
' FMTR '	3.5-inch disk formatting code
' FOBJ '	Folder information for an MFS volume
' FRSV '	IDs of system fonts
' INTL '	International resource (obsolete)
' KMAP '	Keyboard mapping (hardware); maps raw key codes to virtual key codes
' KSWP '	Defines special key combinations for Script Manager operations
' MBDF '	Default menu definition function
' MMAP '	Mouse-tracking code
' NBPC '	AppleTalk bundle
' PDEF '	Printing code
' PTCH '	ROM patch code
' ROv# '	List of ROM resources to override
' ROvr '	Code for overriding ROM resources
' SERD '	RAM Serial Driver

Resource IDs

A resource is identified by its resource type and resource ID (or, optionally, its resource type and resource name). The IDs for resources used by the system software and those used by applications are assigned from separate ranges. By using these ranges correctly, you can avoid resource ID conflicts.

In general, system resources use IDs in the range –32767 through 127, and application resources must use IDs that fall between 128 and 32767. The IDs for some categories of resources, such as definition procedures and font families, fall in different ranges or in ranges that are broken down for more specific purposes. This list shows the resource ID ranges used for most resources.

Range	Description
–32768 through –16385	Reserved; do not use. Any application resource whose ID is in this range will not work properly in current versions of system software.
–16384 through –4065	Used for system resources owned by other system resources.
–4064 through –4033	Reserved for use by control panels. (See the chapter “Control Panels” in this book.)
–4032 through –1	Used for system resources owned by other system resources. The exception is the 'SIZE' resource, whose ID can be –1, 0 (preferred size), or 1 (minimum size).
0 through 127	Used for system resources and any definition procedures in the system software. Applications should not use these resource IDs.
128 through 32767	Available for your use. Your application's definition procedures should use IDs in the range 128 through 4095, although other resources may use these IDs as well. Font families for individual script systems have additional restrictions defined in the appendix on international resources in <i>Inside Macintosh: Text</i> .

For a general discussion of font family resource IDs, see *Inside Macintosh: Text*.

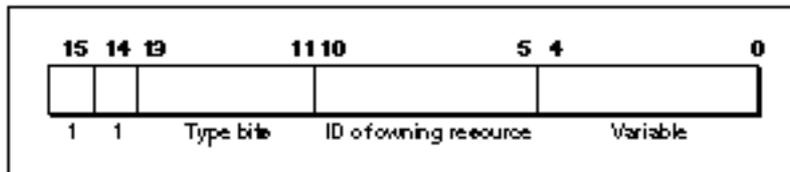
The ID range of definition procedures (which are usually contained in resources such as the 'WDEF' or 'CDEF' resources) is limited to 12 bits (0 through 4095). The system software's own definition procedures, which are located in the System file, have resource IDs from 0 through 127. The IDs of your definition procedures should be in the range 128 through 4095.

Resource IDs of Owned Resources

Certain types of resources used by system software may have resources of their own in the same resource fork; the “owning” resource consists of code that reads the “owned” resource into memory. For example, a desk accessory might have its own pattern and string resources. This section describes the numbering convention used for owned resources. This information can be useful if you are writing a desk accessory or other driver or special types of definition functions for windows, controls, or menus.

You should use the numbering convention described in this section to associate owned resources with the resources to which they belong. This allows resource-copying programs (such as installers) to recognize which additional resources need to be copied along with an owning resource. Figure 1-10 illustrates the ID of an owned resource.

Figure 1-10 Resource ID of an owned resource



Bits 14 and 15 are always 1. Bits 11 through 13 specify the type of the owning resource, as follows:

Type bits	Type
000	'DRVR'
001	'WDEF'
010	'MDEF'
011	'CDEF'
100	'PDEF'
101	'PACK'
110	Reserved for future use
111	Reserved for future use

Bits 5 through 10 contain the resource ID of the owning resource (limited to 0 through 63). Bits 0 through 4 contain any desired value (0 through 31).

Some types of resources can't be owned because their IDs don't conform to this convention. For example, a resource of type 'WDEF' can own other resources but cannot itself be owned, because its resource ID can't be more than 12 bits long (see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*). The chapters describing individual resources provide detailed information about such restrictions.

An owned resource may itself contain the ID of a resource associated with it. For example, a dialog ('DLOG') resource owned by a desk accessory contains the resource ID of its item list. Although the item list is associated with the dialog resource, it's actually owned (indirectly) by the desk accessory. The resource ID of the item list should conform to the same special convention as the ID of the dialog resource. For example, if the resource ID of the desk accessory is 17, the IDs of both the dialog resource and the item list should contain the value 17 in bits 5 through 10.

A program that copies resources may need to change the resource ID of a resource so as not to duplicate an existing resource ID. Bits 5 through 10 of resources owned, directly or indirectly, by the copied resource should also be changed when those resources are copied. In the example just discussed, if the desk accessory must be given a new ID, bits 5 through 10 of both the dialog resource and the item list resource should also change.

S WARNING

When a resource-copying program changes the ID of an owned resource, it should also change the ID where it appears in other resources (such as an item list's ID contained in a dialog box resource). s

Resource Names

You can use a resource name instead of a resource ID to identify a resource of a given type. Like a resource ID, a resource name should be unique within each type. If you assign the same resource name to two resources of the same type, the second assignment of the name overrides the first, thereby making the first resource inaccessible by name. When comparing resource names, the Resource Manager ignores case (but does not ignore diacritical marks).

Resource Manager Routines

This section describes the routines provided by the Resource Manager. You can use these routines to create, open, and close resource forks; get and set the current resource file; read resources into memory; get and set resource information; modify resources; write to resource forks on disk; get a unique resource ID; count and list resource types; get resource sizes; dispose of resources; read and write partial resources; get and set resource fork attributes; and access resource entries in the resource map.

The `FSpCreateResFile` procedure and the `FSpOpenResFile` function use a file system specification (`FSSpec`) record. These routines are available only in System 7 or later. Use the `Gestalt` function to determine if these routines are available. If they're not available, you can call the equivalent File Manager HFS routines, the `HCreateResFile` procedure and the `HOpenResFile` function.

The Resource Manager provides a means for reporting errors specifically related to resources. After calling a Resource Manager routine, you can call the `ResError` function to determine whether any error occurred. The `ResError` function returns an integer value identifying any error reported by the Resource Manager routine that was executed last. The values listed in the `ResError` description signify only those errors dealing specifically with resources. The `ResError` function can also return values corresponding to Operating System result codes. The description for each Resource Manager routine includes the errors `ResError` may report for that routine under the subheading "Result Codes"; this list includes both the integer result codes for the Resource Manager routine as well as common Operating System result codes.

Initializing the Resource Manager

Unlike other Toolbox managers, the Resource Manager does not need to be explicitly initialized. System software automatically calls the Resource Manager's two initialization routines, the `InitResources` function and the `RsrcZoneInit` procedure—the former when the system starts up, and the latter when the system starts up and when the Process Manager starts up. You should not call either of these routines directly.

InitResources

When the system starts up, it automatically calls the `InitResources` function. This routine is for system use only, and your application should not call it at any time.

```
FUNCTION InitResources: Integer;
```

DESCRIPTION

The `InitResources` function initializes the Resource Manager. `InitResources` creates a special heap zone within the system heap and builds a resource map that points to ROM-resident resources. It opens the resource fork of the System file and reads its resource map into memory. The `InitResources` function returns an integer, which is the file reference number for the System file's resource fork.

Your application does not need to know the file reference number for the System file's resource fork, because every Resource Manager routine with a file reference number parameter also accepts 0 to mean the System file's resource fork.

ASSEMBLY-LANGUAGE INFORMATION

The `InitResources` function sets up three global variables: `SysResName`, `SysMap`, and `SysMapHndl`. These contain, respectively, the name of the System file's resource fork, the file reference number for the resource fork, and a handle to the System file's resource map.

RsrcZoneInit

System software automatically calls the `RsrcZoneInit` procedure when system software starts up and when the Process Manager starts up. Your application should not call this routine directly.

```
PROCEDURE RsrcZoneInit;
```

DESCRIPTION

System software automatically calls the `RsrcZoneInit` procedure at system startup when extensions are loaded, because each extension has its own application heap. System software calls `RsrcZoneInit` once again when the Process Manager starts up. After that, the procedure is not called again.

Checking for Errors

You can use the `ResError` function in your application to retrieve errors that may result from calling Resource Manager routines. You also can use `ResError` to check for an error after application startup (system software opens the resource fork of your application during application startup).

ResError

After calling a Resource Manager routine, you can use the `ResError` function to determine whether an error occurred and, if so, what it was.

```
FUNCTION ResError: Integer;
```

DESCRIPTION

The `ResError` function reads the value stored in the system global variable `ResErr` and returns an integer result code identifying errors, if any, that occurred. If no error occurred, `ResError` returns `noErr`. If an error occurs at the Resource Manager level, `ResError` returns one of the integer result codes listed in this section. If an error occurs at the Operating System level, `ResError` returns an Operating System result code, such as the Memory Manager error `memFullErr` or the File Manager error `ioErr`.

Resource Manager procedures do not report error information directly. Instead, after calling a Resource Manager procedure, your application should call the `ResError` function to determine whether an error occurred.

Resource Manager functions usually return `NIL` or `-1` as the function result when there's an error. For Resource Manager functions that return `-1`, your application can call the `ResError` function to determine the specific error that occurred. For Resource Manager functions that return handles, your application should always check whether the value of the returned handle is `NIL`. If it is, your application can use `ResError` to obtain specific information about the nature of the error. Note, however, that in some cases `ResError` returns `noErr` even though the value of the returned handle is `NIL`.

IMPORTANT

In certain cases, the `ResError` function returns `noErr` even though a Resource Manager routine was unable to perform the requested operation. See the individual routine descriptions for details about the circumstances under which this happens. [s](#)

Only those result codes dealing specifically with resources are listed in this section. See the description of each Resource Manager routine for a list of errors specific to that routine and that the `ResError` function returns.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ResErr` stores the current value of `ResError`, that is, the result code of the most recently performed Resource Manager operation. In addition, you can specify an application-defined procedure to be called whenever an error occurs. To do this, store the address of the procedure in the global variable `ResErrProc`. The value of the `ResErrProc` global variable is usually 0. Before returning a result code other than `noErr`, the `ResError` function puts that result in register D0 and calls the procedure identified by the `ResErrProc` global variable.

If you use `ResErrProc` to detect resource errors, you will get unexpected calls to your application-defined procedure if you call `GetMenu`. The Menu Manager routine `GetMenu` makes a call to `GetResInfo`, requesting resource information about 'MDEF' 0. Unfortunately, because `ROMMapInsert` is set to `FALSE`, this call fails, setting `ResErr` to -192 (`resNotFound`). This, in turn, causes a call to your application-defined procedure, even though the `GetMenu` routine has worked correctly.

To avoid this problem, follow these steps when you call `GetMenu` if you are using `ResErrProc`:

1. Save the address of your application-defined procedure.
2. Clear `ResErrProc`.
3. Call `GetResource` for the menu resource you want to get.
4. Check whether `GetResource` returns a handle whose value is `NIL`; if so, process the error in whatever way is appropriate for your application.
5. Call `GetMenu`.
6. When you are finished calling `GetMenu`, restore the previous value of `ResErrProc`.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resFNotFound</code>	-193	Resource file not found
<code>addResFailed</code>	-194	AddResource procedure failed
<code>rmvResFailed</code>	-196	RemoveResource procedure failed
<code>resAttrErr</code>	-198	Attribute inconsistent with operation
<code>mapReadErr</code>	-199	Map inconsistent with operation

Creating an Empty Resource Fork

You can use `FSpCreateResFile`, `HCreateResFile`, or `CreateResFile` when you want to create an empty resource fork—that is, a resource fork that contains no resource data but does include a resource map. Note that creating a resource fork does not automatically open it. To open a resource fork of a file created with one of these routines, use the corresponding routines `FSpOpenResFile`, `HOpenResFile`, or `OpenResFile`.

The `FSpCreateResFile` procedure is available only in System 7 and later versions of system software. If `FSpCreateResFile` is not available, you can use `HCreateResFile` or `CreateResFile` to create a resource fork. The `HCreateResFile` procedure allows you to specify a directory ID and a volume reference number, and is therefore preferred over `CreateResFile`. The `CreateResFile` procedure is an earlier version of `HCreateResFile` that is still supported but has more restricted capabilities.

Don't use the resource fork of a file for data that is not in resource format. The Resource Manager assumes that any information in a resource fork can be interpreted according to the standard resource format described in this chapter.

The File Manager assumes that the first block of a file's resource fork is part of the resource header and puts information there that it uses during scavenging—for example, after the user presses the Reset switch. For this reason, if you copy a resource file, the duplicate may not be exactly like the original.

FSpCreateResFile

You can use the `FSpCreateResFile` procedure to create an empty resource fork using a file system specification (`FSSpec`) record.

```
PROCEDURE FSpCreateResFile (spec: FSSpec;
                           creator, fileType: OSType;
                           scriptTag: ScriptCode);
```

<code>spec</code>	A file system specification record that indicates the name and location of the file whose resource fork is to be created.
<code>creator</code>	The signature of the application creating the file.
<code>fileType</code>	The file type of the new file.
<code>scriptTag</code>	The script code of the script system in which the Finder and standard file dialog boxes display the file's name.

DESCRIPTION

The `FSpCreateResFile` procedure creates an empty resource fork for a file with the specified type, creator, and script code in the location and with the name designated by the `spec` parameter. (An empty resource fork contains no resource data but does include a resource map.)

This procedure is available only in System 7 and later versions of system software. If `FSpCreateResFile` is not available to your application, you can use `HCreateResFile` or `CreateResFile`.

The `spec` parameter is a file system specification record, which is the standard format in System 7 and later versions for identifying a file or directory. The file system specification record for files and directories is defined by the `FSSpec` data type.

```

TYPE FSSpec = {file system specification}
    RECORD
        vRefNum: Integer;    {volume reference number}
        parID:   LongInt;    {directory ID of parent directory}
        name:    Str63;      {filename or directory name}
    END;

```

Certain File Manager routines—those that open a file’s data fork—also take a file system specification record as a parameter. You can use the same `FSSpec` record in Resource Manager routines that create or open the file’s resource fork.

The `creator` parameter of `FSpCreateResFile` contains the signature of the application that creates the file. Whenever your application creates a document, it assigns a creator and a file type to that document. Typically your application sets its signature as the document’s creator.

The `fileType` parameter indicates the type of file. You can set the file type to a type especially defined for your application or one of the existing general types, such as `'TEXT'` for text (a stream of ASCII characters), or `'pref'` for a preferences file.

Note

The file type should be as descriptive of the file’s data format as possible. You should not use `'TEXT'` as a file type unless the document contains plain ASCII characters. u

The value of the `scriptTag` parameter is the script code of the script system in which the Finder and the Standard File Package dialog boxes display the name of the file. For example, to specify the Roman script system, specify the constant `smRoman` in the `scriptTag` parameter.

If the file specified by the file system specification record doesn’t already exist (that is, if it has neither a data fork nor a resource fork), the `FSpCreateResFile` procedure creates a resource file—that is, a resource fork, including a resource map. In this case the file has a zero-length data fork. The `FSpCreateResFile` procedure also sets the creator, type, and script code fields of the file’s catalog information record to the specified values.

Resource Manager

If the file specified by the file system specification record already exists and includes a resource fork with a resource map, `FSpCreateResFile` does nothing. If the data fork of the file specified by the file system specification record already exists but the file has a zero-length resource fork, `FSpCreateResFile` creates an empty resource fork and resource map for the file; it also changes the creator, type, and script code fields of the catalog information record of the file to the specified values.

If your application uses Standard File Package routines, note that the `StandardPutFile` procedure returns a standard file reply record that contains a file system specification record in the `sfFile` field.

Before you can work with the newly created file's resource fork, you must use the `FSpOpenResFile` function to open it.

SPECIAL CONSIDERATIONS

The `FSpCreateResFile` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>dirFulErr</code>	-33	Directory full
<code>dskFulErr</code>	-34	Disk full
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name (perhaps zero length)
<code>tmfoErr</code>	-42	Too many files open
<code>wPrErr</code>	-44	Disk is write-protected
<code>fLckdErr</code>	-45	File is locked

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51. For information about using the `Gestalt` function to determine whether the `FSpCreateResFile` procedure is available, see "Using the Resource Manager," beginning on page 1-13. For a discussion of the use of the `FSpCreateResFile` procedure, see "Creating and Opening a Resource Fork" beginning on page 1-25. For a description of the `FSpOpenResFile` function, see page 1-58. For information about the `StandardPutFile` procedure and standard file reply records, see *Inside Macintosh: Files*. For more information on creators and file types, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `FSpCreateResFile` procedure are

Trap macro	Selector
<code>_HighLevelFSDispatch</code>	<code>\$000E</code>

HCreateResFile

If the `FSpCreateResFile` procedure is not available, you can use the `HCreateResFile` procedure to create an empty resource fork.

```
PROCEDURE HCreateResFile (vRefNum: Integer; dirID: LongInt;
                          fileName: Str255);
```

`vRefNum` The volume reference number of the volume on which the file is located.
`dirID` The directory ID of the directory where the file is located.
`fileName` The name of the file whose resource fork is to be created.

DESCRIPTION

The `HCreateResFile` procedure creates a file with an empty resource fork in the directory specified by the `vRefNum` and `dirID` parameters. (An empty resource fork contains no resource data but does include a resource map.)

If no other file with the given name exists in the specified directory, `HCreateResFile` creates a resource file—that is, a resource fork, including a resource map. In this case the file has a zero-length data fork.

If a file with the specified name already exists and includes a resource fork with a resource map, `HCreateResFile` does nothing. If the data fork of the specified file already exists but the file has a zero-length resource fork, `HCreateResFile` creates an empty resource fork and resource map for the file.

Before you can work with the newly created file's resource fork, you must first use `HOpenResFile` or a related function to open it.

SPECIAL CONSIDERATIONS

The `HCreateResFile` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>dirFulErr</code>	-33	Directory full
<code>dskFulErr</code>	-34	Disk full
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name (perhaps zero length)
<code>tmfoErr</code>	-42	Too many files open
<code>wPrErr</code>	-44	Disk is write-protected
<code>fLckdErr</code>	-45	File is locked

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `HOpenResFile` function, see page 1-62.

CreateResFile

If the `FSpCreateResFile` procedure is not available, you can use the `CreateResFile` procedure to create an empty resource fork.

```
PROCEDURE CreateResFile (fileName: Str255);
```

`fileName` **The name of the file to be created.**

DESCRIPTION

The `CreateResFile` procedure creates a file with an empty resource fork in your application's default directory—that is, the directory in which your application is located.

If no other file with the given name exists in the default directory or any of the other directories searched by `PBOpenRF` (see the following section, “Special Considerations”), `CreateResFile` creates a resource file—that is, a resource fork, including a resource map. In this case the file has a zero-length data fork.

If a file with the specified name already exists and includes a resource fork with a resource map, `CreateResFile` does nothing. Call `ResError` to determine whether an error occurred. If the data fork of the specified file already exists but the file has a zero-length resource fork, `CreateResFile` creates an empty resource fork and resource map for the file.

Before you can work with the newly created file's resource fork, you must use `OpenResFile` or a related function to open it.

SPECIAL CONSIDERATIONS

The `CreateResFile` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

The `CreateResFile` procedure first checks whether a file with the specified name exists. (If so, `ResError` returns the result code `dupFNErr`.) To perform this check, `CreateResFile` calls `PBOpenRF`, which looks first in the default directory for a file with the same name, then in the root directory of the boot volume (if the default directory is on the boot volume), and then in the System Folder (if one exists on the same volume as the default directory). It is thus impossible, for example, to use `CreateResFile` to create a file in the default directory if a file with the same name already exists in the System Folder. To avoid this problem, use `FSpCreateResFile` or `HCreateResFile` whenever possible.

RESULT CODES

noErr	0	No error
dirFulErr	-33	Directory full
dskFulErr	-34	Disk full
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
tmfoErr	-42	Too many files open
wPrErr	-44	Disk is write-protected
fLckdErr	-45	File is locked
dupFNERR	-48	Another file with the same name exists in the default directory, the root directory of the boot volume, or the System Folder

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `OpenResFile` function, see page 1-66.

Opening Resource Forks

To open a resource fork, the Resource Manager calls the appropriate File Manager routine and returns the file reference number that it gets from the File Manager. If the file reference number returned is greater than 0, you can use this number to refer to the resource fork in some other Resource Manager routines.

The `FSpOpenResFile`, `HOpenResFile`, `OpenRFPPerm`, and `OpenResFile` functions all open resource forks. Use the `FSpOpenResFile` function to open a resource fork using a file system specification (`FSSpec`) record. You can determine whether `FSpOpenResFile` is available by calling the `Gestalt` function with the `gestaltFSAttr` selector code.

If `FSpOpenResFile` is not available, you can use `HOpenResFile`, `OpenRFPPerm`, or `OpenResFile` to open a resource fork. The `HOpenResFile` function allows you to specify both a directory ID and a volume reference number, and is therefore preferred if `FSpOpenResFile` is not available. The `OpenRFPPerm` and `OpenResFile` functions are earlier versions of `HOpenResFile` that are still supported but are more restricted in their capabilities.

FSpOpenResFile

You can use the `FSpOpenResFile` function to open a file's resource fork using a file system specification (`FSSpec`) record.

```
FUNCTION FSpOpenResFile (spec: FSSpec;
                        permission: SignedByte): Integer;
```

Resource Manager

<code>spec</code>	A file system specification record specifying the name and location of the file whose resource fork is to be opened.
<code>permission</code>	A value that specifies a read/write permission combination.

DESCRIPTION

The `FSpOpenResFile` function opens the resource fork of the file identified by the `spec` parameter. It also makes this file the current resource file.

This function is available only in System 7 and later versions of system software. If `FSpOpenResFile` is not available to your application, you can use `HOpenResFile`, `OpenRFParm`, or `OpenResFile` instead.

The `spec` parameter is a file system specification record, which is a standard format in System 7 and later versions for identifying a file or directory. The file system specification record for files and directories is defined by the `FSSpec` data type.

```

TYPE FSSpec = {file system specification}
    RECORD
        vRefNum: Integer;    {volume reference number}
        parID:   LongInt;    {directory ID of parent directory}
        name:    Str63;      {filename or directory name}
    END;

```

You can specify the access path permission for the resource fork by setting the `permission` parameter to one of these constants:

```

CONST
    fsCurPerm   = 0; {whatever is currently allowed}
    fsRdPerm    = 1; {read-only permission}
    fsWrPerm    = 2; {write permission}
    fsRdWrPerm  = 3; {exclusive read/write permission}
    fsRdWrShPerm= 4; {shared read/write permission}

```

Use `fsCurPerm` to request whatever permission is currently allowed. If write access is unavailable (because the file is locked or because the resource fork is already open with write access), then read permission is granted. Otherwise, read/write permission is granted.

Use `fsRdPerm` to request permission to read the file, and `fsWrPerm` to write to it. If write permission is granted, no other access paths are granted write permission. Because the File Manager doesn't support write-only access to a file, `fsWrPerm` is synonymous with `fsRdWrShPerm`.

Use `fsRdWrPerm` and `fsRdWrShPerm` to request exclusive or shared read/write permission, respectively. If your application is granted exclusive read/write permission, no users are granted permission to write to the file; other users may, however, be granted

permission to read the file. Shared read/write permission allows multiple access paths for writing and reading.

The Resource Manager reads the resource map from the specified file's resource fork into memory. It also reads into memory every resource in the resource fork whose `resPreload` attribute is set.

The `FSpOpenResFile` function returns a file reference number for the resource fork. You can use this reference number to refer to the resource fork in other Resource Manager routines.

If you attempt to use `FSpOpenResFile` to open a resource fork that is already open, `FSpOpenResFile` returns the existing file reference number or a new one, depending on the access permission for the existing access path. For example, your application receives a new file reference number after a successful request for read-only access to a file previously opened with write access, whereas it receives the same file reference number in response to a second request for write access to the same file. In this case, `FSpOpenResFile` doesn't make that file the current resource file.

If the `FSpOpenResFile` function fails to open the specified file's resource fork (for instance, because there's no file with the given file system specification record or because there are permission problems), it returns `-1` as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `FSpOpenResFile` to open the System file's resource fork or an application file's resource fork. These resource forks are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after your application starts up and before you open any other resource forks.

The `FSpOpenResFile` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

SPECIAL CONSIDERATIONS

The `FSpOpenResFile` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

It's possible to create multiple, unique, read-only access paths to a resource fork using `FSpOpenResFile`; however, you should avoid doing so. If a resource fork is opened twice—once with read/write permission and once with read-only permission—two copies of the resource map exist in memory. If you change one of the resources in memory using one of the resource maps, the two resource maps become inconsistent and the file will appear damaged to the second resource map.

If you must use this technique for read-only access, call `FSpOpenResFile` immediately before your application reads information from the file and close the file immediately afterward. Otherwise, your application may get unexpected results.

Resource Manager

If an application attempts to open a second access path with write access and the application is different from the one that originally opened the resource fork, `FSpOpenResFile` returns `-1`, and the `ResError` function returns the result code `opWrErr`.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `FSpOpenResFile` with calls to `SetResLoad` with the load parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. If you don't do this, the Segment Loader Manager treats any preloaded 'CODE' resources as your code resources when you make an intersegment call that triggers a call to `LoadSeg` while the other application is first in the resource chain. In this case, your application can begin executing the other application's code, and severe problems will ensue. If you need to get 'CODE' resources from the other application's resource fork, you can still prevent the Segment Loader Manager problem by calling `UseResFile` with your application's file reference number to make your application the current resource file.

ASSEMBLY-LANGUAGE INFORMATION

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`. The trap macro and routine selector for the `FSpOpenResFile` are

Trap macro	Selector
<code>_HighLevelFSDispatch</code>	<code>\$0000</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>nsvErr</code>	<code>-35</code>	No such volume
<code>ioErr</code>	<code>-36</code>	I/O error
<code>bdNamErr</code>	<code>-37</code>	Bad filename or volume name (perhaps zero length)
<code>eofErr</code>	<code>-39</code>	End of file
<code>tmfoErr</code>	<code>-42</code>	Too many files open
<code>fnfErr</code>	<code>-43</code>	File not found
<code>opWrErr</code>	<code>-49</code>	File already open with write permission
<code>permErr</code>	<code>-54</code>	Permissions error (on file open)
<code>extFSErr</code>	<code>-58</code>	Volume belongs to an external file system
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone
<code>dirNFErr</code>	<code>-120</code>	Directory not found
<code>mapReadErr</code>	<code>-199</code>	Map inconsistent with operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51. For information about using the `Gestalt` function to determine whether the `FSpOpenResFile` procedure is available, see "Using the Resource Manager" beginning on page 1-13. For an example of the use of `FSpOpenResFile` to open a resource fork, see Listing 1-7 on page 1-27.

For information about the `CurResFile` and `UseResFile` routines, see page 1-68 and page 1-69, respectively.

For more information about permission parameter constants or the `OpenRF` function, see *Inside Macintosh: Files*.

HOpenResFile

If the `FSpOpenResFile` function is not available, you can use `HOpenResFile` to open a file's resource fork.

```
FUNCTION HOpenResFile (vRefNum: Integer; dirID: LongInt;
                      fileName: Str255;
                      permission: SignedByte): Integer;
```

`vRefNum` The volume reference number of the volume on which the file is located.

`dirID` The directory ID of the directory where the file is located.

`fileName` The name of the file whose resource fork is to be opened.

`permission` A constant for one of the read/write permission combinations.

DESCRIPTION

The `HOpenResFile` function opens the resource fork of the file with the name specified by the `fileName` parameter in the directory specified by the `vRefNum` and `dirID` parameters. It also makes this file the current resource file.

You can specify the access path permission for the resource fork by setting the `permission` parameter to one of these constants:

```
CONST
    fsCurPerm      = 0; {whatever is currently allowed}
    fsRdPerm        = 1; {read-only permission}
    fsWrPerm        = 2; {write permission}
    fsRdWrPerm     = 3; {exclusive read/write permission}
    fsRdWrShPerm   = 4; {shared read/write permission}
```

See page 1-59 for information about specifying access path permission with `FSpOpenResFile`. The same information applies to `HOpenResFile`.

The Resource Manager reads the resource map from the resource fork of the specified file into memory. It also reads into memory every resource whose `resPreload` attribute is set.

The `HOpenResFile` function returns a file reference number for the file. You can use this file reference number to refer to the file in other Resource Manager routines. If the file's resource fork is already open, `HOpenResFile` returns the file reference number but does not make that file the current resource file.

If the `HOpenResFile` function fails to open the specified file's resource fork (because there's no file with the specified name or because there are permission problems), it returns `-1` as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `HOpenResFile` to open the System file's resource fork or an application file's resource fork. These files are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after the application starts up and before you open the resource forks for any other files.

The `HOpenResFile` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

SPECIAL CONSIDERATIONS

The `HOpenResFile` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

It's possible to create multiple, unique, read-only access paths to a resource fork using `HOpenResFile`; however, you should avoid doing so. See page 1-60 for discussion of this issue in relation to `FSpOpenResFile`. The `HOpenResFile` function works the same way.

Versions of system software before System 7 do not allow you to use `HOpenResFile` to open a second access path, with write access, to a resource fork. In this case, `HOpenResFile` returns the reference number already assigned to the file.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `HOpenResFile` with calls to `SetResLoad` with the `load` parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. The discussion of this issue in relation to `FSpOpenResFile` (page 1-60) also applies to `HOpenResFile`.

ASSEMBLY-LANGUAGE INFORMATION

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`.

RESULT CODES

noErr	0	No error
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
eofErr	-39	End of file
tmfoErr	-42	Too many files open
fnfErr	-43	File not found
opWrErr	-49	File already open with write permission
permErr	-54	Attempt to open locked file for writing
extFSerr	-58	Volume belongs to an external file system
memFullErr	-108	Not enough room in heap zone
dirNFErr	-120	Directory not found
mapReadErr	-199	Map inconsistent with operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about permission parameter constants and the `OpenRF` function, see *Inside Macintosh: Files*.

OpenRFPPerm

If the `FSpOpenResFile` and `HOpenResFile` functions are not available, you can use the `OpenRFPPerm` function to open a file's resource fork.

```
FUNCTION OpenRFPPerm (fileName: Str255; vRefNum: Integer;
                    permission: SignedByte): Integer;
```

<code>fileName</code>	The name of the file whose resource fork is to be opened.
<code>vRefNum</code>	The volume reference number or directory ID for the volume or directory in which the file is located.
<code>permission</code>	A constant for one of the read/write permission combinations.

DESCRIPTION

The `OpenRFPPerm` function opens the resource fork of the file with the name specified by the `fileName` parameter in the directory or volume specified by the `vRefNum` parameter. It also makes this file the current resource file.

In addition to opening the resource fork for the file with the specified name, `OpenRFPPerm` lets you specify in the `permission` parameter the read/write permission of the resource fork the first time it is opened.

You can use the `OpenRFPPerm` function if the `FSpOpenResFile` function is not available. You can determine whether `FSpOpenResFile` is available by calling the `Gestalt` function with the `gestaltFSAttr` selector code. The `OpenRFPPerm` is an earlier version of the `HOpenResFile` function.

You can specify the access path permission for the resource fork by setting the `permission` parameter to one of these constants:

```
CONST
    fsCurPerm      = 0; {whatever is currently allowed}
    fsRdPerm       = 1; {read-only permission}
    fsWrPerm       = 2; {write permission}
    fsRdWrPerm     = 3; {exclusive read/write permission}
    fsRdWrShPerm  = 4; {shared read/write permission}
```

See page 1-59 for information about specifying access path permission with `FSpOpenResFile`. The same information applies to `OpenRFPPerm`.

The Resource Manager reads the resource map from the resource fork for the specified file into memory. It also reads into memory every resource in the resource fork whose `resPreload` attribute is set.

The `OpenRFPPerm` function returns a file reference number for the file whose resource fork it has opened. You can use this file reference number to refer to the file in other Resource Manager routines. If the file's resource fork is already open, `OpenRFPPerm` returns the file reference number but does not make that file the current resource file.

If the `OpenRFPPerm` function fails to open the specified file's resource fork (because there's no file with the given name or because there are permission problems), it returns -1 as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `OpenRFPPerm` to open the System file's resource fork or an application file's resource fork. These files are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after the application starts up and before you open the resource forks for any other files.

The `OpenRFPPerm` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

SPECIAL CONSIDERATIONS

The `OpenRFPPerm` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

It's possible to create multiple, unique, read-only access paths to a resource fork using `OpenRFPPerm`; however, you should avoid doing so. See page 1-60 for discussion of this issue in relation to `FSpOpenResFile`; `OpenRFPPerm` works the same way.

Versions of system software before System 7 do not allow you to use `OpenRFPPerm` to open a second access path, with write access, to a resource fork. In this case, `OpenRFPPerm` returns the reference number already assigned to the file.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `OpenRFPPerm` with calls to `SetResLoad` with the `load` parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. The discussion of this issue in relation to `FSpOpenResFile` (page 1-60) also applies to `OpenRFPPerm`.

ASSEMBLY-LANGUAGE INFORMATION

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`.

RESULT CODES

<code>noErr</code>	0	No error
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name (perhaps zero length)
<code>eofErr</code>	-39	End of file
<code>tmfoErr</code>	-42	Too many files open
<code>fnfErr</code>	-43	File not found
<code>opWrErr</code>	-49	File already open with write permission
<code>permErr</code>	-54	Attempt to open locked file for writing
<code>extFSErr</code>	-58	Volume belongs to an external file system
<code>memFullErr</code>	-108	Not enough room in heap zone
<code>dirNFErr</code>	-120	Directory not found
<code>mapReadErr</code>	-199	Map inconsistent with operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about permission parameter constants and the `OpenRF` function, see *Inside Macintosh: Files*.

OpenResFile

If the `FSpOpenResFile` function is not available, you can use the `OpenResFile` function to open a resource fork.

```
FUNCTION OpenResFile (fileName: Str255): Integer;
```

`fileName` The name of the file whose resource fork is to be opened.

DESCRIPTION

The `OpenResFile` function opens the resource fork of the file with the name specified by the `fileName` parameter in the application's default directory—that is, the directory in which the application is located. It also makes this file the current resource file.

Like the `OpenRFPPerm` function, the `OpenResFile` function takes a filename and opens the resource fork for the file with that name. Unlike `OpenRFPPerm`, `OpenResFile` does not let you specify the read/write permission of the resource fork the first time it is opened. The `OpenResFile` function is an earlier version of the `OpenRFPPerm` function.

If it finds the specified file in your application's default directory, `OpenResFile` reads the file's resource map into memory and returns a file reference number for the file. It also reads into memory every resource in the resource fork whose `resPreload` attribute is set.

You can use the file reference number returned by `OpenResFile` to refer to the file in other Resource Manager routines. If the file's resource fork is already open, `OpenResFile` returns the file reference number but does not make that file the current resource file.

If the `OpenResFile` function fails to open the specified file's resource fork (for instance, because there's no file with the given name), it returns `-1` as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `OpenResFile` to open the System file's resource fork or an application file's resource fork. These resource forks are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after the application starts up and before you open the resource forks for any other files.

The `OpenResFile` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

SPECIAL CONSIDERATIONS

The `OpenResFile` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `OpenResFile` with calls to `SetResLoad` with the `load` parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. The discussion of this issue in relation to `FSpOpenResFile` (page 1-60) also applies to `OpenResFile`.

ASSEMBLY-LANGUAGE INFORMATION

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`.

RESULT CODES

noErr	0	No error
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
eofErr	-39	End of file
tmfoErr	-42	Too many files open
fnfErr	-43	File not found
opWrErr	-49	File already open with write permission
permErr	-54	Attempt to open locked file for writing
extFSerr	-58	Volume belongs to an external file system
memFullErr	-108	Not enough room in heap zone
dirNFErr	-120	Directory not found
mapReadErr	-199	Map inconsistent with operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

Getting and Setting the Current Resource File

Most of the Resource Manager routines assume that the current resource file is the file on whose resource fork they should operate or, in the case of a search, the file where they should begin. In general, the current resource file is the last one whose resource fork your application opened unless you specify otherwise.

Two routines work specifically with the current resource file: `CurResFile` and `UseResFile`. The `CurResFile` function tells you which of the files whose resource forks are currently open is the current resource file. The `UseResFile` procedure sets the current resource file.

The `HomeResFile` function gets the file reference number associated with a particular resource.

CurResFile

You can use the `CurResFile` function to get the file reference number of the current resource file.

```
FUNCTION CurResFile: Integer;
```

DESCRIPTION

The `CurResFile` function returns the file reference number associated with the current resource file. You can call this function when your application starts up (before opening the resource fork of any other file) to get the file reference number of your application's resource fork.

If the current resource file is the System file, `CurResFile` returns the actual file reference number. You can use this number or 0 with routines that take a file reference number for the System file. All Resource Manager routines recognize both 0 and the actual file reference number as referring to the System file.

ASSEMBLY-LANGUAGE INFORMATION

The current resource file's reference number is stored in the global variable `CurMap`.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `CurResFile` function, see Listing 1-8 on page 1-29.

UseResFile

You can use the `UseResFile` procedure to set the current resource file.

```
PROCEDURE UseResFile (refNum: Integer);
```

`refNum` The file reference number for a resource fork.

DESCRIPTION

The `UseResFile` procedure searches the list of files whose resource forks have been opened for the file specified by the `refNum` parameter. If the specified file is found, the Resource Manager sets the current resource file to the specified file. If there's no resource fork open for a file with that reference number, `UseResFile` does nothing. To set the current resource file to the System file, use 0 for the `refNum` parameter.

Open resource forks are arranged as a linked list with the most recently opened resource fork at the beginning. When searching open resource forks, the Resource Manager starts with the most recently opened file. You can call the `UseResFile` procedure to set the current resource file to a file opened earlier, and thereby start subsequent searches with the specified file. In this way, you can cause any files higher in the resource chain to be left out of subsequent searches.

When a new resource fork is opened, this action overrides previous calls to `UseResFile` and the entire list is searched. For example, if five resource forks are opened in the order R0, R1, R2, R3, and R4, the search order is R4-R3-R2-R1-R0. Calling `UseResFile(R2)` changes the search order to R2-R1-R0; R4 and R3 are not searched. When the resource fork of a new file (R5) is opened, the search order becomes R5-R4-R3-R2-R1-R0.

You typically call `CurResFile` to get and save the current resource file, `UseResFile` to set the current resource file to the desired file, then (after you are finished using the resource) `UseResFile` to restore the current resource file to its previous value. Calling `UseResFile(0)` causes the Resource Manager to search only the System file's resource map. This is useful if you no longer wish to override a system resource with one by the same name in your application's resource fork.

SPECIAL CONSIDERATIONS

The `FSpOpenResFile`, `HOpenResFile`, and `OpenResFile` functions, which also set the current resource file, override previous calls to `UseResFile`.

ASSEMBLY-LANGUAGE INFORMATION

The settings of the system global variables `RomMapInsert` and `TmpResLoad` affect resource search order. These global variables determine whether the Resource Manager searches ROM-resident resources before the System file's resources.

The Resource Manager normally searches ROM resources only when you use the `RGetResource` function to get a handle to the resource, and even then only after it searches the System file's resource fork. To search for a resource in ROM before searching the System file's resource fork, your application must first alter the resource search order by inserting the ROM resource map in front of the System file's resource map.

When the value of the system global variable `RomMapInsert` is `TRUE`, the Resource Manager inserts the ROM resource map before the System file's resource map for the next call only (including any Resource Manager routine that gets a resource, not just `RGetResource`). When the value of `RomMapInsert` is `TRUE`, the adjacent variable `TmpResLoad` determines whether the value of the global variable `ResLoad` is considered `TRUE` or `FALSE`, overriding the actual value of `ResLoad` for the next call only. The values of the `RomMapInsert` and `TmpResLoad` variables are cleared after each call to a Resource Manager routine.

You can use two global constants to set these variables in tandem. Set the system global variable `RomMapInsert` to the global constant `mapTrue` to insert the ROM resource map with `SetResLoad(TRUE)`. Set the system global variable `RomMapInsert` to the global constant `mapFalse` to insert the ROM resource map with `SetResLoad(FALSE)`.

RESULT CODES

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `UseResFile` procedure, see Listing 1-8 on page 1-29.

For descriptions of the `FSpOpenResFile`, `HOpenResFile`, and `OpenResFile` functions, see page 1-58 through page 1-66. For a description of the `SetResLoad` procedure, see page 1-79.

HomeResFile

To get the file reference number associated with a particular resource, use the `HomeResFile` function.

```
FUNCTION HomeResFile (theResource: Handle): Integer;
```

```
theResource
```

A handle to a resource.

DESCRIPTION

Given a handle to a resource, the `HomeResFile` function returns the file reference number for the resource fork containing the specified resource. If the given handle isn't a handle to a resource, `HomeResFile` returns `-1`, and the `ResError` function returns the result code `resNotFound`. If `HomeResFile` returns `0`, the resource is in the System file's resource fork. If `HomeResFile` returns `1`, the resource is ROM-resident.

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>resNotFound</code>	<code>-192</code>	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

Reading Resources Into Memory

The routines described in this section allow your application to read resource data into memory. The `GetResource` and `Get1Resource` functions get a resource specified by a resource type and a resource ID. The `GetNamedResource` and `Get1NamedResource` functions get a resource specified by name. The `RGetResource` function searches the ROM-resident resources as well as the open resource forks.

The `SetResLoad` procedure enables and disables automatic loading of resource data into memory for routines that return handles to resources, and the `LoadResource` procedure reads resource data into memory for a purged resource or after you've called `SetResLoad` with the `load` parameter set to `FALSE`.

When your application requests a resource, the Resource Manager normally looks in the current resource file's resource map in memory. If it can't find an entry for the specified resource, the Resource Manager searches the resource maps for each open resource fork in the reverse order that the resource forks were opened. If it can't find an entry for the specified resource in any of these resource maps, the Resource Manager searches your application's resource map. If it can't find an entry for the specified resource in your application's resource map, the Resource Manager searches the resource map for the System file.

The Resource Manager determines whether or not to load the specified resource into memory according to the entry for that resource in the resource map. If the resource's resource map entry contains a valid handle, the Resource Manager returns that handle. If the value of the handle is `NIL`, the Resource Manager reads the resource data into memory.

Before reading the resource data into memory, the Resource Manager calls the Memory Manager to allocate a relocatable block for the resource data. The Memory Manager allocates the block, assigns a master pointer to the block, and returns to the Resource Manager a pointer to the master pointer. The Resource Manager then installs this handle in the resource map and also returns a handle to the resource.

If the resource's resource map entry contains an empty handle (a handle whose master pointer is set to `NIL`) and the value of the system global variable `ResLoad` is `TRUE`, the Resource Manager routines that get resources reallocate the resource's handle and read the resource data from disk back into memory.

IMPORTANT

In certain situations, a Resource Manager routine can return an empty handle (a handle whose master pointer is set to `NIL`). For instance, if you've called `SetResLoad` with the `load` parameter set to `FALSE` and the resource data isn't already in memory, and then you call the `GetResource` function (or any of the other Resource Manager routines that get a resource), the Resource Manager routine returns an empty handle (a handle whose master pointer is set to `NIL`). This can also happen if you read resource data for a purgeable resource into memory and then call `SetResLoad` with the `load` parameter set to `FALSE`. If the resource data is later purged, when you call `GetResource` (or other routines that get a resource), the Resource Manager returns an empty handle. You should test for an empty handle in these situations. To make the handle a valid handle to resource data in memory, you can call the `LoadResource` procedure. u

GetResource

You can use the `GetResource` function to get resource data for a resource specified by resource type and resource ID.

```
FUNCTION GetResource (theType: ResType; theID: Integer): Handle;
```

`theType` A resource type.

`theID` An integer that uniquely identifies a resource of the specified type.

DESCRIPTION

The `GetResource` function searches the resource maps in memory for the resource specified by the parameters `theType` and `theID`. The resource maps in memory, which represent all the open resource forks, are arranged as a linked list. When searching this list, `GetResource` starts with the current resource file and progresses through the list (that is, searching the resource maps in reverse order of opening) until it finds the resource's entry in one of the resource maps.

If the `GetResource` function finds the specified resource entry in one of the resource maps and the entry contains a valid handle, it returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `GetResource` attempts to read the resource into memory.

If `GetResource` can't find the resource data, it returns `NIL`, and `ResError` returns the result code `resNotFound`. The `GetResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code. If you call `GetResource` with a resource type that can't be found in any of the resource maps of the open resource forks, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the resource map search order by calling the `UseResFile` procedure before `GetResource`.

SPECIAL CONSIDERATIONS

Calling `GetResource` may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of `GetResource`, see page 1-18 through page 1-24.

To include ROM-resident system resources in the Resource Manager's search of the resource maps of open resource forks, use the `RGetResource` function as described on page 1-78.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

Get1Resource

You can use the `Get1Resource` function to get resource data for a resource in the current resource file.

```
FUNCTION Get1Resource (theType: ResType; theID: Integer): Handle;
```

`theType` A resource type.

`theID` An integer that uniquely identifies a resource of the specified type.

DESCRIPTION

The `Get1Resource` function searches the current resource file's resource map in memory for the resource specified by the `theType` and `theID` parameters. If `Get1Resource` finds an entry for the resource in the current resource file's resource map and the entry contains a valid handle, it returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `Get1Resource` attempts to read the resource into memory.

If `Get1Resource` can't find the resource data, it returns `NIL`, and `ResError` returns the result code `resNotFound`. The `Get1Resource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code.

If you call `Get1Resource` with a resource type that can't be found in the resource map of the current resource file, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the resource map search order by calling the `UseResFile` procedure before `Get1Resource`.

SPECIAL CONSIDERATIONS

Calling `Get1Resource` may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of the `Get1Resource` function, see Listing 1-8 on page 1-29 and Listing 1-9 on page 1-32.

To include ROM-resident system resources in the Resource Manager's search of the resource maps for open resource forks, use the `RGetResource` function as described on page 1-78.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

GetNamedResource

You can use the `GetNamedResource` function to get a named resource.

```
FUNCTION GetNamedResource (theType: ResType; name: Str255)
                          : Handle;
```

`theType` A resource type.

`name` A name that uniquely identifies a resource of the specified type. Strings passed in this parameter are case-sensitive.

DESCRIPTION

The `GetNamedResource` function searches the resource maps in memory for the resource specified by the parameters `theType` and `name`. The resource maps in memory, which represent all the open resource forks, are arranged as a linked list. When `GetNamedResource` searches this list, it starts with the current resource file and progresses through the list in order (that is, in reverse chronological order in which the resource forks were opened) until it finds the resource's entry in one of the resource maps.

Resource Manager

If `GetNamedResource` finds the specified resource entry in one of the resource maps and the entry contains a valid handle, the function returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the load parameter set to `FALSE`, `GetNamedResource` attempts to read the resource into memory.

If the `GetNamedResource` function can't find the resource data, it returns `NIL`, and `ResError` returns the result code `resNotFound`. The `GetNamedResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code. If you call `GetNamedResource` with a resource type that can't be found in any of the resource maps of the open resource forks, the function returns `NIL` as well, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the resource map search order by calling the `UseResFile` procedure before `GetNamedResource`.

SPECIAL CONSIDERATIONS

The `GetNamedResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

To include ROM-resident system resources in the Resource Manager's search of the resource maps for open resource forks, use the `RGetResource` function as described on page 1-78.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

Get1NamedResource

You can use the `Get1NamedResource` function to get a named resource in the current resource file.

```
FUNCTION Get1NamedResource (theType: ResType; name: Str255)
                        : Handle;
```

`theType` A resource type.

`name` A name that uniquely identifies a resource of the specified type.

DESCRIPTION

The `Get1NamedResource` function searches the current resource file's resource map in memory for the resource specified by the parameters `theType` and `name`. If `Get1NamedResource` finds an entry for the resource in the current resource file's resource map and the entry contains a valid handle, the function returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `Get1NamedResource` attempts to read the resource into memory.

If it can't find the resource data, `Get1NamedResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. The `Get1NamedResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code.

If you call `Get1NamedResource` with a resource type that can't be found in the resource map of the current resource file, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the search order by calling the `UseResFile` procedure before `Get1NamedResource`.

SPECIAL CONSIDERATIONS

The `Get1NamedResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

To include ROM-resident system resources in the Resource Manager's search of the resource maps for open resource forks, use the `RGetResource` function, described next.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

RGetResource

You can use the `RGetResource` function to get resource data for a resource and include ROM-resident system resources in the Resource Manager's search of resource maps.

```
FUNCTION RGetResource (theType: ResType; theID: Integer): Handle;
```

`theType` A resource type.

`theID` An integer that uniquely identifies a resource of the specified type.

DESCRIPTION

The `RGetResource` function searches the resource maps in memory for the resource specified by the parameters `theType` and `theID`. The resource maps in memory, which represent all open resource forks, are arranged as a linked list. The `RGetResource` function first uses `GetResource` to search this list. The `GetResource` function starts with the current resource file and progresses through the list in order (that is, in reverse chronological order in which the resource forks were opened) until it finds the resource's entry in one of the resource maps. If `GetResource` doesn't find the specified resource in its search of the resource maps of open resource forks (which includes the System file's resource fork), `RGetResource` sets the global variable `RomMapInsert` to `TRUE`, then calls `GetResource` again. In response, `GetResource` performs the same search, but this time it looks in the resource map of the ROM-resident resources before searching the resource map of the System file.

If `RGetResource` finds the specified resource entry in one of the resource maps and the entry contains a valid handle, the function returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `RGetResource` attempts to read the resource into memory.

If it can't find the resource data, `RGetResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. The `RGetResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code. If you call `RGetResource` with a resource type that can't be found in any of the resource maps of the open resource forks, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

SPECIAL CONSIDERATIONS

The `RGetResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information, see “Inserting the ROM Resource Map” beginning on page 1-134.

For a description of the `UseResFile` procedure, see page 1-69. The `SetResLoad` procedure is described next.

SetResLoad

You can use the `SetResLoad` procedure to enable and disable automatic loading of resource data into memory for routines that return handles to resources.

```
PROCEDURE SetResLoad (load: Boolean);
```

<code>load</code>	A Boolean value that determines whether Resource Manager routines should read resource data into memory. If you set this parameter to <code>TRUE</code> , Resource Manager routines that return handles will, during subsequent calls, automatically read resource data into memory if it is not already in memory; if you set this parameter to <code>FALSE</code> , Resource Manager routines will not automatically read resource data into memory.
-------------------	--

DESCRIPTION

Routines that return handles to resources normally read the resource data into memory if it's not already there. The default setting (`load = TRUE`) maintains this state. If the `load` parameter is set to `FALSE`, routines that return handles to resources will not, during subsequent calls, load the resource data into memory. Instead, such routines return a handle whose master pointer is set to `NIL` unless the resource is already in memory. In addition, when first opening a resource fork the Resource Manager won't load into memory resources whose `resPreload` attribute is set.

You can use the `SetResLoad` procedure when you want to read from the resource map without reading the resource data into memory. To read the resource data into memory after a call to `SetResLoad`, call the `LoadResource` procedure, which is described next.

S WARNING

If you call `SetResLoad` with the `load` parameter set to `FALSE`, be sure to call `SetResLoad` with the `load` parameter set to `TRUE` as soon as possible. Other parts of system software that call the Resource Manager expect this value to be `TRUE`, and some routines won't work if resources are not loaded automatically. s

ASSEMBLY-LANGUAGE INFORMATION

The current value of `SetResLoad` is stored in the global variable `ResLoad`.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about the global variable `ResLoad`, see “Inserting the ROM Resource Map” beginning on page 1-134.

LoadResource

You can use the `LoadResource` procedure to get resource data after you've called `SetResLoad` with the `load` parameter set to `FALSE` or when the resource is purgeable.

```
PROCEDURE LoadResource (theResource: Handle);
```

`theResource`

A handle to a resource.

DESCRIPTION

Given a handle to a resource, `LoadResource` reads the resource data into memory. If the resource is already in memory, or if the `theResource` parameter doesn't contain a handle to a resource, then `LoadResource` does nothing. To determine whether either of these situations occurred, call `ResError`. If the resource is already in memory, `ResError` returns `noErr`; if the handle is not a handle to a resource, `ResError` returns `resNotFound`.

SPECIAL CONSIDERATIONS

If you've changed the resource data for a purgeable resource and the resource is purged before being written to the file, the changes will be lost. In this case, `LoadResource` rereads the original resource from the file's resource fork. You should use `ChangedResource` or `SetResPurge` before calling `LoadResource` to ensure that changes made to purgeable resources are written to the resource fork.

ASSEMBLY-LANGUAGE INFORMATION

The `LoadResource` procedure preserves all registers.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For information about the `SetResLoad`, `ChangedResource`, and `SetResPurge` procedures, see page 1-79, page 1-88, and page 1-94, respectively.

Getting and Setting Resource Information

The Resource Manager provides four routines that allow you to get and set information about resources. The `GetResInfo` procedure returns the resource ID, resource type, and resource name for a specified resource. The `SetResInfo` procedure sets the resource name and resource ID for a specified resource. The `GetResAttrs` function returns a resource's attributes, and the `SetResAttrs` function sets a resource's attributes.

GetResInfo

You can use the `GetResInfo` procedure to get a resource's resource ID, resource type, and resource name.

```
PROCEDURE GetResInfo (theResource: Handle; VAR theID: Integer;
                     VAR theType: ResType; VAR name: Str255);
```

`theResource`

A handle to a resource.

`theID`

`GetResInfo` returns the resource ID of the specified resource in this parameter.

Resource Manager

`theType` `GetResInfo` returns the resource type of the specified resource in this parameter.

`name` `GetResInfo` returns the name of the specified resource in this parameter.

DESCRIPTION

Given a handle to a resource, the `GetResInfo` procedure returns the resource's resource ID, resource type, and resource name. If the handle isn't a valid handle to a resource, `GetResInfo` does nothing; to determine whether this has occurred, call `ResError`.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

To set a resource's ID, resource type, or resource name, use the `SetResInfo` procedure. It is described next.

SetResInfo

You can use the `SetResInfo` procedure to change the name and resource ID of a resource.

```
PROCEDURE SetResInfo (theResource: Handle; theID: Integer;
                     name: Str255);
```

`theResource` A handle to a resource.

`theID` The new resource ID.

`name` The new name or an empty string to preserve the resource name.

DESCRIPTION

Given a handle to a resource, `SetResInfo` changes the resource ID and the resource name of the specified resource to the values given in `theID` and `name`. If you pass an empty string for the `name` parameter, the resource name is not changed. The `SetResInfo` procedure changes the information in the resource map in memory, not in the resource file itself.

S WARNING

Do not change a system resource's resource ID or name. Other applications may already access the resource and may not work properly if you change the resource ID, resource name, or both. *s*

If the parameter `theResource` doesn't contain a handle to an existing resource, `SetResInfo` does nothing, and `ResError` returns the result code `resNotFound`. If the resource map becomes too large to fit in memory (for example, after an unnamed resource is given a name), `SetResInfo` does nothing, and `ResError` returns an appropriate Memory Manager result code. The same is true if the resource data in memory can't be written to the resource fork (for example, because the disk is full). If the `resProtected` attribute is set for the resource, `SetResInfo` does nothing, and `ResError` returns the result code `resAttrErr`.

If you want to write changes to the resource map on disk after updating the resource map in memory, call the `ChangedResource` procedure for the same resource after you call `SetResInfo`.

IMPORTANT

Even if you don't call `ChangedResource` after using `SetResInfo` to change the name and resource ID of a resource, the change may be written to disk when the Resource Manager updates the resource fork. If you call `ChangedResource` for any resource in the same resource fork, or if you add or remove a resource, the Resource Manager writes the entire resource map to disk after a call to `UpdateResFile` or when your application terminates. In these cases, all changes to resource information in the resource map become permanent. If you want any of the changes to be temporary, you should restore the original information before the resource is updated. *s*

SPECIAL CONSIDERATIONS

The `SetResInfo` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute does not permit operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `ChangedResource` and `UpdateResFile` procedures, see page 1-88 and page 1-92, respectively.

GetResAttrs

You can use the `GetResAttrs` function to get a resource's attributes.

```
FUNCTION GetResAttrs (theResource: Handle): Integer;
```

`theResource`

A handle to a resource.

DESCRIPTION

Given a handle to a resource, the `GetResAttrs` function returns the resource's attributes as recorded in its entry in the resource map in memory. If the value of the `theResource` parameter isn't a handle to a valid resource, `GetResInfo` does nothing, and the `ResError` function returns the result code `resNotFound`.

The `GetResAttrs` function returns the resource's attributes in the low-order byte of the function result. Each attribute is identified by a specific bit in the low-order byte. If the bit corresponding to an attribute contains 1, then that attribute is set; if the bit contains 0, then that attribute is not set. You can use these constants to refer to each attribute:

```
CONST
```

```
resSysHeap      = 64;  {set if read into system heap}
resPurgeable    = 32;  {set if purgeable}
resLocked       = 16;  {set if locked}
resProtected    = 8;   {set if protected}
resPreload      = 4;   {set if to be preloaded}
resChanged      = 2;   {set if to be written to resource fork}
```

The `resSysHeap` attribute indicates whether the resource is read into the system heap (`resSysHeap` attribute is set to 1) or your application's heap (`resSysHeap` attribute is set to 0).

If the `resPurgeable` attribute is set to 1, the resource is purgeable; if it's 0, the resource is nonpurgeable.

Because a locked resource is nonrelocatable and nonpurgeable, the `resLocked` attribute overrides the `resPurgeable` attribute. If the `resLocked` attribute is 1, the resource is nonpurgeable regardless of whether `resPurgeable` is set. If it's 0, the resource is purgeable or nonpurgeable depending on the value of the `resPurgeable` attribute.

If the `resProtected` attribute is set to 1, your application can't use Resource Manager routines to change the resource ID or resource name, modify the resource contents, or remove the resource from its resource fork. However, you can use the `SetResAttrs` procedure to remove this protection.

If the `resPreload` attribute is set to 1, the Resource Manager reads the resource's resource data into memory immediately after opening its resource fork. You can use this setting to make multiple resources available for your application as soon as possible,

rather than reading each one into memory individually. If both the `resPreload` attribute and the `resLocked` attribute are set, the Resource Manager loads the resource as low in the heap as possible.

If the `resChanged` attribute is set to 1, the resource has been changed; if it's 0, the resource hasn't been changed. This attribute is used only while the resource map is in memory. The `resChanged` attribute must be 0 in the resource fork on disk.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about resource attributes, see “The Resource Map” beginning on page 1-8.

To change a resource's attributes in the resource map in memory, use the `SetResAttrs` procedure. It is described next.

SetResAttrs

You can use the `SetResAttrs` procedure to change a resource's attributes in the resource map in memory.

```
PROCEDURE SetResAttrs (theResource: Handle; attrs: Integer);
```

`theResource`

A handle to a resource.

`attrs`

The resource attributes to set.

DESCRIPTION

Given a handle to a resource, `SetResAttrs` changes the resource attributes of the resource to those specified in the `attrs` parameter. The `SetResAttrs` procedure changes the information in the resource map in memory, not in the file on disk. The `resProtected` attribute changes immediately. Other attribute changes take effect the next time the specified resource is read into memory but are not made permanent until the Resource Manager updates the resource fork.

If the value of the parameter `theResource` isn't a valid handle to a resource, `SetResAttrs` does nothing, and the `ResError` function returns the result code `resNotFound`.

Each attribute is identified by a specific bit in the low-order byte of a word. If the bit corresponding to an attribute contains 1, then that attribute is set; if the bit contains 0, then that attribute is not set. You can use these constants to specify each attribute:

```
CONST
    resSysHeap      = 64;   {set if read into system heap}
    resPurgeable   = 32;   {set if purgeable}
    resLocked      = 16;   {set if locked}
    resProtected   = 8;    {set if protected}
    resPreload     = 4;    {set if to be preloaded}
    resChanged     = 2;    {set if to be written to resource fork}
```

The `resSysHeap` attribute determines whether the resource is read into your application's heap (`resSysHeap` attribute set to 0) or the system heap (`resSysHeap` attribute set to 1). You should set this bit to 0 for your application's resources. Note that if you do set the `resSysHeap` attribute to 1 and the resource is too large for the system heap, the bit is cleared and the resource is read into the application heap.

Set the `resPurgeable` attribute to 1 to make the resource purgeable; you can set it to 0 to make the resource nonpurgeable. However, do not use `SetResAttrs` to make a purgeable resource nonpurgeable.

Because a locked resource is nonrelocatable and nonpurgeable, the `resLocked` attribute overrides the `resPurgeable` attribute. If you set the `resLocked` attribute to 1, the resource is nonpurgeable regardless of whether or not you set `resPurgeable`. If you set the `resLocked` attribute to 0, the resource is purgeable or nonpurgeable depending on the value of the `resPurgeable` attribute.

If you set the `resProtected` attribute to 1, your application can't use Resource Manager routines to change the resource ID or resource name, modify the resource contents, or remove the resource from its resource fork. If you set the `resProtected` attribute to 0, you remove this protection. Note that this attribute change takes effect immediately.

If you set the `resPreload` attribute to 1, the Resource Manager reads the resource's resource data into memory immediately after opening its resource fork. You can use this setting to make multiple resources available for your application as soon as possible, rather than reading each one into memory separately.

The `resChanged` attribute indicates whether or not the resource has been changed; do not use `SetResAttrs` to set the `resChanged` attribute. Be sure the `attrs` parameter passed to `SetResAttrs` doesn't change the current setting of this attribute. To determine the attribute's current setting, call the `GetResAttrs` function. To set the `resChanged` attribute, call the `ChangedResource` procedure. Note that the `resChanged` attribute is used only while the resource map is in memory. The `resChanged` attribute must be 0 in the resource fork on disk.

If you want the Resource Manager to write the modified resource map to disk after a subsequent call to `UpdateResFile` or when your application terminates, call the `ChangedResource` procedure after you call `SetResAttrs`.

S WARNING

Do not use `SetResAttrs` to change a purgeable resource. If you make a purgeable resource nonpurgeable by setting the `resPurgeable` attribute with `SetResAttrs`, the resource doesn't become nonpurgeable until the next time the specified resource is read into memory. Thus, the resource might be purged while you're changing it. s

SPECIAL CONSIDERATIONS

The `SetResAttrs` procedure does not return an error if you are setting the attributes of a resource in a resource file that has a read-only resource map. To find out whether this is the case, use `GetResFileAttrs`.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about resource attributes, see “The Resource Map” beginning on page 1-8.

For a description of the `GetResFileAttrs` function, see page 1-116. To mark a resource as changed, use the `ChangedResource` procedure, described next.

Modifying Resources

The Resource Manager provides two routines that change the `resChanged` attribute of a specified resource. The `ChangedResource` procedure allows you to indicate that a resource in memory has been changed, and the `AddResource` procedure allows you to add a new resource to a resource map.

If the `resChanged` attribute for a resource has been set and your application calls `UpdateResFile` or quits, the Resource Manager writes both the entire resource map and the resource data for that resource to the resource fork of the corresponding file on disk. If the `resChanged` attribute has been set and your application calls `WriteResource`, the Resource Manager writes only the resource's data to disk.

ChangedResource

If you've changed a resource's data or changed an entry in a resource map, you can use the `ChangedResource` procedure to set a flag in the resource's resource map entry in memory to show that you've made changes.

```
PROCEDURE ChangedResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

DESCRIPTION

Given a handle to a resource, the `ChangedResource` procedure sets the `resChanged` attribute for that resource in the resource map in memory. If the `resChanged` attribute for a resource has been set and your application calls `UpdateResFile` or quits, the Resource Manager writes the resource data for that resource (and for all other resources whose `resChanged` attribute is set) and the entire resource map to the resource fork of the corresponding file on disk. If the `resChanged` attribute for a resource has been set and your application calls `WriteResource`, the Resource Manager writes only the resource data for that resource to disk.

If you change information in the resource map with a call to `SetResInfo` or `SetResAttrs` and then call `ChangedResource` and `UpdateResFile`, the Resource Manager still writes both the resource map and the resource data to disk. If you want any of your changes to the resource map or resource data to be temporary, you must restore the original information before the Resource Manager updates the resource fork on disk.

After writing a resource to disk, the Resource Manager clears the resource's `resChanged` attribute in the appropriate entry of the resource map in memory.

If the given handle isn't a handle to a resource, if the modified resource data can't be written to the resource fork, or if the `resProtected` attribute is set for the modified resource, `ChangedResource` does nothing. To find out whether any of these errors occurred, call `ResError`.

When your application calls `ChangedResource`, the Resource Manager attempts to reserve enough disk space to contain the changed resource. If the modified resource data can't be written to the resource fork (for example, if there's not enough room on disk), the `resChanged` attribute is not set to 1. If this is the case and you call `UpdateResFile` or `WriteResource`, the Resource Manager won't know that the resource data has been changed. Thus, the routine won't write the modified resource data to the resource fork and won't return an error. For this reason, always make sure that the `ResError` function returns the result code `noErr` after a call to `ChangedResource`.

IMPORTANT

If you need to make changes to a purgeable resource using routines that may cause the Memory Manager to purge the resource, you should make the resource temporarily not purgeable. To do so, use the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState` to make sure the resource data remains in memory while you change it and until the resource data is written to disk. (You can't use the `SetResAttrs` procedure for this purpose, because the changes don't take effect immediately.) First call `HGetState` and `HNoPurge`, then change the resource as necessary. To make a change to a resource permanent, use `ChangedResource` and `UpdateResFile` or `WriteResource`; then call `HSetState` when you're finished. Or, instead of calling `WriteResource` to write the resource data immediately, you can call `SetResPurge` with the `install` parameter set to `TRUE` before making changes to purgeable resource data.

If your application doesn't make its resources purgeable, or if the changes you are making to a purgeable resource don't involve routines that may cause the resource to be purged, you don't need to take these precautions.

SPECIAL CONSIDERATIONS

The `ChangedResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

When called, `ChangedResource` reserves disk space for the changed resource. The procedure reserves space every time you call it, but the resource is not actually written until you call `WriteResource` or `UpdateResFile`. Thus, if you call `ChangedResource` several times before the resource is actually written, the procedure reserves much more space than is needed. If the resource is large, you may unexpectedly run out of disk space. When the resource is actually written, the file's end-of-file (EOF) is set correctly, and the next call to `ChangedResource` will work as expected.

If your application frequently changes the contents of resources (especially large resources), you should call `WriteResource` or `UpdateResFile` immediately after calling `ChangedResource`.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute inconsistent with operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of the `ChangedResource` procedure, see Listing 1-2 on page 1-21 and Listing 1-11 on page 1-38.

For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For descriptions of the `SetResInfo`, `SetResAttrs`, and `SetResPurge` procedures, see page 1-82, page 1-85, and page 1-94, respectively.

For information about using the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState`, see *Inside Macintosh: Memory*.

AddResource

You can use the `AddResource` procedure to add a resource to the current resource file.

```
PROCEDURE AddResource (theData: Handle; theType: ResType;
                      theID: Integer; name: Str255);
```

<code>theData</code>	A handle to data in memory to be added as a resource to the current resource file (not a handle to an existing resource).
<code>theType</code>	The resource type of the resource to be added.
<code>theID</code>	The resource ID of the resource to be added.
<code>name</code>	The name of the resource to be added.

DESCRIPTION

Given a handle to any type of data in memory (but not a handle to an existing resource), `AddResource` adds the given handle, resource type, resource ID, and resource name to the current resource file's resource map in memory. The `AddResource` procedure sets the `resChanged` attribute to 1; it does not set any of the resource's other attributes—that is, all other attributes are set to 0.

S WARNING

The `AddResource` procedure doesn't verify whether the resource ID you pass in the parameter `theID` is already assigned to another resource of the same type. You should call the `UniqueID` or `Unique1ID` function to get a unique resource ID before adding a resource with `AddResource`.

If the `resChanged` attribute of a resource has been set and your application calls `UpdateResFile` or quits, the Resource Manager writes both the resource map and the resource data for that resource to the resource fork of the corresponding file on disk. If the `resChanged` attribute for a resource has been set and your application calls `WriteResource`, the Resource Manager writes only the resource data for that resource to disk.

If you add a resource to the current resource file, the Resource Manager writes the entire resource map to disk when it updates the file. If you want any of your changes to the resource map or resource data to be temporary, you must restore the original information before the Resource Manager updates the file on disk.

If the value of the parameter `theData` is an empty handle (that is, a handle whose master pointer is set to `NIL`), the Resource Manager writes zero-length resource data to disk when it updates the resource. If the value of `theData` is either `NIL` or a handle to an existing resource, `AddResource` does nothing, and the `ResError` function returns the result code `addResFailed`. If the resource map becomes too large to fit in memory, `AddResource` does nothing, and `ResError` returns an appropriate result code. The same is true if the resource data in memory can't be written to the resource fork (for example, because the disk is full).

When your application calls `AddResource`, the Resource Manager attempts to reserve disk space for the new resource. If the new resource data can't be written to the resource fork (for example, if there's not enough room on disk), the `resChanged` attribute is not set to 1. If this is the case and you call `UpdateResFile` or `WriteResource`, the Resource Manager won't know that resource data has been added. Thus, the routine won't write the new resource data to the resource fork and won't return an error. For this reason, always make sure that the `ResError` function returns the result code `noErr` after a call to `AddResource`.

To copy an existing resource, get a handle to the resource you want to copy, call the `DetachResource` procedure, then call `AddResource`. To add the same resource data to several different resource forks, call the Memory Manager function `HandToHand` to duplicate the handle for each resource.

RESULT CODES

<code>noErr</code>	0	No error
<code>addResFailed</code>	-194	<code>AddResource</code> procedure failed

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of the `AddResource` procedure, see Listing 1-4 on page 1-24 and Listing 1-11 on page 1-38.

For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For descriptions of the `UniqueID` and `Unique1ID` functions, see page 1-96. For a description of the `DetachResource` procedure, see page 1-108.

For information about using the Memory Manager procedure `HandToHand`, see *Inside Macintosh: Memory*.

Writing to Resource Forks

The Resource Manager provides three procedures that you can use to write resource information to disk. The `UpdateResFile` procedure updates the resource map and resource data of a resource fork on disk so that it matches the corresponding resource map and resource data in memory. The `WriteResource` procedure updates the resource data of just one resource on disk. The `SetResPurge` procedure sets up the Resource Manager's own purge-warning procedure so that the Memory Manager checks with the Resource Manager before purging a purgeable resource.

UpdateResFile

You can use the `UpdateResFile` procedure to update the resource map and resource data for a resource fork without closing it.

```
PROCEDURE UpdateResFile (refNum: Integer);
```

`refNum` A file reference number for a resource fork.

DESCRIPTION

Given the reference number of a file whose resource fork is open, `UpdateResFile` performs three tasks. The first task is to change, add, or remove resource data in the file's resource fork to match the resource map in memory. Changed resource data for each resource is written only if that resource's `resChanged` bit has been set by a successful call to `ChangedResource` or `AddResource`. The `UpdateResFile` procedure calls the `WriteResource` procedure to write changed or added resources to the resource fork.

The second task is to compact the resource fork, closing up any empty space created when a resource was removed, made smaller, or made larger. If a resource is made larger, the Resource Manager writes it at the end of the resource fork rather than at its original location. It then compacts the space occupied by the original resource data. The actual size of the resource fork is adjusted when a resource is removed or made larger, but not when a resource is made smaller.

The third task is to write the resource map in memory to the resource fork if your application has called the `ChangedResource` procedure for any resource listed in the resource map or if it has added or removed a resource. All changes to resource information in the resource map become permanent at this time; if you want any of these changes to be temporary, you must restore the original information before calling `UpdateResFile`.

If there's no open resource fork with the given reference number, `UpdateResFile` does nothing, and the `ResError` function returns the result code `resNotFound`. If the value of the `refNum` parameter is 0, it represents the System file's resource fork. If you call `UpdateResFile` but the `mapReadOnly` attribute of the resource fork is set, `UpdateResFile` does nothing, and the `ResError` function returns the result code `resAttrErr`.

Because the `CloseResFile` procedure calls `UpdateResFile` before it closes the resource fork, you need to call `UpdateResFile` directly only if you want to update the file without closing it.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute inconsistent with operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of `UpdateResFile`, see Listing 1-11 on page 1-38. For descriptions of the `ChangedResource`, `AddResource`, and `CloseResFile` procedures, see page 1-88, page 1-90, and page 1-110, respectively. The `WriteResource` procedure is described next.

WriteResource

You can use the `WriteResource` procedure to write resource data in memory immediately to a file's resource fork. Note that `WriteResource` does not write the resource's resource map entry to disk.

```
PROCEDURE WriteResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

DESCRIPTION

Given a handle to a resource, `WriteResource` checks the `resChanged` attribute of that resource. If the `resChanged` attribute is set to 1 (after a successful call to the `ChangedResource` or `AddResource` procedure), `WriteResource` writes the resource data in memory to the resource fork, then clears the `resChanged` attribute in the resource's resource map in memory.

Note

When your application calls `ChangedResource` or `AddResource`, the Resource Manager attempts to reserve disk space for the changed resource. If the modified resource data can't be written to the resource fork (for example, if there's not enough room on disk), the `resChanged` attribute is not set to 1. If this is the case and you call `WriteResource`, the Resource Manager won't know that the resource data has been changed. Thus, the routine won't write the modified resource data to the resource fork and won't return an error. For this reason, always make sure that the `ResError` function returns the result code `noErr` after a call to `ChangedResource` or `AddResource`.

If the resource is purgeable and has been purged, `WriteResource` writes zero-length resource data to the resource fork. If the resource's `resProtected` attribute is set to 1, `WriteResource` does nothing, and the `ResError` function returns the result code `noErr`. The same is true if the `resChanged` attribute is not set (that is, set to 0). If the given handle isn't a handle to a resource, `WriteResource` does nothing, and `ResError` returns the result code `resNotFound`.

The resource fork is updated automatically when your application quits, when you call `UpdateResFile`, or when you call `CloseResFile` (which in turn calls `UpdateResFile`). Thus, you should call `WriteResource` only if you want to write just one or a few resources immediately.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about resource attributes, see "The Resource Map" beginning on page 1-8. For descriptions of the `ChangedResource` and `AddResource` procedures, see page 1-88 and page 1-90, respectively. For a description of the `UpdateResFile` procedure, see page 1-92. For a description of the `CloseResFile` procedure, see page 1-110.

SetResPurge

You can use the `SetResPurge` procedure to have the Memory Manager pass the handle of a resource to the Resource Manager before purging the data specified by that handle.

```
PROCEDURE SetResPurge (install: Boolean);
```

`install` A Boolean value that specifies whether the Memory Manager checks with the Resource Manager before purging a resource handle.

DESCRIPTION

Specify `TRUE` in the `install` parameter to make the Memory Manager pass the handle for a resource to the Resource Manager before purging the resource data to which the handle points. The Resource Manager determines whether the handle points to a resource in the application heap. It also checks if the resource's `resChanged` attribute is set to 1. If these two conditions are met, the Resource Manager calls the `WriteResource` procedure to write the resource's resource data to the resource fork before returning control to the Memory Manager.

Specify `FALSE` in the `install` parameter to restore the normal state, so that the Memory Manager purges resource data when it needs to without calling the Resource Manager.

You can use `SetResPurge` in applications that modify purgeable resources. You should also take precautions in such applications to ensure that the resource won't be purged while you're changing it.

SPECIAL CONSIDERATIONS

If you call `SetResPurge` with the `install` parameter set to `TRUE` and then call the Memory Manager procedure `MoveHHi` to move a handle to a resource, the Resource Manager calls the `WriteResource` procedure to write the resource data to disk even if the data has not been changed. To prevent this, call `SetResPurge` with the `install` parameter set to `FALSE` before you call `MoveHHi`, then call `SetResPurge` with the `install` parameter set to `TRUE` immediately after you call `MoveHHi`.

Whenever you call `SetResPurge` with the `install` parameter set to `TRUE`, the Resource Manager installs its own purge-warning procedure, overriding any purge-warning procedure you've specified to the Memory Manager.

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `SetResAttrs` and `WriteResource` procedures, see page 1-85 and page 1-93, respectively.

For more information about the Memory Manager procedure `MoveHHi`, see *Inside Macintosh: Memory*.

Getting a Unique Resource ID

The Resource Manager provides two routines that return a unique resource ID. The `UniqueID` function returns a resource ID that isn't currently assigned to any resource of the specified type in any open resource fork. The `Unique1ID` function returns a resource ID that isn't currently assigned to any resource of the specified type in the resource fork of the current resource file.

UniqueID

You can use the `UniqueID` function to get a unique resource ID for a resource.

```
FUNCTION UniqueID (theType: ResType): Integer;
```

`theType` A resource type.

DESCRIPTION

The `UniqueID` function returns as its function result a resource ID greater than 0 that isn't currently assigned to any resource of the specified type in any open resource fork. You should use this function before adding a new resource to ensure that you don't duplicate a resource ID and override an existing resource.

SPECIAL CONSIDERATIONS

In versions of system software earlier than System 7, the `UniqueID` function may return a resource ID in the range 0 through 127, which is generally reserved for system resources. You should check that the resource ID returned is not in this range. If it is, call `UniqueID` again, and continue doing so until you get a resource ID greater than 127.

In System 7 and later versions, `UniqueID` won't return a resource ID of less than 128.

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about restrictions on resource IDs for specific resource types, see "Resource IDs" on page 1-46.

Unique1ID

You can use the `Unique1ID` function to get a resource ID that's unique with respect to resources in the current resource file.

```
FUNCTION Unique1ID (theType: ResType): Integer;
```

`theType` A resource type.

DESCRIPTION

The `Unique1ID` function returns as its function result a resource ID greater than 0 that isn't currently assigned to any resource of the specified type in the current resource file. You should use this routine before adding a new resource to ensure that you don't duplicate a resource ID and override an existing resource.

SPECIAL CONSIDERATIONS

In versions of system software earlier than System 7, the `Unique1ID` function may return a resource ID in the range 0 through 127, which is generally reserved for system resources. You should check that the resource ID returned is not in this range. If it is, call `Unique1ID` again, and continue doing so until you get a resource ID greater than 127.

In System 7 and later versions, `Unique1ID` won't return a resource ID of less than 128.

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about restrictions on resource IDs for specific resource types, see "Resource IDs" on page 1-46.

Counting and Listing Resource Types

The Resource Manager provides several routines that count or list resource types. The `CountResources` function returns the total number of resources of a given type that are currently available in all resource forks open to your application, and the `Count1Resources` function returns the total number of resources of a given type in the current resource file.

You can call the `GetIndResource` function repeatedly to generate handles to all resources of a given type in all resource forks open to your application. You can call the `Get1IndResource` function repeatedly to generate handles to all resources of a given type in the current resource file.

The `CountTypes` function tells you the number of resource types in all resource forks open to your application. The `Count1Types` function tells you the number of resource types in the current resource file. You can call the `GetIndType` procedure repeatedly to get all the resource types available in all resource forks open to your application. Similarly, you can call the `Get1IndType` procedure repeatedly to get all the resource types available in the current resource file.

CountResources

You can use the `CountResources` function to get the total number of available resources of a given type.

```
FUNCTION CountResources (theType: ResType): Integer;
```

`theType` A resource type.

DESCRIPTION

Given a resource type, the `CountResources` function reads the resource maps in memory for all resource forks open to your application. It returns as its function result the total number of resources of the given type in all resource forks open to your application.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

Count1Resources

You can use the `Count1Resources` function to get the total number of resources of a given type in the current resource file.

```
FUNCTION Count1Resources (theType: ResType): Integer;
```

`theType` A resource type.

DESCRIPTION

Given a resource type, the `Count1Resources` function reads the resource map in memory of the current resource file. It returns as its function result the total number of resources of the given type in the current resource file only.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `CountResources` function, see Listing 1-10 on page 1-34.

GetIndResource

You can use the `GetIndResource` function repeatedly to get handles to all resources of a given type in all resource forks open to your application.

```
FUNCTION GetIndResource (theType: ResType;
                        index: Integer): Handle;
```

`theType` A resource type.

`index` An integer ranging from 1 to the number of resources of a given type returned by `CountResources`, which is the number of resource types in all open resource forks.

DESCRIPTION

Given an index ranging from 1 to the number of resources of a given type returned by `CountResources` (that is, the number of resources of that type in all resource forks open to your application), the `GetIndResource` function returns a handle to a resource of the given type. If you call `GetIndResource` repeatedly over the entire range of the index, it returns handles to all resources of the given type in all resource forks open to your application.

The function reads the resource data into memory if it's not already there, unless you've called `SetResLoad` with the `load` parameter set to `FALSE`.

IMPORTANT

If you've called `SetResLoad` with the `load` parameter set to `FALSE` and the data isn't already in memory, `GetIndResource` returns an empty handle (a handle whose master pointer is set to `NIL`). This can also happen if you read resource data for a purgeable resource into memory and then call `SetResLoad` with the `load` parameter set to `FALSE`. If the resource data is later purged and you call the `GetIndResource` function, `GetIndResource` returns an empty handle. You should test for an empty handle in these situations. To make the handle a valid handle to resource data in memory, you can call the `LoadResource` procedure. [u](#)

The `GetIndResource` function returns handles for all resources in the most recently opened resource fork first, and then for those in resource forks opened earlier in reverse chronological order.

Note

The `UseResFile` procedure affects which file the Resource Manager searches first when looking for a particular resource; this is not the case when you use `GetIndResource` to get an indexed resource. ^u

If you want to find out how many resources of a given type are in a particular resource fork, set the current resource file to that resource fork, then call `Count1Resources` and use `Get1IndResource` to get handles to the resources of that type.

If you provide an index to `GetIndResource` that's either 0 or negative, `GetIndResource` returns `NIL`, and the `ResError` function returns the result code `resNotFound`. If the given index is larger than the value returned by `CountResources`, `GetIndResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. If the resource to be read won't fit into memory, `GetIndResource` returns `NIL`, and `ResError` returns the appropriate result code.

SPECIAL CONSIDERATIONS

The `GetIndResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `CountResources` function, see page 1-98. For a description of the `UseResFile` procedure, see page 1-69. For descriptions of the `SetResLoad` and `LoadResource` procedures, see page 1-79 and page 1-80, respectively.

Get1IndResource

You can use the `Get1IndResource` function repeatedly to get handles to all resources of a given type in the current resource file.

```
FUNCTION Get1IndResource (theType: ResType;
                        index: Integer): Handle;
```

`theType` A resource type.

`index` An integer ranging from 1 to the number of resources of a given type returned by `Count1Resources`, which is the number of resource types in the current resource file.

DESCRIPTION

Given an index ranging from 1 to the number of resources of a given type returned by `Count1Resources` (that is, the number of resources of that type in the current resource file), the `Get1IndResource` function returns a handle to a resource of the given type. If you call `Get1IndResource` repeatedly over the entire range of the index, it returns handles to all resources of the given type in the current resource file.

The function reads the resource data into memory if it's not already there, unless you've called `SetResLoad` with the `load` parameter set to `FALSE`.

IMPORTANT

If you've called `SetResLoad` with the `load` parameter set to `FALSE` and the data isn't already in memory, `Get1IndResource` returns an empty handle (that is, a handle whose master pointer is set to `NIL`). This can also happen if you read resource data for a purgeable resource into memory and then call `SetResLoad` with the `load` parameter set to `FALSE`. If the resource data is later purged and you call the `Get1IndResource` function, `Get1IndResource` returns an empty handle. You should test for an empty handle in these situations. To make the handle a valid handle to resource data in memory, you can call the `LoadResource` procedure. u

If you provide an index to `Get1IndResource` that's either 0 or negative, `Get1IndResource` returns `NIL`, and the `ResError` function returns the result code `resNotFound`. If the given index is larger than the value returned by `Count1Resources`, `Get1IndResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. If the resource to be read won't fit into memory, `Get1IndResource` returns `NIL`, and `ResError` returns the appropriate result code.

SPECIAL CONSIDERATIONS

The `Get1IndResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `Get1IndResource` function, see Listing 1-10 on page 1-34.

For a description of the `Count1Resources` function, see page 1-98. For a description of the `UseResFile` procedure, see page 1-69. For descriptions of the `SetResLoad` and `LoadResource` procedures, see page 1-79 and page 1-80, respectively.

CountTypes

You can use the `CountTypes` function to get the number of resource types in all resource forks open to your application.

```
FUNCTION CountTypes: Integer;
```

DESCRIPTION

The `CountTypes` function reads the resource maps in memory for all resource forks open to your application. It returns an integer representing the total number of unique resource types.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

Count1Types

You can use the `Count1Types` function to get the number of resource types in the current resource file.

```
FUNCTION Count1Types: Integer;
```

DESCRIPTION

The `Count1Types` function reads the resource map in memory for the current resource file. It returns an integer representing the total number of unique resource types in the current resource file.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

GetIndType

You can call the `GetIndType` procedure repeatedly to get all the resource types available in all resource forks open to your application.

```
PROCEDURE GetIndType (VAR theType: ResType; index: Integer);
```

`theType` `GetIndType` returns, in this parameter, the resource type for the specified index among all the resource forks open to your application.

`index` An integer ranging from 1 to the number of resource types in all resource forks open to your application.

DESCRIPTION

Given an index number from 1 to the number of resource types in all resource forks open to your application (as returned by `CountTypes`), the `GetIndType` procedure returns a resource type in the parameter `theType`. You can call `GetIndType` repeatedly over the entire range of the index to get all the resource types available in all resource forks open to your application. If the given index isn't in the range from 1 to the number of resource types as returned by `CountTypes`, `GetIndType` returns four null characters (ASCII code 0).

SPECIAL CONSIDERATIONS

The `GetIndType` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about the `CountTypes` function, see page 1-102.

Get1IndType

You can use the `Get1IndType` procedure to get all the resource types available in the current resource file.

```
PROCEDURE Get1IndType (VAR theType: ResType; index: Integer);
```

`theType` `Get1IndType` returns, in this parameter, the resource type with the specified index in the current resource file.

`index` An integer ranging from 1 to the number of resource types in the current resource file.

DESCRIPTION

Given an index number from 1 to the number of resource types in the current resource file (as returned by `Count1Types`), the `Get1IndType` procedure returns a resource type in the parameter `theType`. You can call `Get1IndType` repeatedly over the entire range of the index to get all the resource types available in the current resource file. If the given index isn't in the range from 1 to the number of resource types as returned by `Count1Types`, `Get1IndType` returns four null characters (ASCII code 0).

SPECIAL CONSIDERATIONS

The `Get1IndType` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RESULT CODE

`noErr` 0 No error

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `Count1Types` function, see page 1-102.

Getting Resource Sizes

The Resource Manager provides two routines that allow you to get the size of a resource. The `GetResourceSizeOnDisk` and `GetMaxResourceSize` functions get the exact size and maximum size, respectively, of a resource. To change the size of a resource on disk, use the `SetResourceSize` procedure.

GetResourceSizeOnDisk

You can use the `GetResourceSizeOnDisk` function to get the exact size of a resource. The `GetResourceSizeOnDisk` function is also available as the `SizeResource` function.

```
FUNCTION GetResourceSizeOnDisk (theResource: Handle): LongInt;
```

theResource

A handle to a resource.

DESCRIPTION

Given a handle to a resource, the `GetResourceSizeOnDisk` function checks the resource on disk (not in memory) and returns its exact size, in bytes. If the handle isn't a handle to a valid resource, `GetResourceSizeOnDisk` returns `-1`, and `ResError` returns the result code `resNotFound`.

You can call `GetResourceSizeOnDisk` before reading a resource into memory to make sure there's enough memory available to do so successfully.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

GetMaxResourceSize

You can use the `GetMaxResourceSize` function to get the approximate size of a resource. The `GetMaxResourceSize` function is also available as the `MaxSizeRsrc` function.

```
FUNCTION GetMaxResourceSize (theResource: Handle): LongInt;
```

theResource

Handle to a resource.

DESCRIPTION

Like `GetResourceSizeOnDisk`, `GetMaxResourceSize` takes a handle and returns the size of the corresponding resource. However, `GetMaxResourceSize` does not check the resource on disk; instead, it either checks the resource size in memory or, if the resource is not in memory, calculates its size, in bytes, on the basis of information in the resource map in memory. This gives you an approximate size for the resource that you can count on as the resource's maximum size. It's possible that the resource is actually smaller than the offsets in the resource map indicate because the file has not yet been compacted. If you want the exact size of a resource on disk, either call `GetResourceSizeOnDisk` or call `UpdateResFile` before calling `GetMaxResourceSize`.

If the value of the `theResource` parameter isn't a handle to a valid resource, `GetMaxResourceSize` returns `-1`, and `ResError` returns the result code `resNotFound`.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `UpdateResFile` and `GetResourceSizeOnDisk` routines, see page 1-92 and page 1-105, respectively.

Disposing of Resources

The Resource Manager provides three procedures for disposing of resources. The `ReleaseResource` procedure releases the memory associated with a resource, setting the handle's master pointer to `NIL`, thus making your application's handle to the resource invalid. The `DetachResource` procedure sets a resource's handle in the resource map to `NIL` but keeps the resource data in memory. The `RemoveResource` procedure removes the resource's entry from the resource map in memory; the Resource Manager removes the resource data from memory (and from the file's resource fork) when it updates the file's resource fork.

ReleaseResource

You can use the `ReleaseResource` procedure to release the memory a resource occupies when you have finished using it.

```
PROCEDURE ReleaseResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

DESCRIPTION

Given a handle to a resource, `ReleaseResource` releases the memory occupied by the resource data, if any, and sets the master pointer of the resource's handle in the resource map in memory to `NIL`. If your application previously obtained a handle to that resource, the handle is no longer valid. If your application subsequently calls the Resource Manager to get the released resource, the Resource Manager assigns a new handle.

If the given resource isn't a handle to a resource, `ReleaseResource` does nothing, and `ResError` returns the result code `resNotFound`. Be aware that `ReleaseResource` won't release a resource whose `resChanged` attribute has been set, but `ResError` still returns the result code `noErr`.

SPECIAL CONSIDERATIONS

The `ReleaseResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about releasing resources, see "Releasing and Detaching Resources" beginning on page 1-22. For an example of the use of the `ReleaseResource` procedure, see Listing 1-8 on page 1-29.

DetachResource

You can use the `DetachResource` procedure to set the value of a resource's handle in the resource map in memory to `NIL` while keeping the resource data in memory.

```
PROCEDURE DetachResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

DESCRIPTION

Given a handle to a resource, `DetachResource` sets the value of the resource's handle in the resource map in memory to `NIL`. After this call, the Resource Manager no longer recognizes the handle as a handle to a resource. However, `DetachResource` does not release the memory used for the resource data, and the master pointer is still valid. Thus, you can access the resource data directly by using the handle.

If your application subsequently calls a Resource Manager routine to get the released resource, the Resource Manager assigns a new handle. If the parameter `theResource` doesn't contain a handle to a resource or if the resource's `resChanged` attribute is set, `DetachResource` does nothing. To determine whether either of these errors occurred, call `ResError`.

You can use `DetachResource` if you want to access the resource data directly without using Resource Manager routines. You can also use the `DetachResource` procedure to keep resource data in memory after closing a resource fork.

To copy a resource and install an entry for the duplicate in the resource map, call `DetachResource`, then call `AddResource` using a different resource ID.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute does not permit operation

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about detaching resources, see "Releasing and Detaching Resources" beginning on page 1-22. For an example of the use of the `DetachResource` procedure, see Listing 1-4 on page 1-24.

For a description of the `AddResource` procedure, see page 1-90.

RemoveResource

You can use the `RemoveResource` procedure to remove a resource's entry from the current resource file's resource map in memory. The `RemoveResource` procedure is also available as the `RmveResource` procedure.

```
PROCEDURE RemoveResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

DESCRIPTION

Given a handle to a resource in the current resource file, `RemoveResource` removes the resource entry (resource type, resource ID, resource name, if any, and resource attributes) from the current resource file's resource map in memory.

The `RemoveResource` procedure doesn't immediately release the memory occupied by the resource data; instead, the Resource Manager releases the memory when your application quits, when you call `UpdateResFile`, or when you call `CloseResFile` (which in turn calls `UpdateResFile`). If the `resProtected` attribute for the resource is set or if the `theResource` parameter doesn't contain a handle to a resource, `RemoveResource` does nothing, and `ResError` returns the result code `rmvResFailed`.

IMPORTANT

If you've removed a resource, the Resource Manager writes the entire resource map when it updates the resource fork, and all changes made to the resource map become permanent. If you want any of the changes to be temporary, you should restore the original information before the Resource Manager updates the resource fork. [s](#)

If you want to release the memory before updating or closing the resource fork, call the Memory Manager procedure `DisposeHandle` after you call `RemoveResource`.

SPECIAL CONSIDERATIONS

The `RemoveResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>rmvResFailed</code>	-196	<code>RemoveResource</code> procedure failed

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `AddResource` and `UpdateResFile` procedures, see page 1-90 and page 1-92, respectively. The `CloseResFile` procedure is described next.

For more information about the Memory Manager procedure `DisposeHandle`, see *Inside Macintosh: Memory*.

Closing Resource Forks

When your application terminates, the Resource Manager automatically closes every resource fork open to your application except the System file's resource fork. The `CloseResFile` procedure allows you to close a resource fork before your application terminates.

CloseResFile

You can use the `CloseResFile` procedure to close a resource fork before your application terminates.

```
PROCEDURE CloseResFile (refNum: Integer);
```

`refNum` The file reference number for the resource fork to close.

DESCRIPTION

Given a file reference number for a file whose resource fork is open, the `CloseResFile` procedure performs four tasks. First, it updates the file by calling the `UpdateResFile` procedure. Second, it releases the memory occupied by each resource in the resource fork by calling the `DisposeHandle` procedure. Third, it releases the memory occupied by the resource map. The fourth task is to close the resource fork.

If the `refNum` parameter does not contain a file reference number for a file whose resource fork is open, `CloseResFile` does nothing, and the `ResError` function returns the result code `resFNotFound`. If the value of the `refNum` parameter is 0, it represents the System file and is ignored. You cannot close the System file's resource fork.

When your application terminates, the Resource Manager automatically closes every resource fork open to your application except the System file's resource fork. You need to call the `CloseResFile` procedure only if you want to close a resource fork before your application terminates.

RESULT CODES

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `CloseResFile` procedure, see Listing 1-9 on page 1-32.

For descriptions of the `UpdateResFile` and `ReleaseResource` procedures, see page 1-92 and page 1-107, respectively.

Reading and Writing Partial Resources

You can use the `ReadPartialResource`, `WritePartialResource`, and `SetResourceSize` procedures to work with a portion of a large resource that may not otherwise fit in memory.

When using partial resource routines, you should call the `SetResLoad` procedure, specifying `FALSE` for the `load` parameter, before you call `GetResource`. Using the `SetResLoad` procedure prevents the Resource Manager from reading the entire resource into memory. Be sure to restore the normal state by calling `SetResLoad` again, with the `load` parameter set to `TRUE`, immediately after you call `GetResource`. Then use `ReadPartialResource` to read a portion of the resource into a buffer and `WritePartialResource` as needed to write a portion of the resource from a buffer to disk.

Note that the partial resources routines work with the data in the memory pointed to by the `buffer` parameter, not the memory referenced through the resource's handle. Therefore, you may experience problems if you have a copy of a resource in memory when you are using the partial resource routines. If you have modified the copy in memory and then access the resource on disk using the `ReadPartialResource` procedure, `ReadPartialResource` reads the data on disk, not the data in memory, which is referenced through the resource's handle. Similarly, `WritePartialResource` writes data from the specified buffer, not from the data in memory, which is referenced through the resource's handle.

ReadPartialResource

You can use the `ReadPartialResource` procedure to read part of a resource into memory and work with a small subsection of a large resource.

```
PROCEDURE ReadPartialResource (theResource: Handle;
                               offset: LongInt; buffer: UNIV Ptr;
                               count: LongInt);
```

`theResource`

A handle to a resource.

`offset`

The beginning of the resource subsection to be read, measured in bytes from the beginning of the resource.

CHAPTER 1

Resource Manager

buffer A pointer to the buffer into which the partial resource is to be read.
count The length of the resource subsection.

DESCRIPTION

The `ReadPartialResource` procedure reads the resource subsection identified by the `theResource`, `offset`, and `count` parameters into a buffer specified by the `buffer` parameter. Your application is responsible for the buffer's memory management. You cannot use the `ReleaseResource` procedure to release the memory the buffer occupies.

The `ReadPartialResource` procedure always tries to read resources from disk. If a resource is already in memory, the Resource Manager still reads it from disk, and the `ResError` function returns the result code `resourceInMemory`. If you try to read past the end of a resource or the value of the `offset` parameter is out of bounds, `ResError` returns the result code `inputOutOfBounds`. If the handle in the parameter `theResource` doesn't refer to a resource in an open resource fork, `ResError` returns the result code `resNotFound`.

When using partial resource routines, you should call the `SetResLoad` procedure, specifying `FALSE` for the `load` parameter, before you call `GetResource`. Using the `SetResLoad` procedure prevents the Resource Manager from reading the entire resource into memory. Be sure to restore the normal state by calling `SetResLoad` again, with the `load` parameter set to `TRUE`, immediately after you call `GetResource`. Then use `ReadPartialResource` to read a portion of the resource into a buffer.

Note

If the entire resource is in memory and you want only part of its data, it's faster to use the Memory Manager procedure `BlockMove` instead of the `ReadPartialResource` procedure. If you read a partial resource into memory and then change its size, you can use `SetResourceSize` to change the entire resource's size on disk as necessary. u

SPECIAL CONSIDERATIONS

The `ReadPartialResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `ReadPartialResource` are

Trap macro	Selector
<code>_ResourceDispatch</code>	<code>\$7001</code>

RESULT CODES

noErr	0	No error
resourceInMemory	-188	Resource already in memory
inputOutOfBounds	-190	Offset or count out of bounds
resNotFound	-192	Resource not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `ReadPartialResource` procedure, see Listing 1-12 on page 1-41.

For descriptions of the `GetResource`, `SetResLoad`, and `ReleaseResource` routines, see page 1-73, page 1-79, and page 1-107, respectively. For a description of the `SetResourceSize` procedure, see page 1-115.

For information about the Memory Manager procedure `BlockMove`, see *Inside Macintosh: Memory*.

WritePartialResource

You can use the `WritePartialResource` procedure to write part of a resource to disk when working with a small subsection of a large resource.

```
PROCEDURE WritePartialResource (theResource: Handle;
                               offset: LongInt; buffer: UNIV Ptr;
                               count: LongInt);
```

`theResource`

A handle to a resource.

`offset`

The beginning of the resource subsection to write, measured in bytes from the beginning of the resource.

`buffer`

A pointer to the buffer containing the data to write.

`count`

The length of the resource subsection to write.

DESCRIPTION

The `WritePartialResource` procedure writes the data specified by the `buffer` parameter to the resource subsection identified by the `theResource`, `offset`, and `count` parameters. Your application is responsible for the buffer's memory management.

If the disk or the file is locked, the `ResError` function returns an appropriate File Manager result code. If you try to write past the end of a resource, the Resource Manager attempts to enlarge the resource. The `ResError` function returns the result code `writingPastEnd` if the attempt succeeds. If the Resource Manager cannot enlarge the resource, `ResError` returns an appropriate File Manager result code. If you pass an invalid value in the `offset` parameter, `ResError` returns the result code `inputOutOfBounds`.

The `WritePartialResource` procedure tries to write the data from the buffer to disk. If the attempt is successful and the resource data (referenced through the resource's handle) is in memory, `ResError` returns the result code `resourceInMemory`. In this situation, be aware that the data of the resource subsection on disk matches the data from the buffer, not the resource data referenced through the resource's handle. If the attempt to write the data from the buffer to the disk fails, `ResError` returns an appropriate error.

When using partial resource routines, you should call the `SetResLoad` procedure, specifying `FALSE` for the `load` parameter, before you call `GetResource`. Doing so prevents the Resource Manager from reading the entire resource into memory. Be sure to restore the normal state by calling `SetResLoad` again, with the `load` parameter set to `TRUE`, immediately after you call `GetResource`.

If you read a partial resource into memory and then change its size, you must use `SetResourceSize` to change the entire resource's size on disk as necessary before you write the partial resource.

SPECIAL CONSIDERATIONS

The `WritePartialResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `WritePartialResource` are

Trap macro	Selector
<code>_ResourceDispatch</code>	<code>\$7002</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>dskFullErr</code>	<code>-34</code>	Disk full
<code>resourceInMemory</code>	<code>-188</code>	Resource already in memory
<code>writingPastEnd</code>	<code>-189</code>	Writing past end of file
<code>inputOutOfBounds</code>	<code>-190</code>	Offset or count out of bounds

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `GetResource` and `SetResLoad` routines, see page 1-73 and page 1-79, respectively. The `SetResourceSize` procedure is described next.

SetResourceSize

You can use the `SetResourceSize` procedure to change the size of a resource on disk. This procedure is normally used only with `ReadPartialResource` and `WritePartialResource`.

```
PROCEDURE SetResourceSize (theResource: Handle; newSize: LongInt);
```

`theResource`

A handle to a resource.

`newSize`

The size, in bytes, that you want the resource to occupy on disk.

DESCRIPTION

Given a handle to a resource, `SetResourceSize` sets the size field of the specified resource on disk without writing the resource data. You can change the size of any resource, regardless of the amount of memory you have available.

If the specified size is smaller than the resource's current size on disk, you lose any data from the cutoff point to the end of the resource. If the specified size is larger than the resource's current size on disk, all data is preserved, but the additional area is uninitialized (arbitrary data).

If you read a partial resource into memory and then change its size, you must use `SetResourceSize` to change the entire resource's size on disk as necessary. For example, suppose the entire resource occupies 1 MB and you use `ReadPartialResource` to read in a 200 KB portion of the resource. If you then increase the size of this partial resource to 250 KB, you must call `SetResourceSize` to set the size of the resource on disk to 1.05 MB. Note that in this case you must also keep track of the resource data on disk and move any data that follows the original partial resource on disk. Otherwise, there will be no space for the additional 50 KB when you call `WritePartialResource` to write the modified partial resource to disk.

Under certain circumstances, the Resource Manager overrides the size you set with a call to `SetResourceSize`. For instance, suppose you read an entire resource into memory by calling `GetResource` or related routines, then use `SetResourceSize` successfully to set the resource size on disk, and finally attempt to write the resource to disk using `UpdateResFile` or `WriteResource`. In this case, the Resource Manager adjusts the resource size on disk to conform with the size of the resource in memory.

If the disk is locked or full, or the file is locked, the `SetResourceSize` procedure does nothing, and the `ResError` function returns an appropriate File Manager result code. If the resource is in memory, the Resource Manager tries to set the size of the resource on disk. If the attempt succeeds, `ResError` returns the result code `resourceInMemory`, and the Resource Manager does not update the copy in memory. If the attempt fails, `ResError` returns an appropriate File Manager result code.

SPECIAL CONSIDERATIONS

The `SetResourceSize` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SetResourceSize` are

Trap macro	Selector
<code>_ResourceDispatch</code>	<code>\$7003</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>resourceInMemory</code>	<code>-188</code>	Resource already in memory
<code>writingPastEnd</code>	<code>-189</code>	Writing past end of file

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

Getting and Setting Resource Fork Attributes

The `GetResFileAttrs` function and the `SetResFileAttrs` procedure allow you to get and set a resource fork's attributes. You usually don't need to use these routines.

GetResFileAttrs

You can use the `GetResFileAttrs` function to get the attributes of a resource fork.

```
FUNCTION GetResFileAttrs (refNum: Integer): Integer;
```

`refNum` A file reference number for the resource fork whose attributes you want to get.

DESCRIPTION

Given a file reference number, the `GetResFileAttrs` function returns the attributes of the file's resource fork. Specify 0 in the `refNum` parameter to get the attributes of the System file's resource fork. If there's no open resource fork for the given file reference number, `GetResFileAttrs` does nothing, and the `ResError` function returns the result code `resNotFound`.

Like individual resources, resource forks have attributes that are specified by bits in the low-order byte of a word. The Resource Manager provides the following masks for testing these bits:

CONST

```
mapReadOnly    = 128;  {set if file is read-only}
mapCompact     = 64;   {set to compact file on update}
mapChanged     = 32;   {set to write map on update}
```

When the `mapReadOnly` attribute is set to 1, the Resource Manager doesn't write anything to the resource fork on disk. It also doesn't check whether the resource data can be written to disk when the resource map is modified. When this attribute is set to 1, the `UpdateResFile` and `WriteResource` procedures do nothing, but the `ResError` function returns the result code `noErr`.

When the `mapCompact` attribute is set to 1, the Resource Manager compacts the resource fork when it updates the file. The Resource Manager sets this attribute when a resource is removed or when a resource is made larger and thus must be written at the end of a resource fork. You may want to set the `mapCompact` attribute to force the Resource Manager to compact a resource fork when your changes have made resources smaller.

When the `mapChanged` attribute is set to 1, the Resource Manager writes the resource map to disk when the file is updated. For example, you can set `mapChanged` if you've changed resource attributes only and don't want to call `ChangedResource` because you don't want to write the resource data to disk.

SPECIAL CONSIDERATIONS

The Resource Manager sets the `mapChanged` attribute for the resource fork when you call the `ChangedResource`, the `AddResource`, or the `RemoveResource` procedure.

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-193	Resource file not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `ChangedResource` and `AddResource` procedures, see page 1-88 and page 1-90, respectively. For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For a description of the `RemoveResource` procedure, see page 1-109.

SetResFileAttrs

You can use the `SetResFileAttrs` procedure to change a resource fork's attributes.

```
PROCEDURE SetResFileAttrs (refNum: Integer; attrs: Integer);
```

`refNum` A file reference number for the resource fork whose attributes you want to set.

`attrs` The attributes to set.

DESCRIPTION

Given a file reference number, the `SetResFileAttrs` procedure sets the attributes of the file's resource fork to those specified in the `attrs` parameter. If the `refNum` parameter is 0, it represents the System file's resource fork. However, you shouldn't change the attributes of the System file's resource fork. If there's no resource fork with the given reference number, `SetResFileAttrs` does nothing, and the `ResError` function returns the result code `noErr`.

Like individual resources, resource forks have attributes that are specified by bits in the low-order byte of a word. The Resource Manager provides the following masks for setting these bits:

```
CONST
```

```
mapReadOnly     = 128; {set to make file read-only}
mapCompact     = 64;  {set to compact file on update}
mapChanged     = 32;  {set to write map on update}
```

When the `mapReadOnly` attribute is set to 1, the Resource Manager doesn't write anything to the resource fork on disk. It also doesn't check whether the resource data can be written to disk when the resource map is modified. When this attribute is set to 1, the `UpdateResFile` and `WriteResource` procedures do nothing, but the `ResError` function returns the result code `noErr`.

S WARNING

If you set the `mapReadOnly` attribute but later clear it, the resource data is written to disk even if there's no room for it. This operation may destroy the resource fork. *s*

When the `mapCompact` attribute is set to 1, the Resource Manager compacts the resource fork when it updates the file. The Resource Manager sets this attribute when a resource is removed or when a resource is made larger and thus must be written at the end of a resource fork. You may want to set the `mapCompact` attribute to force the Resource Manager to compact a resource fork when your changes make resources smaller.

When the `mapChanged` attribute is set to 1, the Resource Manager writes the resource map to disk when the file is updated. For example, you can set `mapChanged` if you've changed resource attributes only and don't want to call `ChangedResource` because you don't want to write the resource data to disk.

When the Resource Manager first creates a resource fork after a call to `FSpOpenResFile` or a related routine, it does not set any of the resource forks's attributes—that is, they are all set to 0.

SPECIAL CONSIDERATIONS

The Resource Manager sets the `mapChanged` attribute for the resource fork when you call the `ChangedResource`, the `AddResource`, or the `RemoveResource` procedure.

RESULT CODES

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `ChangedResource` and `AddResource` procedures, see page 1-88 and page 1-90, respectively. For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For a description of the `RemoveResource` procedure, see page 1-109.

Accessing Resource Entries in a Resource Map

The `RsrcMapEntry` function is an advanced routine that provides a way to access the resource entries in a resource map in memory. Because the Resource Manager provides routines for opening, retrieving, and changing resources, there's usually no reason to access resource entries directly.

RsrcMapEntry

To access the resource entries in a resource map in memory directly, you can use the `RsrcMapEntry` function.

```
FUNCTION RsrcMapEntry (theResource: Handle): LongInt;
```

`theResource`

A handle to a resource.

DESCRIPTION

Given a handle to a resource, `RsrcMapEntry` returns the offset of the specified resource's entry from the beginning of the resource map in memory. If it doesn't find the resource entry, `RsrcMapEntry` returns 0, and the `ResError` function returns the result code `resNotFound`. If you pass a handle whose value is `NIL`, `RsrcMapEntry` returns arbitrary data, but `ResError` returns the result code `noErr`.

S WARNING

Because the Resource Manager provides routines for opening, retrieving, and changing resources, there's usually no reason to access a resource map directly. To avoid damaging the file for which it's called, you should use `RsrcMapEntry` extremely carefully. s

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

SEE ALSO

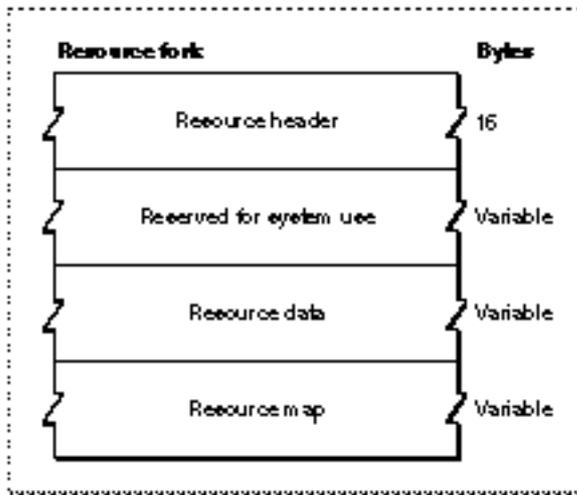
For an overview of the resource map, see "The Resource Map" beginning on page 1-8. For details of the structure of the resource map, see Figure 1-14 on page 1-123.

Resource File Format

You need to know the exact format of a resource fork, which is described in this section, only if you're writing an application that creates or modifies a resource fork directly, without using Resource Manager routines.

Figure 1-11 shows the format of a compiled resource fork.

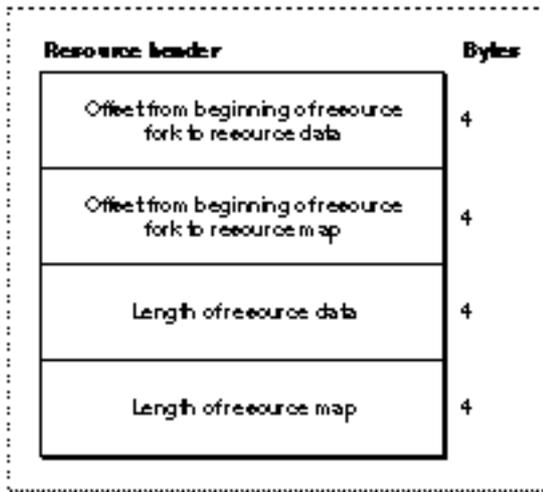
Figure 1-11 Format of a resource fork



As Figure 1-11 shows, every resource fork begins with a resource header. Because the resource header contains an offset to the resource map, the resource map does not necessarily have to be located at the end of the resource fork.

Figure 1-12 shows the format of a resource header.

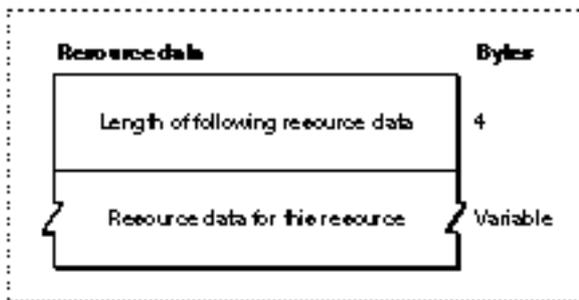
Figure 1-12 Format of a resource header in a resource fork



The resource data in a resource fork consists of the data in its individual resources.

Figure 1-13 shows the format of resource data for a single resource.

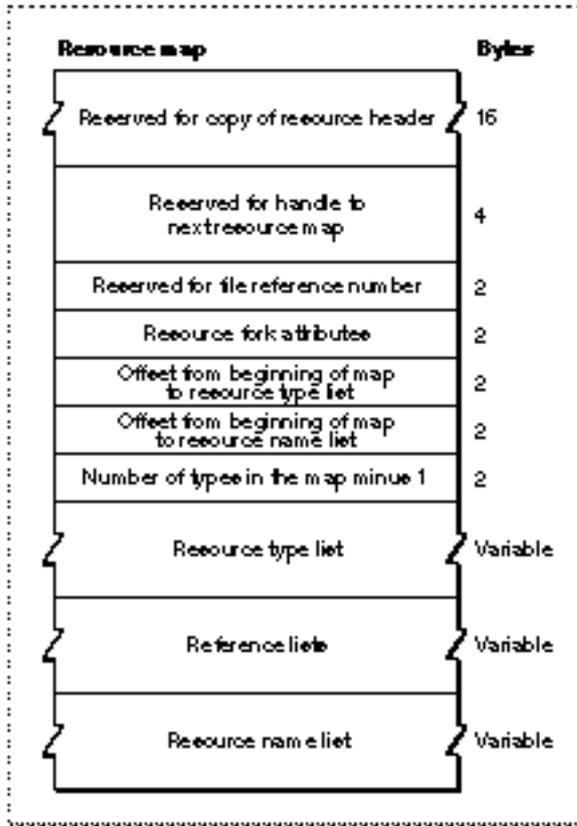
Figure 1-13 Format of resource data for a single resource



For detailed descriptions of the resource data for various standard resource types, see the appropriate books in the *Inside Macintosh* series.

The resource data in a resource fork is followed by the resource map. Figure 1-14 shows the format of a resource map.

Figure 1-14 Format of the resource map in a resource fork

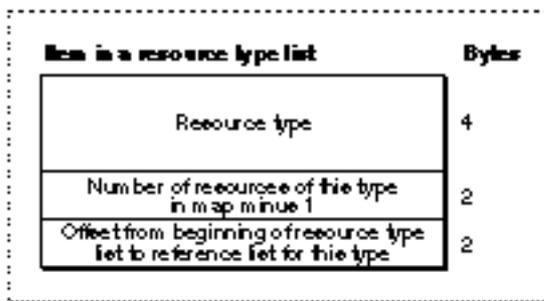


After reading the resource map into memory, the Resource Manager stores the indicated information in the reserved areas at the beginning of the map.

Each item in a resource type list specifies one resource type used in the resource fork, the number of resources of that type, and the location of the reference list for that type.

Figure 1-15 shows the format of an item in a resource type list.

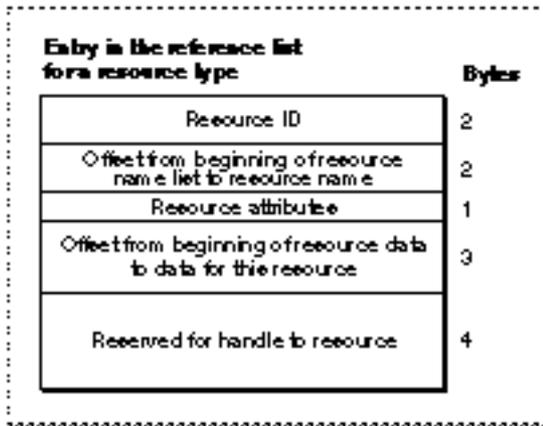
Figure 1-15 Format of an item in a resource type list



The resource type list is followed by the reference lists for each type of resource. Each resource type has a corresponding reference list that contains entries for each resource of that type. The reference lists are contiguous and in the same order as the types in the resource type list.

Figure 1-16 shows the format of an entry in a reference list.

Figure 1-16 Format of an entry in the reference list for a resource type



If a resource does not have a name, the offset to the resource name in the resource's entry in the reference list is -1. If a resource does have a name, the offset identifies the location of the name's entry in the resource name list. Figure 1-17 shows the format of an item in the resource name list.

Figure 1-17 Format of an item in a resource name list

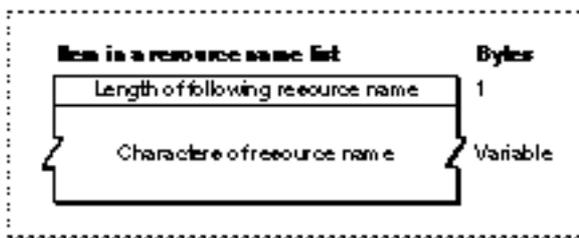
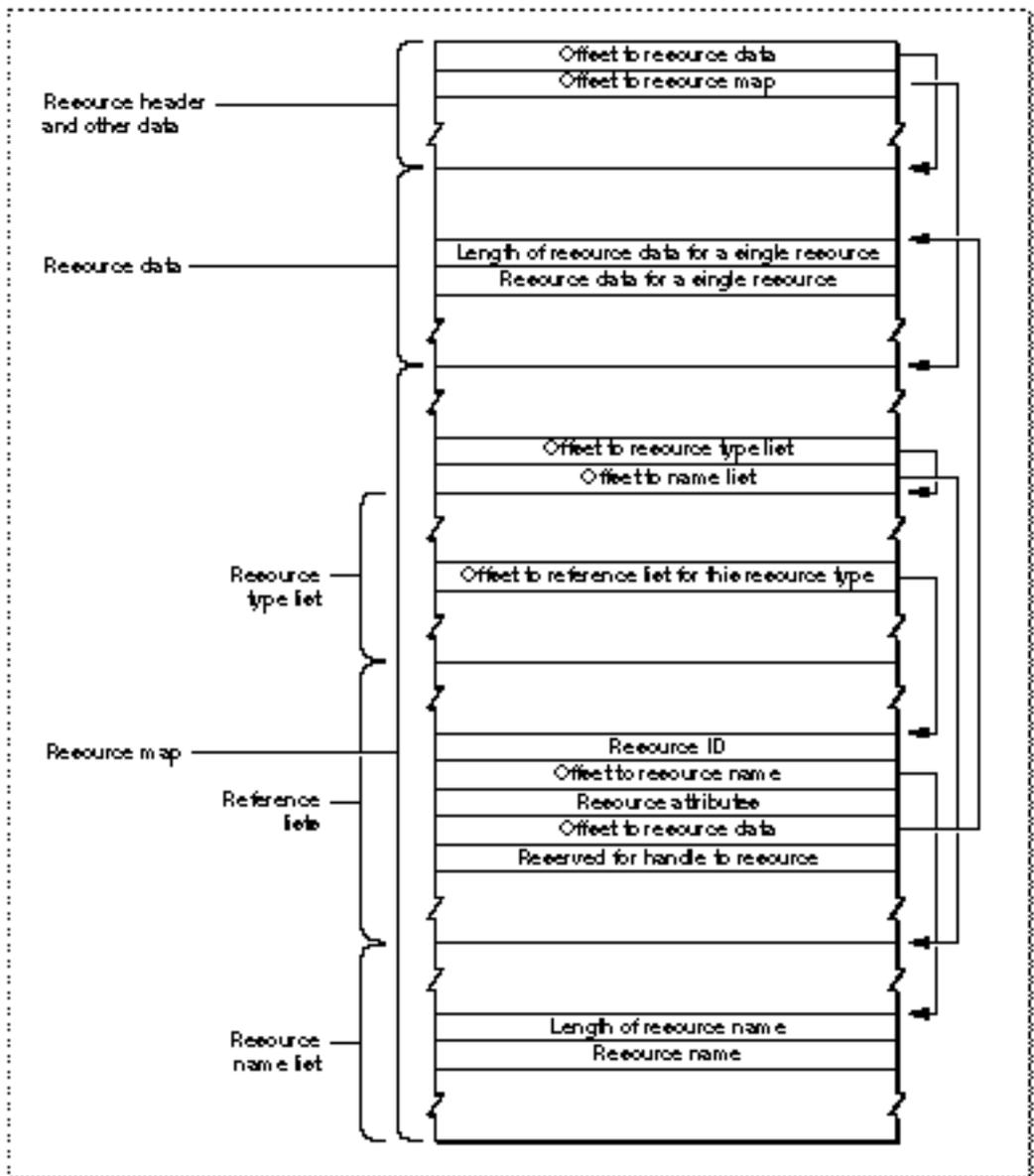


Figure 1-18 illustrates the use of various offsets in the resource header and resource map, including the offsets for an entry in a reference list for an individual resource. Although the figure shows the resource map after the resource data, the resource map can be located almost anywhere in the resource fork as long as the offset to the map in the resource header points to the right location.

Figure 1-18 Offsets in a resource fork and an entry for a single resource in a reference list



Resources in the System File

The System file's resource fork contains resources that are shared by all applications and system software. The sections that follow describe these resources.

S WARNING

Your application should not directly add resources to, delete resources from, or modify resources in the System file. *s*

If your application needs to install drivers, you should ship it with the Installer and an Installer script for drivers. To distribute the Installer, you need to license the Apple system software, which includes the Installer.

The next section describes resources in the System file that provide information about the computer on which your application is currently running, such as the user's name, the computer name, and the current printer type. You can use Resource Manager routines or the `Gestalt` function to obtain this information. Subsequent sections list system software routines kept in packages in the System file and function key resources.

In System 7 and later versions of system software, users can add resources such as scripts, keyboards, and sounds to the System file by dragging the resource icons to the System Folder. Desk accessories and resources such as system extensions are stored in the subdirectories of the System Folder, not in the System file. In System 7.0, users can also add resources such as fonts to the System file by dragging their icons to the System Folder. In System 7.1 and later versions, fonts are stored in a subdirectory of the System Folder rather than in the System file. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for details.)

The folders in the System Folder and some system resources are represented by standard icons. "Standard Icons" beginning on page 1-129 lists the most important standard icons.

User Information Resources

The following resources in the resource fork of the System file provide the user's name, the computer name, the model of computer, the icon for the computer model, and the current printer type:

Information	Resource ID	Resource type	Description
User name	-16096	'STR '	The name of the person who "owns" the computer or is the current user. Use the <code>GetString</code> function with this resource ID to return the user name.
Computer name	-16413	'STR '	The name of the computer, which is distinct from the user name and from any internal hard disks that may be present. The default name of the computer is "User name's Macintosh." Use the <code>GetString</code> function with this resource ID to return the computer name.
Computer model	-16395	'STR#'	The model of the computer, such as Macintosh SE/30 or Macintosh IIfx. The Gestalt selector for the computer model is <code>gestaltMachineModel</code> , and the Gestalt function returns a response value for this selector. You can use this value as an index into the 'STR#' resource using the <code>GetIndString</code> procedure. You should never use the model of the computer as an indication of what software features or hardware may be available.
Computer icon	Value of response parameter returned from Gestalt	'ICN#' 'icl4' 'icl8' 'ics#' 'ics4' 'ics8'	The icon for the computer model, such as the Macintosh II or Macintosh IIfx. The icons for computers are stored in icon families. The Gestalt selector for the computer icon is <code>gestaltMachineIcon</code> . Use the value from the response value for this selector as the resource ID of the icon resource you want. (For more information about icon families, see the chapter "Finder Interface" in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> .)
Printer type	-8192	'STR '	The type of printer to which the computer sends documents, such as a LaserWriter printer. There is no method for retrieving the name of the printer. Use the <code>GetString</code> function to return the type of printer.

You should use `GetString`, not `GetResource`, to get the string for the user name or the computer name. Once you have the string, you should not release it, dispose of it, or make it purgeable. You will find that the resource was already loaded when you asked for it, so it should remain loaded when you are finished. Do not change the contents of either of these strings or mark them as changed. System 6 and earlier versions of system software do not necessarily have the computer name resource, and for this reason you should provide error checking as appropriate.

The `GetString` function, `GetIndString` procedure, and `Gestalt` function are documented in *Inside Macintosh: Operating System Utilities*.

Packages

A package is a set of routines and data types that forms a part of the Toolbox or Operating System and is stored as a resource of type 'PACK'. In early models of the Macintosh computer, all packages were disk-based and brought into memory only when needed; some packages are now in ROM. The System file contains the standard Macintosh packages and the resources they use or own.

Package name	Resource ID
List Manager	0
Disk Initialization Manager	2
Standard File Package	3
Floating-Point Arithmetic Package	4
Transcendental Functions Package	5
Text Utilities	6
Text Utilities (formerly referred to as the Binary-Decimal Conversion Package)	7
Apple Event Manager	8
PPC Browser	9
Edition Manager	11
Color Picker	12
Data Access Manager	13
Help Manager	14
Picture Utilities	15

Function Key Resources

Function key resources (of the 'FKEY' resource type) are Command-Shift-number key combinations that are captured and processed by the `WaitNextEvent` function. The screen utility resource (a function key resource with resource ID 3) produces a picture of the main screen, contained in a 'PICT' file, when the user presses Command-Shift-3. The 'FKEY' resource IDs 0 through 4 are reserved for future use by Apple Computer, Inc. The `WaitNextEvent` function is described in the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Standard Icons

System software uses icons to represent documents, applications, folders, disks, and other elements of the Macintosh interface. Many of these standard icons are stored in the System file. You can design your own icons for your application and its documents. If you do not provide your own icons, the Finder displays a default icon. Your application can retrieve any of the icons in the System file by using the `GetResource` function. You should refer to these icons by their constant names and not by their resource IDs. For a description of the `GetResource` function, see page 1-73.

Most icons are available in at least two sizes: large (32 by 32 pixels) and small (16 by 16 pixels). They are also available in three bit depths: 8-bit, 4-bit, and black-and-white. An icon family consists of the large and small icons for an object, each with a mask, and each available in the three different color depths. See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for information about how to create your own icons.

Many of the icons in the System file are also available in a small size (16 by 16 pixels), represented by the 'SICN' resource. These icons are used in Standard File Package dialog boxes. The Finder also uses icons in the System file to display in its windows the contents of disks or folders by name, date, size, or kind. The Views menu in System 7 allows the user to display large or small icons for a given window.

The icons listed in Table 1-4 represent default icons for documents (including special classes of documents such as stationery), applications, and desk accessories. The icons show the 'icl8' resource from the icons' icon family. You can include customized versions of the icons in Table 1-4 with your documents and applications. There are icon families and 'SICN' resources for all of these icons unless otherwise noted.

Table 1-4 Document and application icons

Constant name and icon	Resource ID	Description
genericDocumentIconResource 	-4000	The default document icon. The Finder displays this icon if your application does not provide its own icon for documents.
genericApplicationIconResource 	-3996	The default application icon. The Finder displays this icon for any application that does not provide its own icon.
genericDeskAccessoryIconResource 	-3991	The default desk accessory icon. In System 7 and later versions, desk accessories are represented on the desktop as applications are, each with its own icon. The Finder displays this icon for any desk accessory that does not provide its own icon.
genericEditionFileIconResource 	-3989	The default edition file icon. (See <i>Inside Macintosh: Interapplication Communication</i> for information about editions.)
genericStationeryIconResource 	-3985	The default stationery file icon. (See <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for information about stationery.)
genericPreferencesIconResource 	-3971	The default preferences file icon. Preference files appear in the Preferences folder, which is located inside the System Folder. There is no 'SICN' resource for this icon.
genericQueryDocumentIconResource 	-16506	The default query document icon. (See <i>Inside Macintosh: Interapplication Communication</i> for information about query documents.) There is no 'SICN' resource for this icon.
genericExtensionIconResource 	-16415	The default extension icon. The Finder displays this icon for any extension that does not have its own icon. Extension files appear in the Extensions folder, which is located inside the System Folder. There is no 'SICN' resource for this icon.

The icons listed in Table 1-5 represent the different types of folders found on the desktop. The icons shown are the 'ic18' resource for the icons' icon families. There are icon families and 'SICN' resources for all of these icons unless otherwise noted.

Table 1-5 Folder icons

Constant name and icon	Resource ID	Description
genericFolderIconResource 	-3999	The default folder icon.
privateFolderIconResource 	-3994	The icon for a folder to which the user does not have access. It is dimmed and has a distinctly marked border. The Finder displays an alert box when a user without privileges attempts to open this folder.
ownedFolderIconResource 	-3980	The icon for a folder that is owned by a particular user, usually on a shared volume such as a file server. There is no 'SICN' resource for this icon.
dropFolderIconResource 	-3979	The icon for a folder in which any user may store documents, applications, and so on, but from which only a specified group of users can retrieve the contents. There is no 'SICN' resource for this icon.
sharedFolderIconResource 	-3978	The icon for a folder that the owner has made available for file sharing. There is no 'SICN' resource for this icon.
mountedFolderIconResource 	-3977	The icon for a folder that a guest has mounted on a remote volume. This icon appears only for the guest. There is no 'SICN' resource for this icon.

The icons listed in Table 1-6 represent the different types of folders found in the System Folder. The icons shown are the 'ic18' resource for the icons' icon families. You should not alter the appearance of these icons. There are only icon families for these icons.

Table 1-6 System Folder icons

Constant name and icon	Resource ID	Description
systemFolderIconResource 	-3983	The System Folder icon. This folder contains the System file and other system-related folders.
appleMenuFolderIconResource 	-3982	The Apple Menu Items folder icon. This folder contains items found in the Apple menu.
startupFolderIconResource 	-3981	The Startup Items folder icon. This folder contains documents, aliases, applications, and other objects that open when the computer starts up.
controlPanelFolderIconResource 	-3976	The Control Panels folder icon. This folder contains control panels.
printMonitorFolderIconResource 	-3975	The PrintMonitor Documents folder icon. This folder contains documents that are in the queue to be printed.
preferencesFolderIconResource 	-3974	The Preferences folder icon. This folder contains preferences files for the Finder and other software that needs to remember user preferences.
extensionsFolderIconResource 	-3973	The Extensions folder icon. This folder contains system extensions.
fontsFolderIconResource 	-3968	The Fonts folder icon. This folder contains fonts (both bitmapped and outline).

The icons listed in Table 1-7 appear on the desktop. The icons shown are the 'ic18' resource for the icons' icon families. There are icon families and 'SICN' resources for these icons unless otherwise noted.

Table 1-7 Desktop icons

Constant name and icon	Resource ID	Description
floppyIconResource 	-3998	The default icon for any disk, 3.5-inch or otherwise, whose driver doesn't supply its own icon.
trashIconResource 	-3993	The default empty Trash icon.
fullTrashIconResource 	-3984	The default full Trash icon, with bulging midsection. There is no 'SICN' resource for this icon.

The icons listed in Table 1-8 are used only by the Standard File Package and are available only as an 'SICN' resource. The pop-up menu in the standard file dialog boxes indicates where the list of files shown in the dialog box is located (whether on the desktop, at the top level of a volume, or inside a series of folders on a volume).

Table 1-8 Standard File Package icons

Constant name and icon	Resource ID	Description
openFolderIconResource 	-3997	The open folder icon, which appears in a pop-up menu only. The standard file dialog boxes display this icon to indicate which folder is currently open.
genericHardDiskIconResource 	-3995	The hard disk icon, which appears in a pop-up menu only. The same icon is used to represent internal and external disks. A different icon may appear on the desktop, because the manufacturer of the hard disk can design a special icon for a particular volume.

continued

Table 1-8 Standard File Package icons (continued)

Constant name and icon	Resource ID	Description
desktopIconResource 	-3992	The desktop icon, which appears in a pop-up menu only. The standard file dialog boxes display this icon to indicate which files and folders are available on the desktop.
genericFileServerIconResource 	-3972	The file server volume icon. This represents any servers open on the desktop. A different icon may appear on the desktop, because the manufacturer can design a special icon for a particular server.
genericSuitcaseIconResource 	-3970	The suitcase icon. This represents any suitcase, such as font suitcases or desk accessory suitcases. There are different icons for these suitcases in larger sizes, depending on the contents.

ROM Resources

The information in this section is useful only for designers of specialized programs that need to access ROM resources directly, bypassing any patches in the System file, or that need to override ROM resources.

Inserting the ROM Resource Map

Many system resources are stored in ROM. System software calls the `InitResources` function during system startup, and the Resource Manager creates a special heap zone in the system heap and builds a resource map that points to the ROM resources.

The Resource Manager normally searches ROM resources only when you use the `RGetResource` function to get a handle to the resource, and even then only after it searches the System file's resource fork. To search for a resource in ROM before searching the System file's resource fork, your application must first alter the resource search order by inserting the ROM resource map in front of the System file's resource map.

When the value of the global variable `RomMapInsert` is `TRUE`, the Resource Manager inserts the ROM resource map before the System file's resource map for the next call only. When the value of `RomMapInsert` is `TRUE`, the adjacent variable `TmpResLoad` determines whether the value of the global variable `ResLoad` is considered `TRUE` or `FALSE`, overriding the actual value of `ResLoad` for the next call only. The values of the `RomMapInsert` and `TmpResLoad` variables are cleared after each call to a Resource Manager routine.

The `RGetResource` function first calls `GetResource`. If `GetResource` cannot locate the requested resource in the resource chain, `RGetResource` sets `RomMapInsert` to `TRUE`, then calls `GetResource` again.

To set the `RomMapInsert` and `TmpResLoad` variables in tandem yourself, you can use two global constants. Set the system global variable `RomMapInsert` to the global constant `mapTrue` to insert the ROM resource map with `SetResLoad(TRUE)`. Set `RomMapInsert` to the global constant `mapFalse` to insert the ROM resource map with `SetResLoad(FALSE)`.

There is no real resource fork associated with the ROM resources; the ROM resource map has a path number of 1 (an illegal path reference number). There are two ways to determine whether a handle references a ROM resource. First, you can set up `RomMapInsert` and `TmpResLoad` and call `HomeResFile`; if 1 is returned, the handle is to a ROM resource. Second, you can dereference the handle and check whether the master pointer points to ROM by comparing it to the global variable `ROMBase`.

Overriding ROM Resources

You can override some of the ROM resources, such as `'CURS'` resources, simply by putting the substitute resource in your application's resource fork. Other ROM resources, however, such as `'DRVr'` and `'PACK'` resources, cannot be overridden in this way because they are already referenced and in use when your application is launched.

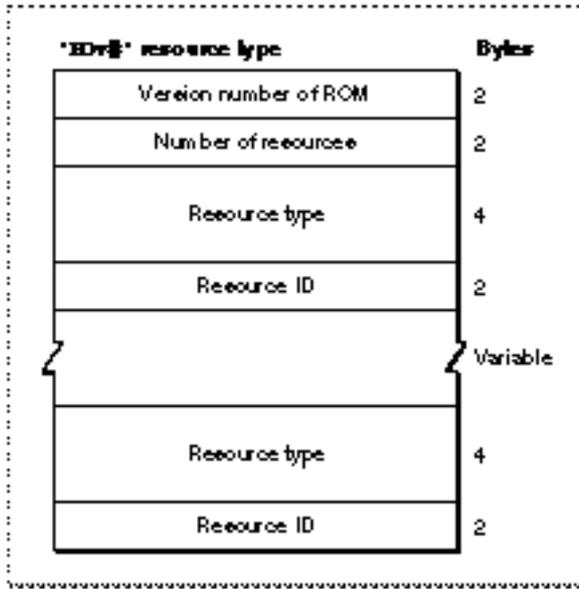
On startup, system software creates a list of ROM resources that should not be referenced. This list is based on information stored in the System file's resource fork in an `'ROv#'` resource whose version word matches the version word of the ROM. You can modify the `'ROv#'` resource so that it includes the ROM resources that you want to override.

S WARNING

You should not override ROM resources unless absolutely necessary. Before overriding ROM resources, you should understand the situation completely. s

Figure 1-19 shows the structure of an 'ROv#' resource.

Figure 1-19 Structure of a compiled ROM override ('ROv#') resource



For information on modifying an 'ROv#' resource, write to Macintosh Developer Technical Support.

Summary of the Resource Manager

Pascal Summary

Constants

CONST

```

gestaltResourceMgrAttr      = 'rsrc';    {Gestalt selector ResMgr}
gestaltPartialRsrcs        = 0;          {check this bit in the }
                                { response parameter}

{resource attributes}
resSysHeap                  = 64;         {set if read into system }
                                { heap}
resPurgeable                = 32;         {set if purgeable}
resLocked                   = 16;         {set if locked}
resProtected                = 8;          {set if protected}
resPreload                  = 4;          {set if to be preloaded}
resChanged                  = 2;          {set if to be written to }
                                { resource fork}

{resource file attributes}
mapReadOnly                  = 128;       {set to make file read-only}
mapCompact                   = 64;       {set to compact file on }
                                { update}
mapChanged                   = 32;       {set to write map on update}

{values for setting the RomMapInsert and TmpResLoad global variables}
mapTrue                      = $FFFF;    {insert ROM map w/ }
                                { TmpResLoad = TRUE}
mapFalse                     = $FF00;    {insert ROM map w/ }
                                { TmpResLoad = FALSE}

{system icon definition IDs}
genericDocumentIconResource  = -4000;    {default document icon}
genericFolderIconResource    = -3999;    {default folder icon}
floppyIconResource           = -3998;    {default disk icon}
openFolderIconResource       = -3997;    {open folder icon}
genericApplicationIconResource = -3996;  {default application }
                                { icon}

```

CHAPTER 1

Resource Manager

genericHardDiskIconResource	= -3995;	{hard disk icon}
privateFolderIconResource	= -3994;	{folder without privileges } { for this user icon}
trashIconResource	= -3993;	{default empty Trash icon}
desktopIconResource	= -3992;	{desktop icon}
genericDeskAccessoryIconResource	= -3991;	{default desk accessory icon}
genericEditionFileIconResource	= -3989;	{default edition icon}
genericStationeryIconResource	= -3985;	{default stationery icon}
systemFolderIconResource	= -3983;	{System Folder icon}
appleMenuFolderIconResource	= -3982;	{Apple Menu Items } { folder icon}
genericFileServerIconResource	= -3972;	{file server icon}
genericPreferencesIconResource	= -3971;	{default preferences } { file icon}
genericSuitcaseIconResource	= -3970;	{default suitcase icon}
genericMoverObjectIconResource	= -3969;	{System file object icon}
genericQueryDocumentIconResource	= -16506;	{default query } { document icon}
genericExtensionIconResource	= -16415;	{default extensions icon}
fullTrashIconResource	= -3984;	{default full Trash icon}
startupFolderIconResource	= -3981;	{Startup Items folder icon}
ownedFolderIconResource	= -3980;	{owned folder icon}
dropFolderIconResource	= -3979;	{drop folder icon}
sharedFolderIconResource	= -3978;	{shared folder icon}
mountedFolderIconResource	= -3977;	{mounted folder icon}
controlPanelFolderIconResource	= -3976;	{Control Panels folder icon}
printMonitorFolderIconResource	= -3975;	{PrintMonitor } { Documents folder icon}
preferencesFolderIconResource	= -3974;	{Preferences folder icon}
extensionsFolderIconResource	= -3973;	{Extensions folder icon}
fontsFolderIconResource	= -3968;	{Fonts folder icon}

Data Type

```
TYPE ResType = PACKED ARRAY[1..4] OF Char;
```

Routines

Initializing the Resource Manager

```
FUNCTION InitResources:      Integer;
PROCEDURE RsrcZoneInit;
```

Checking for Errors

```
FUNCTION ResError:          Integer;
```

Creating an Empty Resource Fork

```
PROCEDURE FSpCreateResFile (spec: FSSpec; creator: OSType;
                             fileType: OSType; scriptTag: ScriptCode);
PROCEDURE HCreateResFile   (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255);
PROCEDURE CreateResFile    (fileName: Str255);
```

Opening Resource Forks

```
FUNCTION FSpOpenResFile     (spec: FSSpec; permission: SignedByte): Integer;
FUNCTION HOpenResFile       (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255;
                             permission: SignedByte): Integer;
FUNCTION OpenRFPPerm        (fileName: Str255; vRefNum: Integer;
                             permission: SignedByte): Integer;
FUNCTION OpenResFile        (fileName: Str255): Integer;
```

Getting and Setting the Current Resource File

```
FUNCTION CurResFile:        Integer;
PROCEDURE UseResFile        (refNum: Integer);
FUNCTION HomeResFile        (theResource: Handle): Integer;
```

Reading Resources Into Memory

```

FUNCTION GetResource      (theType: ResType; theID: Integer): Handle;
FUNCTION Get1Resource    (theType: ResType; theID: Integer): Handle;
FUNCTION GetNamedResource (theType: ResType; name: Str255): Handle;
FUNCTION Get1NamedResource (theType: ResType; name: Str255): Handle;
FUNCTION RGetResource    (theType: ResType; theID: Integer): Handle;
PROCEDURE SetResLoad     (load: Boolean);
PROCEDURE LoadResource   (theResource: Handle);

```

Getting and Setting Resource Information

```

PROCEDURE GetResInfo      (theResource: Handle; VAR theID: Integer;
                          VAR theType: ResType; VAR name: Str255);
PROCEDURE SetResInfo      (theResource: Handle; theID: Integer;
                          name: Str255);
FUNCTION GetResAttrs      (theResource: Handle): Integer;
PROCEDURE SetResAttrs     (theResource: Handle; attrs: Integer);

```

Modifying Resources

```

PROCEDURE ChangedResource (theResource: Handle);
PROCEDURE AddResource     (theData: Handle; theType: ResType;
                          theID: Integer; name: Str255);

```

Writing to Resource Forks

```

PROCEDURE UpdateResFile   (refNum: Integer);
PROCEDURE WriteResource   (theResource: Handle);
PROCEDURE SetResPurge     (install: Boolean);

```

Getting a Unique Resource ID

```

FUNCTION UniqueID         (theType: ResType): Integer;
FUNCTION Unique1ID        (theType: ResType): Integer;

```

Counting and Listing Resource Types

```

FUNCTION CountResources   (theType: ResType): Integer;
FUNCTION Count1Resources  (theType: ResType): Integer;
FUNCTION GetIndResource   (theType: ResType; index: Integer): Handle;
FUNCTION Get1IndResource  (theType: ResType; index: Integer): Handle;

```

Resource Manager

```

FUNCTION CountTypes:      Integer;
FUNCTION Count1Types:    Integer;
PROCEDURE GetIndType     (VAR theType: ResType; index: Integer);
PROCEDURE Get1IndType    (VAR theType: ResType; index: Integer);

```

Getting Resource Sizes

{these routines also available as SizeResource and MaxSizeRsrc, respectively}

```

FUNCTION GetResourceSizeOnDisk
                                (theResource: Handle): LongInt;
FUNCTION GetMaxResourceSize
                                (theResource: Handle): LongInt;

```

Disposing of Resources

```

PROCEDURE ReleaseResource (theResource: Handle);
PROCEDURE DetachResource  (theResource: Handle);
{The RemoveResource procedure is also available as RmveResource}
PROCEDURE RemoveResource  (theResource: Handle);

```

Closing Resource Forks

```

PROCEDURE CloseResFile      (refNum: Integer);

```

Reading and Writing Partial Resources

```

PROCEDURE ReadPartialResource
                                (theResource: Handle;
                                 offset: LongInt; buffer: UNIV Ptr;
                                 count: LongInt);
PROCEDURE WritePartialResource
                                (theResource: Handle;
                                 offset: LongInt; buffer: UNIV Ptr;
                                 count: LongInt);
PROCEDURE SetResourceSize (theResource: Handle; newSize: LongInt);

```

Getting and Setting Resource Fork Attributes

```

FUNCTION GetResFileAttrs (refNum: Integer): Integer;
PROCEDURE SetResFileAttrs (refNum: Integer; attrs: Integer);

```

Accessing Resource Entries in a Resource Map

```

FUNCTION RsrcMapEntry      (theResource: Handle): LongInt;

```

C Summary

Constants

```

enum {
#define gestaltResourceMgrAttr      'rsrc' /*Gestalt selector ResMgr*/
#define gestaltPartialRsrcs        = 0    /*check this bit in the */
                                      /* response parameter*/
};
enum {
    /*resource attributes*/
    resSysHeap          = 64,    /*set if read into system heap*/
    resPurgeable        = 32,    /*set if purgeable*/
    resLocked           = 16,    /*set if locked*/
    resProtected        = 8,     /*set if protected*/
    resPreload          = 4,     /*set if to be preloaded*/
    resChanged          = 2,     /*set if to be written */
                                      /* to resource fork*/

    /*resource fork attributes*/
    mapReadOnly         = 128,   /*set to make file */
                                      /* read-only*/
    mapCompact          = 64,    /*set to compact file */
                                      /* on update*/
    mapChanged         = 32,    /*set to write map */
                                      /* on update*/

    /*values for setting the RomMapInsert and TmpResLoad global variables*/
    mapTrue             = 0xFFFF, /*insert ROM map w/ */
                                      /* TmpResLoad = TRUE*/
    mapFalse            = 0xFF00 /*insert ROM map w/ */
                                      /* TmpResLoad = FALSE*/
};
enum {
    /*system icon definition IDs*/
    genericDocumentIconResource      = -4000,    /*default document icon*/
    genericStationeryIconResource     = -3985,    /*default stationery icon*/
    genericEditionFileIconResource    = -3989,    /*default edition icon*/
    genericApplicationIconResource    = -3996,    /*default application icon*/
    genericDeskAccessoryIconResource  = -3991,    /*default desk accessory */
                                      /* icon*/
    genericFolderIconResource         = -3999,    /*default folder icon*/
    privateFolderIconResource         = -3994,    /*folder without privileges*/
                                      /* for this user icon*/
};

```

CHAPTER 1

Resource Manager

```
floppyIconResource           = -3998,    /*default disk icon*/
trashIconResource           = -3993,    /*default empty Trash icon*/
desktopIconResource         = -3992,    /*desktop icon*/
openFolderIconResource      = -3997,    /*open folder icon*/
genericHardDiskIconResource = -3995,    /*hard disk icon*/
genericFileServerIconResource = -3972,    /*file server icon*/
genericSuitcaseIconResource = -3970,    /*default suitcase icon*/
genericMoverObjectIconResource = -3969,    /*System file object icon*/
genericPreferencesIconResource = -3971,    /*default preferences */
                                /* file icon*/

genericQueryDocumentIconResource = -16506,    /*default query doc icon*/
genericExtensionIconResource = -16415,    /*default extension icon*/
systemFolderIconResource    = -3983,    /*System Folder icon*/
appleMenuFolderIconResource = -3982,    /*Apple Menu Items */
                                /* folder icon*/

};

enum {
    startupFolderIconResource = -3981,    /*Startup Items folder icon*/
    ownedFolderIconResource   = -3980,    /*owned folder icon*/
    dropFolderIconResource    = -3979,    /*drop folder icon*/
    sharedFolderIconResource  = -3978,    /*shared folder icon*/
    mountedFolderIconResource = -3977,    /*mounted folder icon*/
    controlPanelFolderIconResource = -3976,    /*Control Panels folder */
                                /* icon*/

    printMonitorFolderIconResource = -3975,    /*PrintMonitor */
                                /* Documents folder icon*/

    preferencesFolderIconResource = -3974,    /*Preferences folder icon*/
    extensionsFolderIconResource = -3973,    /*Extensions folder icon*/
    fontsFolderIconResource    = -3968,    /*Fonts folder icon*/
    fullTrashIconResource      = -3984    /*default full Trash icon*/
};
```

Data Type

```
typedef unsigned long ResType;
```

Routines

Initializing the Resource Manager

```
pascal short InitResources (void);
pascal void RsrcZoneInit (void);
```

Checking for Errors

```
pascal short ResError (void);
```

Creating an Empty Resource Fork

```
pascal void FSpCreateResFile
    (const FSSpec *spec, OSType creator,
     OSType fileType, ScriptCode scriptTag);
pascal void HCreateResFile (short vRefNum, long dirID,
    ConstStr255Param fileName);
pascal void CreateResFile (ConstStr255Param fileName);
```

Opening Resource Forks

```
pascal short FSpOpenResFile
    (const FSSpec *spec, SignedByte permission);
pascal short HOpenResFile (short vRefNum, long dirID,
    ConstStr255Param fileName,
    char permission);
pascal short OpenRFPPerm (ConstStr255Param fileName, short vRefNum,
    char permission);
pascal short OpenResFile (ConstStr255Param fileName);
```

Getting and Setting the Current Resource File

```
pascal short CurResFile (void);
pascal void UseResFile (short refNum);
pascal short HomeResFile (Handle theResource);
```

Reading Resources Into Memory

```
pascal Handle GetResource (ResType theType, short theID);
pascal Handle Get1Resource (ResType theType, short theID);
pascal Handle GetNamedResource
    (ResType theType, ConstStr255Param name);
pascal Handle Get1NamedResource
    (ResType theType, ConstStr255Param name);
```

Resource Manager

```
pascal Handle RGetResource (ResType theType, short theID);
pascal void SetResLoad (Boolean load);
pascal void LoadResource (Handle theResource);
```

Getting and Setting Resource Information

```
pascal void GetResInfo (Handle theResource, short *theID,
                       ResType *theType, Str255 name);
pascal void SetResInfo (Handle theResource, short theID,
                       ConstStr255Param name);
pascal short GetResAttrs (Handle theResource);
pascal void SetResAttrs (Handle theResource, short attrs);
```

Modifying Resources

```
pascal void ChangedResource (Handle theResource);
pascal void AddResource (Handle theData, ResType theType,
                        short theID, ConstStr255Param name);
```

Writing to Resource Forks

```
pascal void UpdateResFile (short refNum);
pascal void WriteResource (Handle theResource);
pascal void SetResPurge (Boolean install);
```

Getting a Unique Resource ID

```
pascal short UniqueID (ResType theType);
pascal short UniqueID (ResType theType);
```

Counting and Listing Resource Types

```
pascal short CountResources (ResType theType);
pascal short Count1Resources (ResType theType);
pascal Handle GetIndResource (ResType theType, short index);
pascal Handle Get1IndResource (ResType theType, short index);
pascal short CountTypes (void);
pascal short Count1Types (void);
```

CHAPTER 1

Resource Manager

```
pascal void GetIndType      (ResType *theType, short index);
pascal void Get1IndType    (ResType *theType, short index);
```

Getting Resource Sizes

```
/*the GetResourceSizeOnDisk routine is also available as SizeResource*/
pascal long GetResourceSizeOnDisk
                (Handle theResource);

/*the GetMaxResourceSize routine is also available as MaxSizeRsrc*/
pascal long GetMaxResourceSize
                (Handle theResource);
```

Disposing of Resources

```
pascal void ReleaseResource
                (Handle theResource);

pascal void DetachResource (Handle theResource);

/*the RemoveResource routine is also available as RvmeResource*/
pascal void RemoveResource (Handle theResource);
```

Closing Resource Forks

```
pascal void CloseResFile   (short refNum);
```

Reading and Writing Partial Resources

```
pascal void ReadPartialResource
                (Handle theResource, long offset,
                void *buffer, long count);

pascal void WritePartialResource
                (Handle theResource, long offset,
                const void *buffer, long count);

pascal void SetResourceSize
                (Handle theResource, long newSize);
```

Getting and Setting Resource Fork Attributes

```
pascal short GetResFileAttrs
                (short refNum);

pascal void SetResFileAttrs
                (short refNum, short attrs);
```

Accessing Resource Entries in a Resource Map

```
pascal long RsrcMapEntry    (Handle theResource);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

```
_ResourceDispatch
```

Selector	Routine
\$7001	ReadPartialResource
\$7002	WritePartialResource
\$7003	SetResourceSize

```
_HighLevelFSDispatch
```

Selector	Routine
\$0000	FSpOpenResFile
\$000E	FSpCreateResFile

Global Variables

TopMapHndl	long	Handle to resource map of most recently opened resource fork
SysMapHndl	long	Handle to System file's resource fork
SysMap	word	File reference number of System file's resource fork
CurMap	word	File reference number of current resource file
ResLoad	word	Current SetResLoad state
ResErr	word	Current value of ResError
ResErrProc	long	Address of resource error procedure
SysResName	length byte followed by up to 19 characters	Name of System file's resource fork
RomMapInsert	byte	Flag for whether to insert ROM resource map
TmpResLoad	byte	Temporary SetResLoad state for calls using RomMapInsert

Result Codes

noErr	0	No error
dirFulErr	-33	Directory full
dskFulErr	-34	Disk full
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
eofErr	-39	End of file
tmfoErr	-42	Too many files open
fnfErr	-43	File not found
wPrErr	-44	Disk is write-protected
fLckdErr	-45	File is locked
vLckdErr	-46	Volume is locked
dupFNerr	-48	Duplicate filename (rename)
opWrErr	-49	File already open with write permission
permErr	-54	Permissions error (on file open)
extFSerr	-58	Volume belongs to an external file system
memFullErr	-108	Not enough room in heap zone
dirNFerr	-120	Directory not found
resourceInMemory	-188	Resource already in memory
writingPastEnd	-189	Writing past end of file
inputOutOfBounds	-190	Offset or count out of bounds
resNotFound	-192	Resource not found
resFNotFound	-193	Resource file not found
addResFailed	-194	AddResource procedure failed
rmvResFailed	-196	RemoveResource procedure failed
resAttrErr	-198	Attribute inconsistent with operation
mapReadErr	-199	Map inconsistent with operation

Scrap Manager

Contents

Introduction to the Scrap Manager	2-4
The Clipboard	2-10
Intelligent Cut and Paste	2-10
About the Scrap Manager	2-12
Location of the Scrap	2-12
Using the Scrap Manager	2-14
Getting Information About the Scrap	2-15
Putting Data in the Scrap	2-15
Handling the Cut Command	2-15
Handling the Copy Command	2-19
Handling Suspend Events	2-19
Getting Data From the Scrap	2-20
Handling the Paste Command	2-20
Handling Resume Events	2-25
Converting Data Between a Private Scrap and the Scrap	2-26
Converting Data Between the TextEdit Scrap and the Scrap	2-28
Handling Editing Operations in Dialog Boxes	2-31
Scrap Manager Reference	2-31
Data Structures	2-32
The Scrap Information Record	2-32
The Scrap Format Types	2-33
Routines	2-34
Getting Information About the Scrap	2-34
Writing Information to the Scrap	2-35
Reading Information From the Scrap	2-38
Transferring Data Between the Scrap in Memory and the Scrap on Disk	2-40

CHAPTER 2

Summary of the Scrap Manager	2-42
Pascal Summary	2-42
Constants	2-42
Data Types	2-42
Routines	2-42
C Summary	2-43
Data Types	2-43
Routines	2-44
Assembly-Language Summary	2-45
Data Structures	2-45
Result Codes	2-45

Scrap Manager

This chapter describes how your application can allow the user to cut, copy, and paste data between documents or within a document by using the Scrap Manager. When you copy data, your application writes the data to a specific location, and your application writes the data using a standard format. The Scrap Manager makes this data available to other applications. Furthermore, when your application copies data such as text or graphics, you write the data using the standard formats that all Macintosh applications should support. By using standard formats, the user can copy and paste data between documents created by your application and others.

The Scrap Manager supports the sharing of static data between applications. That is, once the data is pasted into another document, there is no connection between the data that was pasted and the original source of the data. To support dynamic sharing of data, where the user can copy data from one document into another document and receive automatic updating of the information when the data in the original document changes, use the Edition Manager. See *Inside Macintosh: Interapplication Communication* for information on the Edition Manager.

You can also support the copying and pasting of sounds, movies, publishers or subscribers, and other formats. For specific information on supporting sounds and movies, see *Inside Macintosh: Sound* and *Inside Macintosh: QuickTime*, respectively. For information on supporting publishers and subscribers, see the chapter “Edition Manager” in *Inside Macintosh: Interapplication Communication*.

If the Translation Manager is available, the Scrap Manager uses its services as necessary to translate data in one format into another format. For specific information on the Translation Manager, see the chapter “Translation Manager” in this book.

If your application uses only TextEdit for all text input, you can use TextEdit routines to cut, copy, and paste data. For complete information on TextEdit, see the chapter “TextEdit” in *Inside Macintosh: Text*.

To support the copying and pasting of data in dialog boxes, use Dialog Manager routines. See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to create and handle dialog boxes.

This chapter discusses the Edit menu commands Cut, Copy, and Paste. For specific information on how to create and handle menus in your application, see the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

To use this chapter, you should be familiar with the Event Manager, in particular, how to handle suspend and resume events. See the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information on the Event Manager.

This chapter begins by describing how the copy-and-paste operation works and the user interface behind it. The chapter then discusses how you can

- n get information about the current contents of the scrap
- n read data from the scrap
- n write data to the scrap

Introduction to the Scrap Manager

You can use the Scrap Manager to

- n copy and paste data within a document created by your application
- n copy and paste data between different documents created by your application
- n copy and paste data between documents created by your application and documents created by other applications

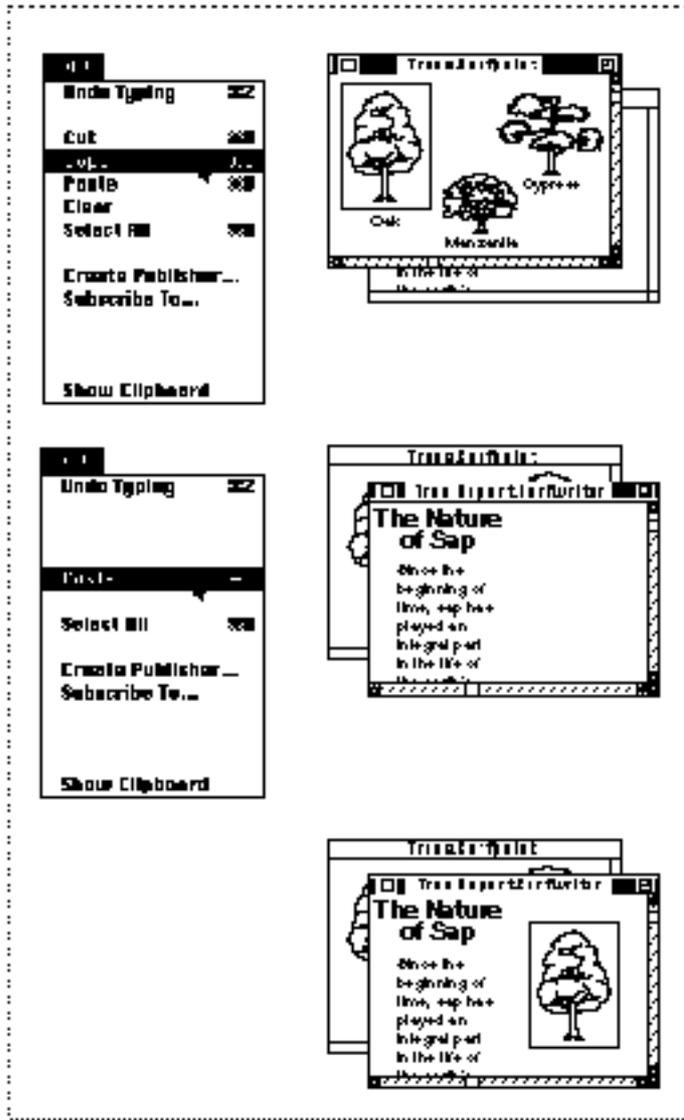
Figure 2-1 shows two documents from two applications (SurfPaint and SurfWriter) that the user currently has open. The user can select the data to copy from the SurfPaint document, choose Copy from the Edit menu, activate the SurfWriter document, then choose Paste from the Edit menu.

In the example shown in Figure 2-1, when the user chooses Copy, the SurfPaint application writes the selected data to the scrap. When the user chooses Paste, the SurfWriter application reads any data from the scrap and inserts the data at the current insertion point.

For each application, the Scrap Manager maintains a storage area to hold the last data cut or copied by the user. The area that is available to an application for this purpose is referred to as the **scrap**. The scrap can reside either in memory or on disk. All applications that support copy-and-paste operations read data from and write data to the scrap.

Whenever the user cuts or copies data, your application should write the data to the scrap (replacing the previous contents of the scrap); and whenever the user pastes data, your application should read the data from the scrap. Alternatively, your application can choose to use its own private scrap, and only write data to and read data from the scrap when necessary. If you use a private scrap, you must copy the data from your private scrap to the scrap upon receiving a suspend event. Upon receiving a resume event you should determine whether the data in the scrap has changed and, if so, either immediately copy the data from the scrap to your private scrap or copy the data from the scrap to your private scrap when the user next chooses the Paste command.

Figure 2-1 Copying and pasting data between two applications using the scrap



Scrap Manager

You use the Edit menu commands Cut, Copy, and Paste to support cutting, copying, and pasting of data within a document and between documents. Table 2-1 describes the actions your application should perform to support these three commands.

Table 2-1 Actions your application performs in response to editing commands

Edit command	Actions your application performs
Cut	Remove the data in the current selection (if any) and save the data, either in the scrap or in your application's private scrap.
Copy	Copy the data in the current selection (if any) and save the copied data, either in the scrap or in your application's private scrap.
Paste	Paste the last data (if any) that the user cut or copied (you get the data to paste by reading the scrap or your application's private scrap). Paste the data at the insertion point, replacing any current selection.

You should implement the editing commands as described in Table 2-1 so that when the user chooses the Paste command—whether applied to the same document or another, in the same application or another—the data last operated upon by the user (cut or copied) can be inserted into the current document. Note that if your application implements the Clear command, in response to the Clear command your application should remove the data in the current selection but should not save the data into the scrap.

The nature of the data that the user can transfer varies between the application that the user copies data from and the application that the user pastes data into. The amount of information retained also depends on the capabilities of the applications supporting the copy-and-paste operation. For example, an application that allows a user to record and edit sounds may write a copied sound to the scrap both in 'snd' and 'TEXT' formats. Other applications choose which format to read from the scrap. A word processor that attempts to paste the sound data may not be able to read the sound in the 'snd' format but should be able to read the data in the 'TEXT' format.

Scrap Manager

You write data to the scrap using the standard formats that all Macintosh applications should support: 'PICT' and 'TEXT'. These scrap format types are defined as follows:

- n 'TEXT': a series of ASCII characters
- n 'PICT': a QuickDraw picture, which is a saved sequence of drawing commands that can be played back with the `DrawPicture` procedure

Your application must be able to write at least one standard format ('PICT' or 'TEXT') to the scrap and should be able to read both. In addition, your application can support other optional popular scrap format types (such as 'snd' or 'movv'). Your application can also write its own private format to the scrap, but must always write one of the standard formats as well.

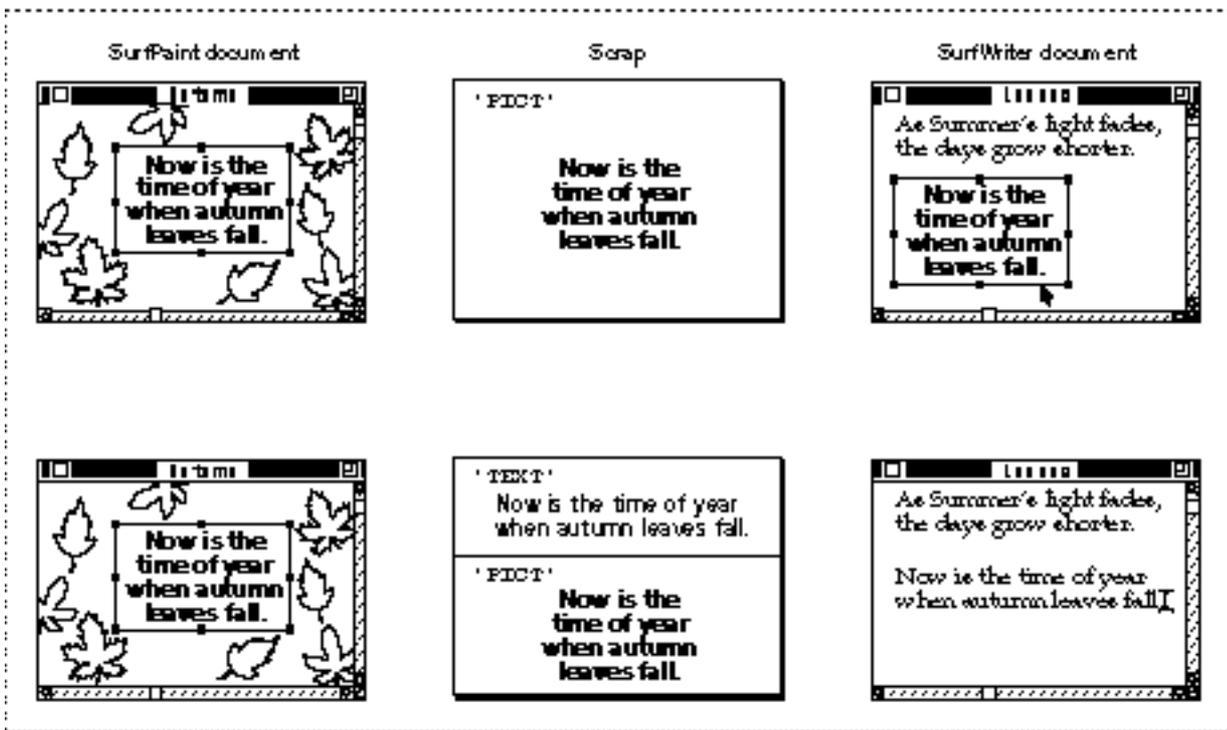
When your application requests data from the scrap, it must specify the scrap format type that the Scrap Manager should retrieve from the scrap. Your application typically requests its preferred scrap format first; if that format isn't available, it requests the data specifying another format type.

If you request a scrap format type that isn't in the scrap and the Translation Manager is available, the Scrap Manager uses the Translation Manager to attempt to convert the data of a scrap format type that does exist in the scrap into the scrap format type requested by your application. For example, if the SurfWriter application requests data from the scrap in the 'SURF' scrap format type, and the data in the scrap is available in the format types 'TEXT', 'PICT', and 'SDBS' (SurfDB's private scrap format type), the Scrap Manager uses the Translation Manager to convert any one of the scrap format types 'TEXT', 'PICT', or 'SDBS' into the 'SURF' scrap format type. The Translation Manager looks in the Extensions folder for a translator that can perform one of these translations. If such a translator is available (for example, a translator that can translate the 'SDBS' scrap format type into the 'SURF' scrap format type), the Translation Manager uses the translator to translate the data in the scrap into the requested scrap format type. If the translation is successful, the Scrap Manager returns to your application the data from the scrap in the requested scrap format type.

Scrap Manager

Whenever possible, your application should write both of the standard data types to the scrap. For example, a graphics application, such as SurfPaint, can choose to write both 'PICT' and 'TEXT' formats to the scrap when the user copies a picture consisting of text. Figure 2-2 shows a SurfPaint document and a SurfWriter document. The user copies, then pastes, a picture consisting of text. The SurfPaint application can choose to write only the 'PICT' format; if it does so, then SurfWriter reads the data from the scrap in 'PICT' format and inserts the data as a picture in the SurfWriter document. If the SurfPaint document writes both 'PICT' and 'TEXT' formats to the scrap, SurfWriter can choose which format to read. In this case, SurfWriter can choose to read the 'TEXT' format of the data and insert the data as editable text into the document.

Figure 2-2 Writing both standard formats to the scrap

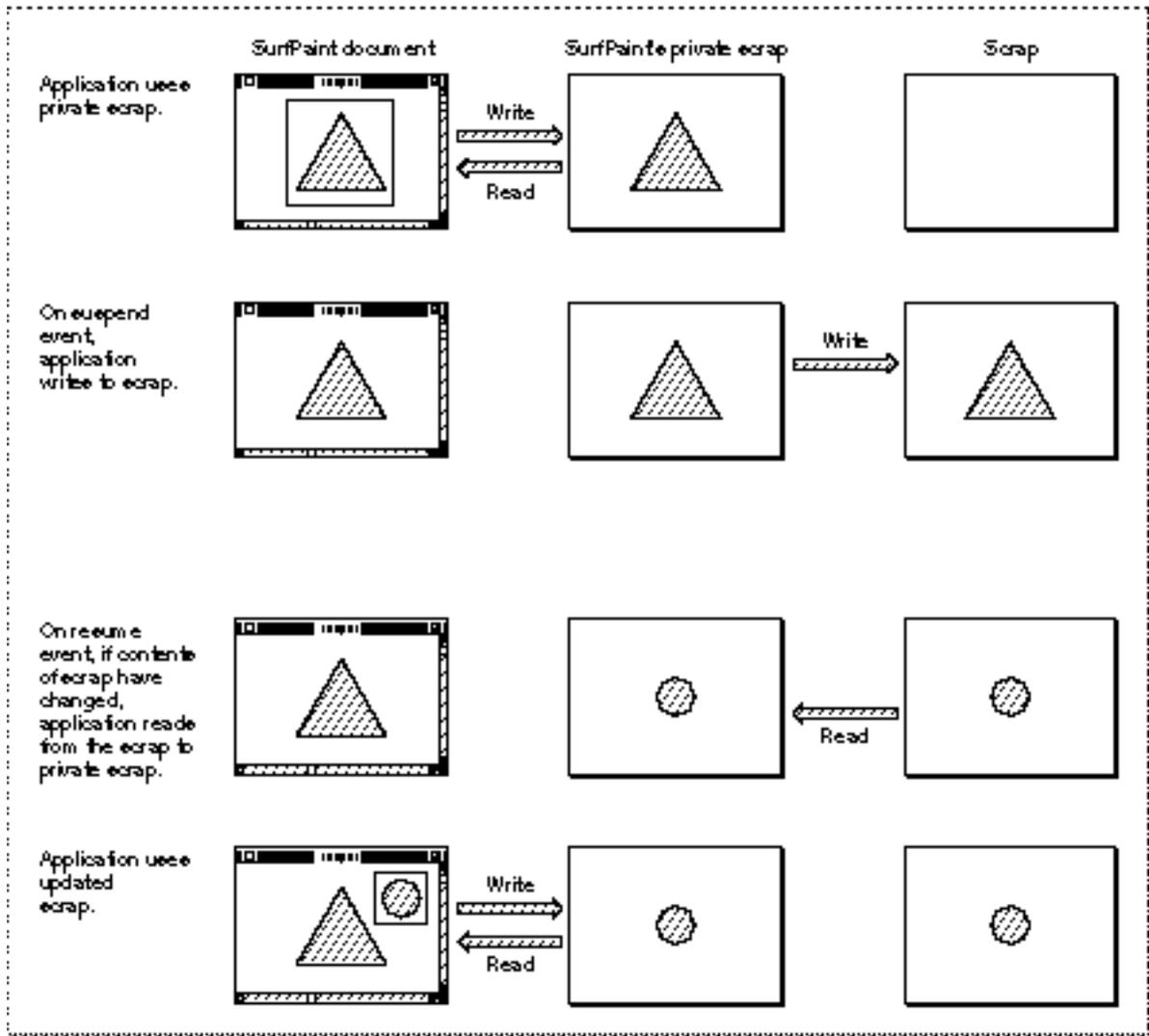


The SurfPaint application uses an application-defined data type to describe the data in its documents. It uses this same format in its private scrap; this implementation works well as long as the user is working exclusively with SurfPaint documents. When the SurfPaint application receives a suspend event, indicating that another application is about to become the foreground process, SurfPaint copies the data from its private scrap to the scrap. SurfPaint writes data to the scrap in its own private format ('SFPN'), in 'PICT' format, and if the picture contains text, it writes the data to the scrap in 'TEXT'

Scrap Manager

format as well. Upon receiving a resume event, SurfPaint determines whether the contents of the scrap have changed and if so, copies the new data from the scrap into its private scrap. Figure 2-3 shows how the SurfPaint application uses its own private scrap.

Figure 2-3 Using a private scrap



Note that when your application receives a resume event, it should determine whether the contents of the scrap have changed. If your application uses a private scrap, either you can choose to copy the data from the scrap to your private scrap immediately or you can delay copying until the data is needed.

Scrap Manager

If your application writes data to the scrap in more than one format, it should write the data in its order of preference. For example, the SurfPaint application writes its preferred scrap format type first (its own private format, 'SFPN'), then writes the data in 'PICT' format, and then, if appropriate, writes the data in 'TEXT' format. However, the size of the scrap is limited; therefore, when your application needs to write a large amount of data to the scrap and there isn't enough room in the scrap for both your application's private scrap format type and one of the standard formats, write the data in the standard format.

As previously described, the Scrap Manager uses the Translation Manager (if it's available) to convert data in one scrap format type into another. If your application writes its own private scrap format type to the scrap, you may want to provide one or more translators that translate your private scrap format type into other format types. See the chapter "Translation Manager" in this book for information on how to write translators.

The Clipboard

The Clipboard refers to what the user views as residing in the scrap. Your application can provide a Show Clipboard/Hide Clipboard command to show or hide a window, referred to as the Clipboard window. When the user chooses this command, your application should display in its Clipboard window the current contents of the scrap. Although multiple scrap format types may reside in the scrap, applications that support a Clipboard window typically display the data in only one format.

If your application provides this command, your application should hide its Clipboard window (if it's showing) whenever it receives a suspend event. It can show the Clipboard window again when it receives a resume event.

Intelligent Cut and Paste

When the user selects text and then chooses the Cut command, or sets the insertion point and then chooses Paste, your application should apply "intelligent cut and paste," that is, discard extra spaces or add spaces, as outlined here. In general, your application should follow these rules to provide intelligent cut and paste:

- n If the user selects a word or range of words, highlight the selection but not any adjacent spaces.
- n When the user chooses the Cut command, if the character to the left of the selection is a space, discard it. Otherwise, if the character to the right of the selection is a space, discard it.
- n When the user chooses the Paste command, if the character to the left or right of the current selection or if the character to the left or right of the insertion point is part of a word, insert a space before pasting the text.

Figure 2-4 shows examples of intelligent cut and paste.

Figure 2-4 Intelligent cut and paste

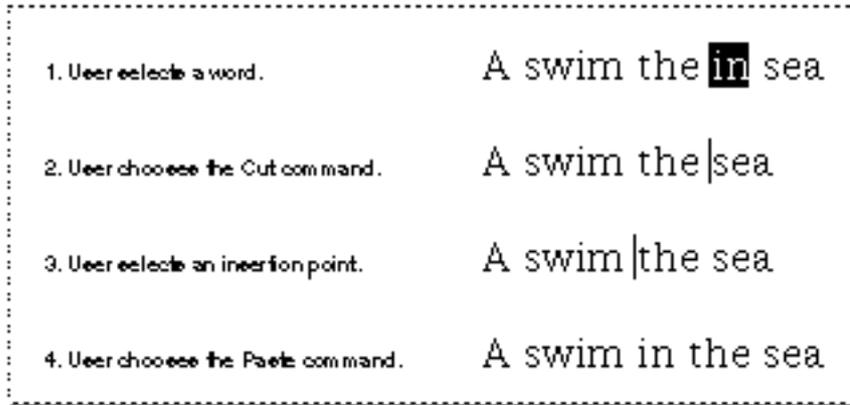
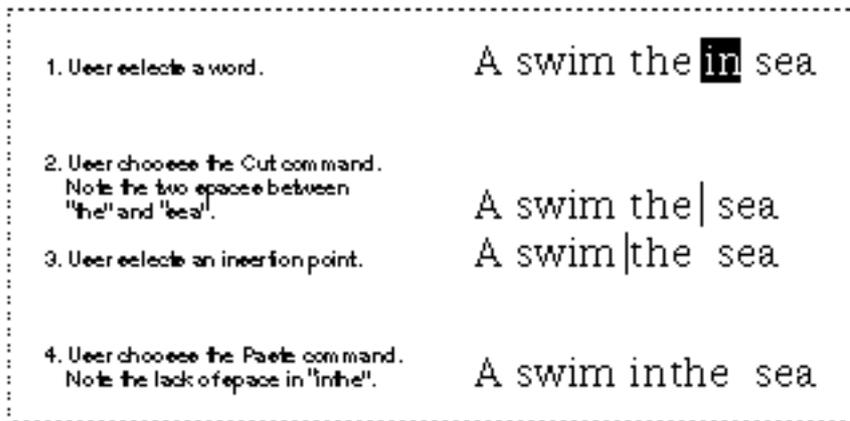


Figure 2-5 shows the results of applying the same operations in an application that doesn't support intelligent cut and paste.

Figure 2-5 Non-intelligent cut and paste



See *Macintosh Human Interface Guidelines* for details of selection techniques and guidelines for selecting words and paragraphs.

About the Scrap Manager

You can use the Scrap Manager to support copying and pasting of data. If your application uses TextEdit (in its windows or dialog boxes), be aware that TextEdit also maintains its own private scrap. You use TextEdit routines to copy data from the TextEdit scrap (if any) to the scrap. See “Converting Data Between the TextEdit Scrap and the Scrap” beginning on page 2-28 for information on TextEdit’s scrap.

The next section describes the location of the scrap. “Using the Scrap Manager” beginning on page 2-14 provides specific information on how you can use Scrap Manager routines in your application.

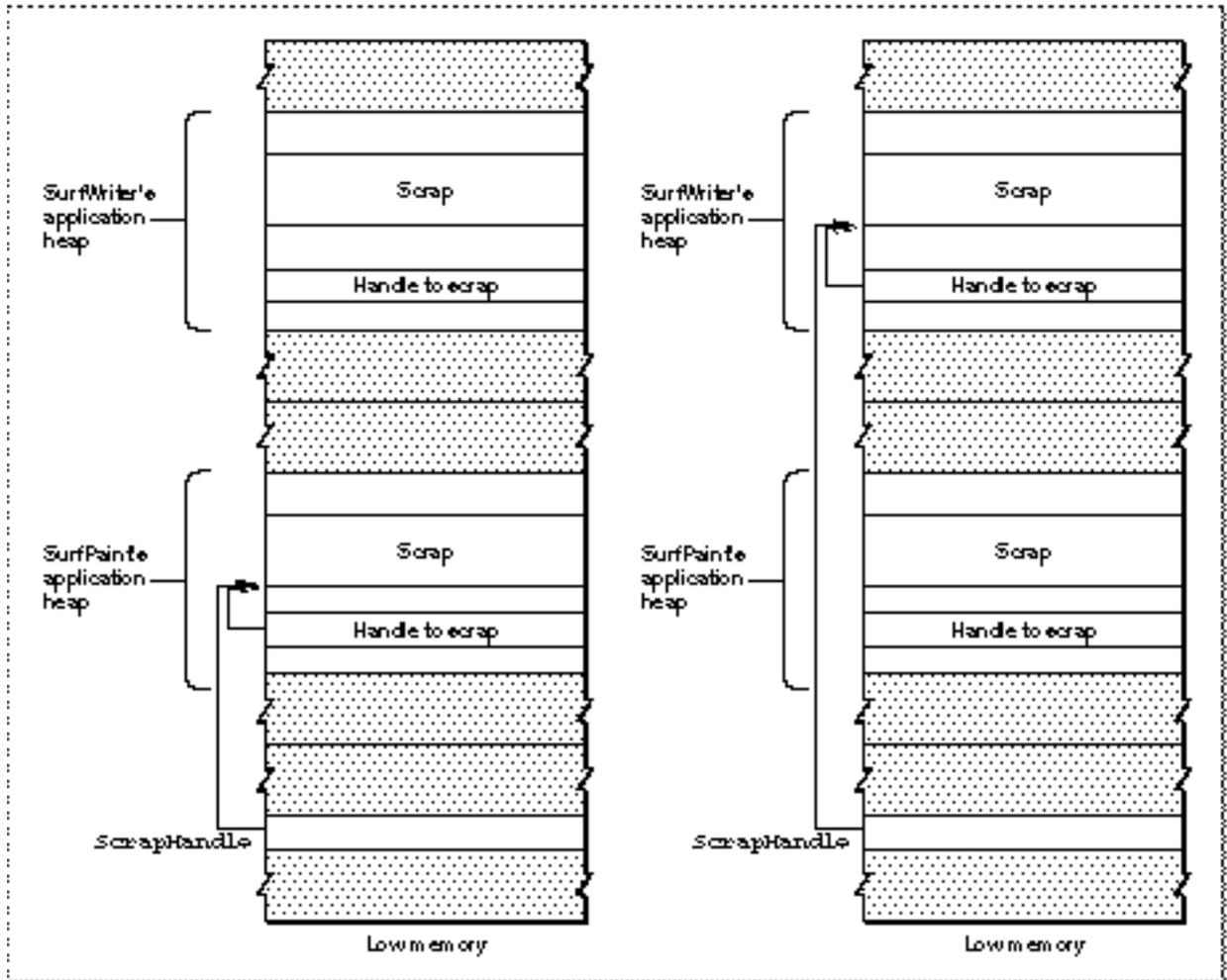
Location of the Scrap

System software allocates space in each application’s heap for the scrap and allocates a handle to reference the scrap. The system global variable `ScrapHandle` contains a handle to the scrap of the current process. When system software launches an application, it copies the data from the scrap of the previously active application into the application heap of the newly active application. If the scrap is too large to fit in the application’s application heap, system software copies the scrap to disk and sets the value of the handle to the scrap in the application heap to `NIL` to indicate that the scrap is on disk.

Figure 2-6 shows two applications (SurfWriter and SurfPaint) that are in memory and shows the handles and allocated space for the scrap in each application’s heap. In this example, SurfPaint was the previously active application and the user switches to the SurfWriter application. At this moment, the system global variable `ScrapHandle` references the scrap in SurfPaint’s application heap. SurfPaint’s application heap contains a handle to the scrap in its application heap.

System software sends SurfPaint a suspend event to begin the switch to the SurfWriter application. Because SurfPaint uses a private scrap, upon receiving the suspend event it copies data from its private scrap to the scrap. After SurfPaint responds to the suspend event, system software copies the data from the scrap in SurfPaint’s application heap to SurfWriter’s application heap, resizing the scrap in SurfWriter’s application heap as necessary. System software sets the handle in SurfWriter’s application heap to reference the new scrap and sets the system global variable `ScrapHandle` to reference the scrap in SurfWriter’s application heap. System software sends SurfWriter a resume event and sets the `convertClipboardFlag` bit in the message field of the event record. System software sets this bit when the contents of the scrap have changed since the previous suspend event, indicating to the application that it should copy the scrap to its private scrap.

Figure 2-6 Location of the scrap in memory



You can get the size of the scrap and a handle to the scrap in your application's heap by calling the `InfoScrap` function.

Although the scrap is usually located in memory, your application can write the contents of the scrap in memory to a scrap file using the `UnloadScrap` function. After writing the contents of the scrap to disk, the `UnloadScrap` function releases the memory previously occupied by the scrap in your application's heap; thereafter, any operations your application performs on data in the scrap affect the scrap as stored in the scrap file on disk.

You can use the `LoadScrap` function to read the contents of the scrap file back into memory. The `LoadScrap` function allocates memory in your application's heap for the scrap and reads the contents of the scrap on disk into memory; thereafter, any operations your application performs on data in the scrap affect the scrap in memory.

Scrap Manager

The Scrap Manager keeps track of whether the scrap is in memory or on disk and always reads data from and writes data to the scrap's current location. As a result, your application seldom needs to know the location of the scrap. Your application should use the `UnloadScrap` function only if the scrap in memory isn't large enough to hold the data you need to write to the scrap.

If your application transfers the scrap from memory to disk and is then switched to the background, system software reads the scrap from disk into the newly active application's heap. When your application returns to the foreground, system software writes the scrap from the previous application's application heap back to disk.

Using the Scrap Manager

This section explains how you can use the Scrap Manager to support copy-and-paste operations in your application. In particular, this section explains how you can

- n get information about the current contents of the scrap
- n handle the Cut and Copy commands
- n respond to suspend events
- n handle the Paste command
- n respond to resume events
- n use `TextEdit` to support the editing commands
- n support copying and pasting of data in dialog boxes

The Scrap Manager uses the services of the Translation Manager (if it's available). To determine whether the Scrap Manager can use the Translation Manager, call the `Gestalt` function with the `gestaltScrapMgrAttr` selector and check the value of the `response` parameter. If the bit indicated by the constant `gestaltScrapMgrTranslationAware` is set, then the Scrap Manager uses the Translation Manager when needed to translate scrap format types.

```
CONST
gestaltScrapMgrAttr          = 'scra'; {Gestalt selector for }
                                { Scrap Mgr attributes }
gestaltScrapMgrTranslationAware = 0;   {check this bit in the }
                                { response parameter }
```

Getting Information About the Scrap

To get information about the scrap, you can use the `InfoScrap` function. The `InfoScrap` function returns a pointer to a scrap information record. (See “The Scrap Information Record” on page 2-32 for detailed information on the fields of this record.) The information in the scrap information record provides

- n the size (in bytes) of the scrap
- n a handle to the scrap if it’s in memory
- n a count, or number that your application can use to determine whether the contents of the scrap have changed
- n the location of the scrap (whether in memory or on disk)
- n the filename of the scrap when it is on the disk

For example, this code uses the `InfoScrap` function to get the size of the scrap.

```
VAR
    curScrapInfoPtr: PScrapStuff;
    curScrapSize:    LongInt;

    curScrapInfoPtr := InfoScrap;
    curScrapSize := curScrapInfoPtr^.scrapSize;
```

Putting Data in the Scrap

Your application should write data to the scrap (or to its own private scrap) whenever the user chooses the Cut or Copy command and the document the user is working with contains a selection. In addition, if your application uses a private scrap, your application must copy the contents of its private scrap to the scrap upon receiving a suspend event. The next sections explain how to perform these tasks.

Handling the Cut Command

When the user chooses the Cut command and the document the user is working with contains a selection, your application should remove the data from the selection and save the data (either in the scrap or in your application’s private scrap).

Scrap Manager

The SurfWriter application doesn't use a private scrap; whenever the user performs a cut operation, SurfWriter writes the current selection to the scrap. The SurfWriter application does define its own private scrap format type and writes this format to the scrap, along with one of the standard scrap formats. Listing 2-1 shows SurfWriter's routine for handling the Cut command (it also uses this routine for the Copy command).

Listing 2-1 Writing data to the scrap

```

PROCEDURE DoCutOrCopyCmd (cut: Boolean);
VAR
    window:           WindowPtr;
    windowType:       Integer;
    isText:           Boolean;
    ptrToScrapData:   Ptr;
    length, myLongErr: LongInt;
BEGIN
    window := FrontWindow;
    windowType := MyGetWindowType(window);
    IF windowType = kMyDocWindow THEN
        BEGIN
            ptrToScrapData := NewPtr(kDefaultSize);
            isText := MyIsSelectionText;
            IF isText THEN {selection contains text}
                BEGIN
                    MyGetSelection('SURF', ptrToScrapData, length);
                    myLongErr := ZeroScrap;
                    myLongErr := PutScrap(length, 'SURF', ptrToScrapData);
                    IF myLongErr <> noErr THEN DoError(myLongErr);
                    MyGetSelection('TEXT', ptrToScrapData, length);
                    myLongErr := PutScrap(length, 'TEXT', ptrToScrapData);
                    IF myLongErr <> noErr THEN DoError(myLongErr);
                END
            ELSE {selection contains graphics}
                BEGIN
                    MyGetSelection('PICT', ptrToScrapData, length);
                    myLongErr := ZeroScrap;
                    myLongErr := PutScrap(length, 'PICT', ptrToScrapData);
                    IF myLongErr <> noErr THEN DoError(myLongErr);
                END;
            DisposePtr(ptrToScrapData);
            IF cut THEN
                MyDeleteSelection;
        END
    END

```

Scrap Manager

```

ELSE
  IF windowType <> kNIL THEN
    BEGIN      {window is a dialog box}
      IF cut THEN
        DialogCut(window)
      ELSE
        DialogCopy(window);
    END;
  END;
END;

```

The `DoCutOrCopyCmd` procedure first determines the type of window that is frontmost. If the frontmost window is a document window, `DoCutOrCopyCmd` uses another application-defined routine, `MyIsSelectionText`, to determine whether the current selection contains text or graphics. If the selection contains only text, `SurfWriter` writes the data to the scrap using two formats: its own private format ('SURF') and the standard format 'TEXT'. The `DoCutOrCopyCmd` procedure uses another application-defined routine, `MyGetSelection`, to return the current selection in a particular format. `DoCutOrCopyCmd` then calls the `ZeroScrap` function to clear the contents of the scrap. After calling `ZeroScrap`, `DoCutOrCopyCmd` calls `PutScrap`, specifying the length of the data, a pointer to the data, and identifying the scrap format type as 'SURF'. `DoCutOrCopyCmd` then uses the `MyGetSelection` routine again, this time to return the current selection in the 'TEXT' format type. `DoCutOrCopyCmd` calls `PutScrap` to write the data to the scrap, specifying a pointer to the data and identifying the scrap format type as 'TEXT'.

If the selection contains a picture, `DoCutOrCopyCmd` uses the `MyGetSelection` routine to return the current selection using the 'PICT' format type. After calling `ZeroScrap`, `DoCutOrCopyCmd` calls `PutScrap` to write the data to the scrap, specifying a pointer to the data and identifying the scrap format type as 'PICT'.

Finally, if `DoCutOrCopyCmd` was called as a result of the user performing a cut operation, `DoCutOrCopyCmd` deletes the selection from the current document.

If the frontmost window is a dialog box, `DoCutOrCopyCmd` uses the Dialog Manager's `DialogCut` (or `DialogCopy`) procedure to write the selected data to the scrap.

Note that you should always call `ZeroScrap` before writing data to the scrap. If you write multiple formats to the scrap, call `ZeroScrap` once before you write the first format to the scrap.

You should always write data to the scrap in your application's preferred order of formats. For example, `SurfWriter`'s preferred format for text data is its own private format ('SURF'), so it writes that format first and then writes the standard format 'TEXT'.

If your application uses `TextEdit` in its document windows, then use the `TextEdit` routine `TECut` (or `TECopy`) instead of `ZeroScrap` and `PutScrap`. See Listing 2-8 on page 2-29 for an application-defined routine that uses `TextEdit` routines to help handle the Cut and Copy commands.

Scrap Manager

If your application uses a private scrap, then copy the selected data to your private scrap rather than to the scrap. For example, the SurfPaint application uses a private scrap. Listing 2-2 shows SurfPaint's application-defined routine that handles the Cut command by writing the selected data to its private scrap.

Listing 2-2 Writing data to a private scrap

```
PROCEDURE DoCutOrCopyCmd (cut: Boolean);
VAR
    window:           WindowPtr;
    windowType:       Integer;
BEGIN
    window := FrontWindow;
    windowType := MyGetWindowType(window);
    IF windowType = kMyDocWindow THEN
        BEGIN
            MyWriteDataToPrivateScrap;
            {reset gScrapNewData to indicate that this app's private }
            { scrap now contains the most recent data}
            IF gScrapNewData THEN
                gScrapNewData := FALSE;
            IF cut THEN
                MyDeleteSelection;
        END
    ELSE
        IF windowType <> kNil THEN
            BEGIN {window is a dialog window}
                IF cut THEN
                    DialogCut(window)
                ELSE
                    DialogCopy(window);
            END;
        END;
END;
```

The application-defined `DoCutOrCopyCmd` procedure shown in Listing 2-2 calls another application-defined procedure, `MyWriteDataToPrivateScrap`, to write the data in the current selection to the application's private scrap. SurfPaint uses the application-defined global variable `gScrapNewData` to indicate when data should be read from the scrap instead of its own private scrap as a result of the user choosing the Paste command. Upon receiving a resume event, if the contents of the scrap have changed, SurfPaint sets the `gScrapNewData` global variable to `TRUE`. If the user chooses Paste and `gScrapNewData` is `TRUE`, SurfPaint reads the scrap to get the data to paste; otherwise SurfPaint reads its own private scrap to get the data to paste.

Scrap Manager

If the user chooses Cut or Copy before the next Paste command, SurfPaint writes the newly selected data to its private scrap, eliminating the need to read the previous contents of the scrap, and thus the `DoCutOrCopyCmd` procedure resets the `gScrapNewData` global variable to `FALSE`.

Handling the Copy Command

When the user chooses the Copy command and the document the user is working with contains a selection, your application should copy the selected data (without deleting it) and save the copied data (either in the scrap or in your application's private scrap). See Listing 2-1 on page 2-16, Listing 2-2 on page 2-18, and Listing 2-8 on page 2-29 for application-defined routines that handle the Copy command.

Handling Suspend Events

As previously described, if your application uses a private scrap, your application must copy the contents of its private scrap to the scrap upon receiving a suspend event. In addition, if your application supports the Show Clipboard command, it should hide the Clipboard window if it's currently showing (because the contents of the scrap may change while your application yields time to another application).

Listing 2-3 shows SurfPaint's routine that responds to suspend events (and resume events).

Listing 2-3 Copying data from the scrap in response to suspend events

```
PROCEDURE DoSuspendResumeEvent (event: EventRecord);
VAR
    currentFrontWindow: WindowPtr;
BEGIN
    currentFrontWindow := FrontWindow;
    IF (BAnd(event.message, resumeFlag) <> 0) THEN
    BEGIN
        {it's a resume event; }
    END
    { handle as shown in Listing 2-6}
    ELSE
    BEGIN
        {it's a suspend event}
        {copy private scrap to the scrap}
        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;
```

Scrap Manager

Listing 2-3 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. See Listing 2-6 on page 2-25 for details on handling resume events.

For suspend events, the `DoSuspendResumeEvent` procedure calls the application-defined `MyConvertScrap` procedure to copy the contents of its private scrap to the scrap. (See Listing 2-7 on page 2-27 for the `MyConvertScrap` procedure.) It then sets the private global flag `gInBackground` to `TRUE` to indicate that the application is in the background. It calls another application-defined routine to deactivate the application's front window. It also calls the application-defined routine `MyHideClipboardWindow` to hide the Clipboard window if it's currently showing.

Getting Data From the Scrap

Your application should read data from the scrap (or from its own private scrap) whenever the user chooses the Paste command. In addition, if your application uses a private scrap, upon receiving a resume event your application should determine whether the contents of the scrap have changed since the previous resume event, and if so, it should take the appropriate actions. The next sections explain how to perform these tasks.

Handling the Paste Command

When the user chooses the Paste command, your application should paste the data last cut or copied by the user. You should insert the new data at the current insertion point or, if a selection exists, replace the selection with the new data. You get the data to paste by reading the data from the scrap or from your application's private scrap.

When you read data from the scrap, your application should request the data in its preferred scrap format type. If that type of format doesn't exist in the scrap, then request the data in another format. For example, SurfWriter's preferred format type is `'surf'`, so it requests data from the scrap in this format. If this format isn't in the scrap, SurfWriter requests its next preferred type, `'TEXT'`. Finally, if the `'TEXT'` format isn't in the scrap, SurfWriter requests the data in the `'PICT'` format.

If your application doesn't have a preferred scrap format type, then read from the scrap each format type your application supports. Along with a pointer to the data of the requested format type, the `GetScrap` function returns an offset, a value that indicates the relative offset of the start of that format of data in the scrap. (Note that the returned value for the offset is valid only if the Translation Manager isn't available; if the Translation Manager is available, then your application should not rely on the offset value.) The format type with the lowest offset is the preferred format type of the application that put the data in the scrap; thus a format with a lower offset is more likely to contain more information than formats in the scrap with higher offsets. So when the Translation Manager isn't available, use the format with the lowest offset when your application doesn't have a particular scrap format that it prefers.

Scrap Manager

If you request a scrap format type that isn't in the scrap and the Translation Manager is available, the Scrap Manager uses the Translation Manager to convert any one of the scrap format types currently in the scrap into the scrap format type requested by your application. The Translation Manager looks for a translator that can perform one of these translations. If such a translator is available (for example, a translator that can translate the 'SDBS' scrap format type into the 'SURF' scrap format type), the Translation Manager uses the translator to translate the data in the scrap into the requested scrap format type. If the translation is successful, the Scrap Manager returns to your application the data from the scrap in the requested scrap format type.

Listing 2-4 shows SurfWriter's routine for handling the Paste command. The SurfWriter application doesn't use a private scrap; whenever the user performs a paste operation, SurfWriter reads the data that is to be pasted from the scrap.

For document windows, the SurfWriter application first determines whether the data in the scrap exists in its own private scrap format ('SURF') by using the `GetScrap` function. If you specify a NIL handle as the location to return the data, `GetScrap` does not return the data but does return as its function result the number of bytes (if any) of data of the specified format that exists in the scrap. If data of this format does exist, SurfWriter reads the data in this format. SurfWriter allocates the handle to hold any returned data but does not need to size the handle; `GetScrap` automatically resizes the handle passed to it to the required size to hold the retrieved data. Once the data is retrieved in 'SURF' format, SurfWriter pastes the data into the current document.

If the scrap does not contain data in 'SURF' format (and the available translators can't convert any of the scrap format types in the scrap to the 'SURF' format), SurfWriter determines whether any data in 'TEXT' format exists in the scrap. If so, SurfWriter uses `GetScrap` to retrieve the data. Once the data is retrieved in 'TEXT' format, SurfWriter pastes the data into the current document.

If the scrap does not contain data in 'TEXT' format, SurfWriter determines whether any data in 'PICT' format exists in the scrap. If so, SurfWriter uses `GetScrap` to retrieve the data. Once the data is retrieved in 'PICT' format, SurfWriter determines the destination rectangle, that is, the rectangle where the picture should be displayed, then uses the `QuickDraw DrawPicture` procedure to draw the picture in the window. SurfWriter stores a handle to this picture and sets other application-defined variables as needed.

Listing 2-4 Handling the Paste command using the scrap

```
PROCEDURE DoPasteCommand;
VAR
    window:           windowPtr;
    windowType:      LongInt;
    offset:           LongInt;
    sizeOfSurfData:   LongInt;
    sizeOfPictData:   LongInt;
    sizeOfTextData:   LongInt;
```

Scrap Manager

```

hDest:           Handle;
myData:          MyDocRecHnd;
teHand:          TEHandle;
destRect:        Rect;
myErr:           OSErr;
BEGIN
  window := FrontWindow;
  windowType := MyGetWindowType(window);
  IF windowType = kMyDocWindow THEN
  BEGIN {handle Paste command in document window. Check }
    { whether the scrap contains any data. This app }
    { checks for its preferred format type, 'SURF', first}
    sizeofSurfData := GetScrap(NIL, 'SURF', offset);
    IF sizeofSurfData > 0 THEN
    BEGIN
      {allocate handle to hold data from scrap--GetScrap }
      hDest := NewHandle(0); { automatically resizes it}
      HLock(hDest);
      {put data into memory referenced thru hDest handle}
      sizeofSurfData := GetScrap(hDest, 'SURF', offset);
      {paste the data into the current document}
      MyPasteSurfData(hDest);
      HUnlock(hDest);
      DisposeHandle(hDest);
    END
  ELSE
  BEGIN {if no 'SURF' data in scrap, check for 'TEXT'}
    sizeofTextData := GetScrap(NIL, 'TEXT', offset);
    IF sizeofTextData > 0 THEN
    BEGIN
      {allocate handle to hold data from scrap--GetScrap }
      hDest := NewHandle(0); { automatically resizes it}
      HLock(hDest);
      {put data into memory referenced thru hDest handle}
      sizeofTextData := GetScrap(hDest, 'TEXT', offset);
      {paste the text into the current document}
      MyPasteText(hDest);
      HUnlock(hDest);
      DisposeHandle(hDest);
    END
  ELSE {if no 'TEXT' data in scrap, check for 'PICT'}
  BEGIN
    sizeofPictData := GetScrap(NIL, 'PICT', offset);

```

Scrap Manager

```

IF sizeOfPictData > 0 THEN
BEGIN
    {allocate handle to hold scrap data--GetScrap }
    hDest := NewHandle(0); { automatically resizes it}
    HLock(hDest);
    {put data into memory referenced thru hDest handle}
    sizeOfPictData := GetScrap(hDest, 'PICT', offset);
    {calculate destination rectangle for plotting the }
    { picture}
    MyGetDestRect(hDest, destRect);
    DrawPicture(PicHandle(hDest), destRect);
    {save information about the picture}
    myData := MyDocRecHnd(GetWRefCon(window));
    myData^^.pictNum := myData^^.pictNum + 1;
    myData^^.pictDestRect[myData^^.pictNum] :=
                                                destRect;
    IF myData^^.windowPicHndl[myData^^.pictNum] = NIL
    THEN
        myData^^.windowPicHndl[myData^^.pictNum] :=
            PicHandle(NewHandle(Size(sizeOfPictData)));
    myData^^.windowPicHndl[myData^^.pictNum] :=
                                                PicHandle(hDest);
    myErr := HandToHand(Handle
        (myData^^.windowPicHndl[myData^^.pictNum]));
    HUnlock(hDest);
    DisposeHandle(hDest);
    END;      {of sizeOfPictData > 0}
    END;      {of "if no 'TEXT' data, check for 'PICT'"}
    END;      {of "if no 'surf' data, check for 'TEXT'"}
END          {of "if windowType = kMyDocWindow"}
ELSE        {window is not a document window}
BEGIN
    IF windowType <> kNil THEN
    BEGIN {handle Paste command in dialog box, }
        { DialogPaste checks whether the dialog box has any }
        { editText items and if so, uses TEPaste to paste }
        { any text from the scrap to the currently selected }
        { editText item, if any}
        DialogPaste(window);
    END;
    END;
END;

```

Scrap Manager

If your application uses `TextEdit` in its document windows, then use the `TextEdit` routine `TEPaste` instead of `GetScrap` to read the data to paste. See Listing 2-9 on page 2-30 for an application-defined routine that uses `TextEdit` to help handle the application's Paste command.

If your application uses a private scrap, then read the data from your private scrap rather than from the scrap (unless the scrap contains the more recent data). Listing 2-5 shows `SurfPaint`'s application-defined routine that handles the Paste command by reading the desired data from its private scrap.

Listing 2-5 Handling the Paste command using a private scrap

```

PROCEDURE DoPasteCmd;
VAR
    window:           WindowPtr;
    windowType:       Integer;
    dataToPaste:      Ptr;
BEGIN
    window := FrontWindow;
    windowType := MyGetWindowType(window);
    IF windowType = kMyDocWindow THEN
    BEGIN
        IF gNewScrap THEN      {if new data in scrap, }
        BEGIN                  { copy to private scrap}
            MyConvertScrap(kClipboardToPrivate);
            gNewScrap := FALSE;
        END;
        {get the data to paste from app's private scrap}
        dataToPaste := NewPtr(kDefaultSize);
        MyReadDataFromPrivateScrap(dataToPaste);
        MyPasteData(dataToPaste);
        DisposePtr(dataToPaste);
    END
    ELSE
    IF windowType <> kNil THEN
    BEGIN {window is a dialog box}
        DialogPaste(window);
    END;
END;

```

The `SurfPaint` application uses a private scrap, and when it receives a resume event, it determines whether the contents of the scrap have changed. If so, `SurfPaint` sets an application-defined global variable, `gScrapNewData`, but does not immediately read in the contents of the scrap. Instead, whenever the user chooses the Paste command, `SurfPaint` checks the value of this global variable. If `gScrapNewData` is `TRUE` `SurfPaint`

Scrap Manager

reads the new data from the scrap to its private scrap, resets the `gScrapNewData` global variable to `FALSE`, and then performs the paste operation. `SurfPaint` also resets the value of the `gScrapNewData` global variable to `FALSE` whenever the user chooses the Cut or Copy command. By using this method, `SurfPaint` reads in new data from the scrap only when necessary and avoids reading in data that the user might not use. This method also decreases the time it takes for the application to return to the foreground, as the application avoids or delays any lengthy translation of data from the scrap.

Handling Resume Events

As previously described, when your application receives a resume event (and your application uses a private scrap), your application should determine whether the contents of the scrap have changed since the previous suspend event. If the contents of the scrap have changed, your application must be sure to use the new data in the scrap for the user's next Paste command (unless the user chooses Cut or Copy before choosing Paste).

In addition, if your application supports the Show Clipboard command and the Clipboard window was showing at the time of the previous suspend event, your application should update its Clipboard window to show the new contents of the scrap.

Listing 2-6 shows `SurfPaint`'s procedure for handling resume events (and suspend events).

Listing 2-6 Handling resume events

```
PROCEDURE DoSuspendResumeEvent (event: EventRecord);
VAR
    currentFrontWindow: WindowPtr;
BEGIN
    currentFrontWindow := FrontWindow;
    IF (BAnd(event.message, resumeFlag) <> 0) THEN
    BEGIN
        {it's a resume event}
        IF (BAnd(event.message, convertClipboardFlag) <> 0) THEN
        BEGIN
            {set flag to indicate there's new data in the scrap}
            gNewScrap := TRUE;
        END;
        gInBackground := FALSE; {app no longer in background}
                                {activate front window}
        DoActivate(currentFrontWindow, NOT gInBackground, event);
        {show Clipboard window if it was showing at last suspend }
        { event and update its contents to match scrap}
        MyShowClipboardWindow(gNewScrap);
        MyShowFloatingWindows; {show any floating windows}
    END
END
```

Scrap Manager

```

ELSE
BEGIN
    {it's a suspend event, }
    { handle as shown in Listing 2-3}

END;
END;

```

Listing 2-6 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 sets an application-defined global variable, `gNewScrap`, to indicate that new data exists in the scrap. When the user next chooses the Paste command, `SurfPaint` checks the value of the `gNewScrap` global variable and, if it's `TRUE`, reads the data from the scrap to its private scrap and then performs the paste operation. If the user chooses the Cut or Copy command before choosing Paste, then `SurfPaint` resets the `gNewScrap` global variable to `FALSE` to indicate that its private scrap contains the most recent data for the Paste command. This technique allows `SurfPaint` to delay or avoid any lengthy translation of data from the scrap to its private scrap and decreases the time it takes for `SurfPaint` to return to the foreground.

The `DoSuspendResumeEvent` procedure 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`, to activate the application's front window. It also calls the application-defined routine `MyShowClipboardWindow` to show the Clipboard window and update its contents if it was showing at the time of the previous suspend event.

Converting Data Between a Private Scrap and the Scrap

If you use a private scrap, you need to copy the data from your private scrap to the scrap upon receiving a suspend event. Upon receiving a resume event, you need to determine whether the contents of the scrap have changed since the previous suspend event. If so, your application must be sure to use the new data in the scrap for the user's next Paste command (unless the user chooses Cut or Copy before choosing Paste). In addition, your application needs to update the contents of its Clipboard window, if necessary.

Listing 2-7 shows the application-defined procedure `MyConvertScrap`. This procedure is called either indirectly as a result of a resume event (indicated by the `kClipboardToPrivate`, constant) or directly as a result of a suspend event (indicated by the `kPrivateToClipboard` constant). If the `whichWay` parameter contains `kClipboardToPrivate`, then the contents of the scrap have changed. In this case, `MyConvertScrap` uses `GetScrap` to read the contents of the scrap. The `MyConvertScrap` procedure checks the scrap for 'PICT' data first, and then for 'TEXT' data if the scrap doesn't contain any data in 'PICT' format. `MyConvertScrap` then copies this data to its private scrap.

Scrap Manager

If the `MyConvertScrap` procedure is called as a result of a suspend event, the procedure copies the data from its private scrap to the scrap. It writes the data to the scrap in its own private format, in 'PICT' format, and, if appropriate, in 'TEXT' format.

Listing 2-7 Converting data between the scrap and a private scrap

```

PROCEDURE MyConvertScrap (whichWay: Integer);
VAR
    sizeOfTextData:   LongInt;
    sizeOfPictData:  LongInt;
    offset:           LongInt;
    hDest:            Handle;
    ptrToScrapData:  Ptr;
    length:           LongInt;
    myLongErr:       LongInt;
BEGIN
    IF whichWay = kClipboardToPrivate THEN
    BEGIN {copy scrap to private scrap}
        sizeOfPictData := GetScrap(NIL, 'PICT', offset);
        IF sizeOfPictData > 0 THEN
        BEGIN
            {get handle to hold data from scrap, GetScrap }
            hDest := NewHandle(0); { automatically resizes it}
            HLock(hDest);
            {put data into memory referenced by hDest handle}
            sizeOfPictData := GetScrap(hDest, 'PICT', offset);
            MyCopyToPrivateScrap(hDest);
            HUnlock(hDest);
            DisposeHandle(hDest);
        END
    ELSE {if no 'PICT' data on scrap, check for 'TEXT'}
    BEGIN
        sizeOfTextData := GetScrap(NIL, 'TEXT', offset);
        IF sizeOfTextData > 0 THEN
        BEGIN
            {allocate handle to hold scrap data--GetScrap }
            hDest := NewHandle(0); { automatically resizes it}
            HLock(hDest);
            {put data into memory referenced by hDest handle}
            sizeOfTextData := GetScrap(hDest, 'TEXT', offset);
            {copy data to private scrap}
            MyCopyToPrivateScrap(hDest);
            HUnlock(hDest);
        END
    END
END

```

Scrap Manager

```

        DisposeHandle(hDest);
    END
END;
END
ELSE
BEGIN {copy private scrap into scrap}
    IF MyGetPrivateScrapSize > 0 THEN {if private scrap }
        myLongErr := ZeroScrap; { not empty, clear the scrap}
        ptrToScrapData := NewPtr(kDefaultSize);
        {retrieve data from private scrap in private format}
        IF (MyGetScrap('SURF', ptrToScrapData, length) > 0) THEN
        BEGIN {copy data to the scrap}
            myLongErr := PutScrap(length, 'SURF', ptrToScrapData);
            IF myLongErr <> noErr THEN DoError(myLongErr);
        END;
        {retrieve data from private scrap in 'PICT' format}
        IF (MyGetScrap('PICT', ptrToScrapData, length) > 0) THEN
        BEGIN {copy data to the scrap}
            myLongErr := PutScrap(length, 'PICT', ptrToScrapData);
            IF myLongErr <> noErr THEN DoError(myLongErr);
        END;
        {retrieve data from private scrap in 'TEXT' format}
        IF (MyGetScrap('TEXT', ptrToScrapData, length) > 0) THEN
        BEGIN {copy data to the scrap}
            myLongErr := PutScrap(length, 'TEXT', ptrToScrapData);
            IF myLongErr <> noErr THEN DoError(myLongErr);
        END;
        DisposePtr(ptrToScrapData);
    END;
END;
END;

```

Converting Data Between the TextEdit Scrap and the Scrap

If your application uses TextEdit to handle text in its document windows, then use TextEdit routines instead of Scrap Manager routines to implement editing commands. For example, use the TextEdit procedures `TECut`, `TECopy`, and `TEPaste` to implement the Cut, Copy, and Paste commands. Upon receiving a suspend event, use `TEToScrap` instead of `PutScrap` to write the data to the scrap (always call `ZeroScrap` before calling `TEToScrap`). Upon receiving a resume event, use `TEFromScrap` instead of `GetScrap` to read data from the scrap. TextEdit uses a private scrap and handles copying data between its private scrap and the scrap. See *Inside Macintosh: Text* for complete information on TextEdit.

Scrap Manager

To implement the Cut (or Copy) commands, use the TextEdit routines `TECut` (or `TECopy`) instead of `ZeroScrap` and `PutScrap`. The TextEdit procedures `TECut` and `TECopy` copy the data in the current selection to TextEdit's private scrap. For example, Listing 2-8 shows an application-defined routine that uses TextEdit to help handle the application's Cut command (assuming the application uses TextEdit to handle text editing in its document windows).

Listing 2-8 Using TextEdit to handle the Cut command

```
PROCEDURE DoCutOrCopyCmd (cut: Boolean);
VAR
    window:           WindowPtr;
    windowType:       Integer;
    myData:            MyDocRecHnd;
    teHand:            TEHandle;
BEGIN
    window := FrontWindow;
    windowType := MyGetWindowType(window);
    IF windowType = kMyDocWindow THEN
        BEGIN
            myData := MyDocRecHnd(GetWRefCon(window));
            teHand := myData^^.editRec;
            IF cut THEN
                TECut(teHand)
            ELSE
                TECopy(teHand);
        END
    ELSE
        IF windowType <> kNIL THEN
            BEGIN {window is a dialog box}
                IF cut THEN
                    DialogCut(window)
                ELSE
                    DialogCopy(window);
            END;
        END;
    END;
```

Scrap Manager

Use the `TextEdit` routine `TEPaste` instead of `GetScrap` to read the data to paste. The `TEPaste` procedure reads the data to paste from `TextEdit`'s private scrap. Listing 2-9 shows an application-defined routine that uses `TextEdit` to help handle the application's `Paste` command (assuming the application uses `TextEdit` to handle text editing in its document windows).

Listing 2-9 Using `TextEdit` to handle the `Paste` command

```
PROCEDURE DoPasteCmd;
VAR
    window:           WindowPtr;
    windowType:       Integer;
    myData:           MyDocRecHnd;
    teHand:           TEHandle;
BEGIN
    window := FrontWindow;
    windowType := MyGetWindowType(window);
    IF windowType = kMyDocWindow THEN
    BEGIN
        myData := MyDocRecHnd(GetWRefCon(window));
        teHand := myData^^.editRec;
        TEPaste(teHand);
    END
    ELSE
    IF windowType <> kNIL THEN
    BEGIN          {window is a dialog box}
        DialogPaste(window);
    END;
END;
```

Upon receiving a suspend event, use `ZeroScrap` and then the `TextEdit` procedure `TEToScrap` to copy data from `TextEdit`'s private scrap to the scrap. Upon receiving a resume event, use the `TextEdit` procedure `TEFromScrap` to copy data from the scrap to `TextEdit`'s private scrap. As with any other private scrap and as explained in "Handling Resume Events" on page 2-25, either you can choose to immediately copy the data from the scrap to `TextEdit`'s private scrap or you can delay performing the copy until the data is needed. See Listing 2-5 on page 2-24 and Listing 2-6 on page 2-25 for code that uses this approach.

Handling Editing Operations in Dialog Boxes

You can use the Dialog Manager to handle most editing operations in dialog boxes. In general, use the procedures `DialogCut`, `DialogCopy`, and `DialogPaste` to support the Cut, Copy, and Paste commands in editable text items in your dialog boxes. As shown in Listing 2-2 on page 2-18 and Listing 2-5 on page 2-24, when the user chooses the Cut, Copy, or Paste command, the application-defined routine uses Dialog Manager routines to perform the editing operation.

The Dialog Manager uses `TextEdit` to perform the editing operation. `TextEdit` copies data between its private scrap and the editable text item in the dialog box. `TextEdit` uses a private scrap, which allows the user to copy and paste data between dialog boxes. However, your application must make sure the user can copy and paste data between your application's dialog boxes and its document windows. That is, when the user selects text in a document window and chooses Copy, then activates a dialog box and chooses Paste, the data previously copied from the document window should appear in the active editable text item. Your application is responsible for maintaining consistency between the scrap (or your application's private scrap) and `TextEdit`'s private scrap.

If your application uses `TextEdit` for all text editing in its document windows, then you can easily allow the user to copy and paste between your application's document windows and its dialog boxes, as your application uses `TECut`, `TECopy`, and `TEPaste` for its document windows and `DialogCut`, `DialogCopy`, and `DialogPaste` (which in turn use `TextEdit` routines) for its dialog boxes. These routines all use `TextEdit`'s private scrap, which maintains consistency of data between editing operations.

If your application does not use `TextEdit` for text handling in your document windows and uses a private scrap, then when the user activates a dialog box you should copy any text data in your private scrap to `TextEdit`'s private scrap. When a document window becomes active, if there's data in `TextEdit`'s private scrap, you should copy the data to your private scrap (or the scrap if your application doesn't use a private scrap).

Similarly, before displaying the Standard File Package's save dialog box, your application should copy any text data in its private scrap to the scrap. The Standard File Package reads the data from the scrap when the user chooses an editing operation and a standard file dialog box is active. So your application needs to put the text data (if any) from the last Cut or Copy command in the scrap before calling `StandardPutFile`.

Scrap Manager Reference

This section describes the data structures and routines that are specific to the Scrap Manager. The "Data Structures" section describes the scrap information record and scrap format types. The "Routines" section describes the routines that your application can use to read and write data to the scrap and to get information about data in the scrap.

Data Structures

This section describes the scrap information record and the standard scrap format types.

The Scrap Information Record

The Scrap Manager returns information about the scrap in a scrap information record. The scrap information record is defined by the `ScrapStuff` data type.

```

TYPE
  ScrapStuff =                {scrap information record}
  RECORD
    scrapSize:    LongInt;    {size (in bytes) of scrap}
    scrapHandle:  Handle;     {handle to scrap}
    scrapCount:   Integer;    {indicates whether the contents }
                                { of the scrap have changed}
    scrapState:   Integer;    {indicates state and location }
                                { of scrap}
    scrapName:    StringPtr;  {filename of the scrap}
  END;
  PScrapStuff = ^ScrapStuff; {pointer to scrap info record}

```

Field descriptions

<code>scrapSize</code>	The size of the scrap in bytes.
<code>scrapHandle</code>	A handle to the scrap if it's in memory; otherwise, this field is <code>NIL</code> .
<code>scrapCount</code>	A number that changes each time your application (or another application) calls the <code>ZeroScrap</code> function. When your application receives a suspend event, it should copy any data from its private scrap to the scrap and it can save the value of the <code>scrapCount</code> field. Upon receiving a resume event, your application can use the <code>InfoScrap</code> function to examine the current value of the <code>scrapCount</code> field. If the value in the <code>scrapCount</code> field is different from the previous value, the contents of the scrap have changed and your application should copy the data from the scrap to its private scrap. Alternatively, rather than saving and examining the value of the <code>scrapCount</code> field, your application can check the <code>convertClipboardFlag</code> bit of the event record for a resume event. If this bit is set, the contents of the scrap have changed and your application should take the appropriate actions.

Scrap Manager

`scrapState` The location and state of the scrap. This field is positive if the scrap data is in memory, 0 if the scrap data is on the disk, or negative if the scrap hasn't been initialized.

Note

In unusual circumstances the value of `scrapState` might be 0 when the scrap is actually in memory. This can occur if the user deletes the scrap file on disk and then performs a cut or copy operation. ^u

`scrapName` The filename of the scrap when the scrap is stored on disk. Usually the scrap file is named "Clipboard". The scrap file is always stored on the startup volume.

The Scrap Format Types

Data in the scrap is defined by a scrap format type, a four-character sequence that defines the type of data.

```
TYPE ResType = PACKED ARRAY[1..4] OF Char;
```

The standard scrap format types are

`n` 'TEXT': a series of ASCII characters

`n` 'PICT': a QuickDraw picture, which is a saved sequence of QuickDraw commands that can be displayed using the `DrawPicture` procedure

Optional scrap format types include

`n` 'styl': a series of bytes that have the same format as a `TextEdit` 'styl' resource and that describe styled text data

`n` 'snd ': a series of bytes that have the same format as an 'snd ' resource and that define a sound

`n` 'movv': a series of bytes that have the same format as an 'movv' resource and that define a movie

Your application should support the 'TEXT' and 'PICT' scrap format types and should optionally support any other scrap format types (such as 'snd ') that are appropriate to your application.

In general, when your application writes data to the scrap, the Scrap Manager appends the data to the scrap in this format:

Number of bytes	Contents
4	Scrap format type
4	Length of following data in bytes
<i>n</i>	Data; <i>n</i> must be even

Routines

This section describes the routines you use to

- n get information about the scrap
- n write data to the scrap
- n read data from the scrap
- n store the scrap in memory onto disk
- n read the scrap from disk into memory

Getting Information About the Scrap

You can get information about the scrap using the `InfoScrap` function.

InfoScrap

You can use the `InfoScrap` function to get information about the scrap.

```
FUNCTION InfoScrap: PScrapStuff;
```

DESCRIPTION

The `InfoScrap` function returns a pointer to a scrap information record. The information in the scrap information record provides

- n the size (in bytes) of the scrap
- n a handle to the scrap if it's in memory
- n a count, or number, that your application can use to determine whether the contents of the scrap have changed
- n the location of the scrap (whether in memory or on disk)
- n the filename of the scrap when it is on the disk

ASSEMBLY-LANGUAGE INFORMATION

You can also access the same information as that stored in the scrap information record using system global variables that have the same names as the fields of the scrap information record.

SEE ALSO

See “Getting Information About the Scrap” on page 2-15 for an example that uses the `InfoScrap` function to get information about the scrap. See page 2-32 for information on the fields of the scrap information record.

Writing Information to the Scrap

To write information to the scrap, first use the `ZeroScrap` function to clear the contents of the scrap, and then use the `PutScrap` function to write data in a specific format to the scrap. You can use the `PutScrap` function multiple times to place data in more than one format in the scrap.

ZeroScrap

You use the `ZeroScrap` function to clear the contents of the scrap before writing data to the scrap.

```
FUNCTION ZeroScrap: LongInt;
```

DESCRIPTION

If the scrap already exists (in memory or on the disk), the `ZeroScrap` function clears its contents; otherwise, `ZeroScrap` initializes the scrap in memory. Whenever your application needs to write data to the scrap as a result of a cut or copy operation by the user, you should call `ZeroScrap` before calling `PutScrap`. Whenever your application needs to write data in one or more formats to the scrap, you should call `ZeroScrap` before the first time you call `PutScrap`.

If your application uses `TEToScrap` to write TextEdit’s scrap to the scrap, your application should call `ZeroScrap` to clear the contents of the scrap first. However, note that your application does not have to call `ZeroScrap` before calling `TECut` or `TECopy`.

The `ZeroScrap` function returns a long integer with the value `noErr` if `ZeroScrap` successfully clears the contents of or initializes the scrap. Otherwise, the `ZeroScrap` function returns a nonzero value, whose value corresponds to a result code.

SPECIAL CONSIDERATIONS

Your application should not call the `ZeroScrap` function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>memFullErr</code>	-108	Not enough memory in heap zone

PutScrap

You can use the `PutScrap` function to write data in a specific format to the scrap.

```
FUNCTION PutScrap (length: LongInt; theType: ResType; source: Ptr)
                 : LongInt;
```

<code>length</code>	The number of bytes of data to write to the scrap.
<code>theType</code>	The scrap format type of the data to be written to the scrap. The scrap format type is a four-character sequence that refers to the particular data format, such as 'TEXT', 'PICT', 'styl', 'snd ', or 'movv'.
<code>source</code>	A pointer to the data that the <code>PutScrap</code> function should write to the scrap.

DESCRIPTION

The `PutScrap` function writes the specified number of bytes of data from the memory location pointed to by the `source` parameter to the scrap. The Scrap Manager writes the data to the current location of the scrap (your application's heap or disk).

Whenever your application needs to write data to the scrap as a result of a cut or copy operation, your application uses the `PutScrap` function to write a representation of the data to the scrap. If your application uses a private scrap, it should copy data from its private scrap to the scrap using the `PutScrap` function whenever it receives a suspend event. Your application can use the `PutScrap` function multiple times to write different formats of the same data to the scrap.

IMPORTANT

Whenever your application needs to write data in one or more formats to the scrap, you should call `ZeroScrap` before the first time you call `PutScrap`.

If your application writes multiple formats to the scrap, you should write your application's preferred scrap format type first. For example, if the `SurfWriter` application's preferred scrap format type is a private scrap format type called 'SURF' and `SurfWriter` also supports the scrap format types 'TEXT' and 'PICT', then `SurfWriter` should write the data to the scrap using the 'SURF' scrap format type first, and then write any other scrap format types that it supports in subsequent order of preference.

S WARNING

Do not write data to the scrap that has the same scrap format type as any data already in the scrap. If you do so, the new data is appended to the scrap. Note that when you request data from the scrap using the `GetScrap` function, `GetScrap` returns the first data that it finds with the requested scrap format type; thus you cannot retrieve any appended data of the same format type using `GetScrap`.

If your application uses `TextEdit` to handle text in its documents, use `TextEdit` routines to implement cut and copy operations and to write the `TextEdit` scrap to the scrap. If your application uses the `Dialog Manager` to handle editable text in your application's dialog boxes and a dialog box is the frontmost window, use the `Dialog Manager` procedure `DialogCut` or `DialogCopy` to copy the data from the current editable text item to the scrap.

If the scrap does not already exist (in memory or on the disk), the `PutScrap` function returns a long integer with the value `noScrapErr`. The `PutScrap` function returns other nonzero values corresponding to result codes if an error occurs.

SPECIAL CONSIDERATIONS

The `PutScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>noScrapErr</code>	-100	Scrap does not exist (not initialized)

SEE ALSO

See Listing 2-1 on page 2-16, Listing 2-2 on page 2-18, and Listing 2-7 on page 2-27 for examples that write data to the scrap. If your application uses a private scrap, see "Handling Editing Operations in Dialog Boxes" on page 2-31 for information on maintaining consistency of the scrap when copying and pasting data between document windows and dialog boxes. See *Inside Macintosh: Text* for information on `TextEdit`. See *Inside Macintosh: Imaging With QuickDraw* for information on the `QuickDraw 'PICT'` format.

Reading Information From the Scrap

To read information from the scrap, use the `GetScrap` function.

GetScrap

You can use the `GetScrap` function to read data of a specific format from the scrap.

```
FUNCTION GetScrap (hDest: Handle; theType: ResType;
                 VAR offset: LongInt): LongInt;
```

<code>hDest</code>	A handle to the memory location where the <code>GetScrap</code> function should place the data from the scrap. If you specify <code>NIL</code> in this parameter, the <code>GetScrap</code> function does not read in the data but does return the offset of the data in the scrap and the number of bytes of the requested scrap data type if the requested type exists in the scrap.
<code>theType</code>	The scrap format type of the data to be read from the scrap.
<code>offset</code>	The <code>GetScrap</code> function returns in this parameter the location of the data in the scrap. This value is expressed as an offset (in bytes) from the beginning of the scrap. If the Translation Manager is available, the value of the <code>offset</code> parameter is undefined.

DESCRIPTION

The `GetScrap` function looks in the scrap for any data of the requested scrap format type and returns the first data of the requested type that it finds. The `GetScrap` function writes the data to the memory location specified by the `hDest` parameter.

The `GetScrap` function reads the data from the scrap, makes a copy of it in memory, and sets the handle specified by the `hDest` parameter to refer to this copy. The `GetScrap` function resizes the handle specified by the `hDest` parameter if necessary.

Your application can use the `GetScrap` function multiple times to read different formats of the same data from the scrap. If more than one format of the same scrap format type exists in the scrap, the `GetScrap` function returns the first occurrence of that format type that it finds. For example, if data of type `'TEXT'`, `'PICT'`, and `'TEXT'` exist on the scrap, and your application requests the data in the scrap with scrap format type `'TEXT'`, the `GetScrap` function returns the first data of type `'TEXT'` that it finds.

If your application supports more than one scrap format type, your application should attempt to read its preferred scrap format type first. If your application doesn't prefer one scrap format type over any other type, it should try reading each of the scrap format types that it supports and use the type that returns the lowest offset. The scrap format type with the lowest offset indicates that this format type was written before any of the others and therefore was preferred by the application that wrote it.

Scrap Manager

Note

The returned value for the `offset` parameter is valid only if the Translation Manager isn't available; if the Translation Manager is available, then your application should not rely on the `offset` value. ^u

If you request a scrap format type that isn't in the scrap and the Translation Manager is available, the Scrap Manager uses the Translation Manager to convert the data of a scrap format type that does exist in the scrap into the scrap format type requested by your application. For example, if the SurfWriter application requests data from the scrap in the 'SURF' scrap format type, and the data in the scrap is available in the format types 'TEXT', 'PICT', and 'SDBS' (SurfDB's private scrap format type), the Scrap Manager uses the Translation Manager to convert any one of the scrap format types 'TEXT', 'PICT', or 'SDBS' into the 'SURF' scrap format type. The Translation Manager looks for a translator that can perform one of these translations. If such a translator is available (for example, a translator that can translate the 'SDBS' scrap format type into the 'SURF' scrap format type), the Translation Manager uses the translator to translate the data in the scrap into the requested scrap format type. If the translation is successful, the Scrap Manager returns to your application the data from the scrap in the requested scrap format type.

If your application uses TextEdit to handle text in its documents, use TextEdit routines to implement the paste operation and to copy data from the scrap to the TextEdit scrap. If your application uses the Dialog Manager to handle editable text items in your application's dialog boxes and a dialog box is the frontmost window, use the Dialog Manager procedure `DialogPaste` to copy data from the scrap to the current editable text item.

If the `GetScrap` function successfully reads the data of the requested scrap format type from the scrap, `GetScrap` returns as its function result the length (in bytes) of the data. Otherwise, `GetScrap` returns a negative function result that indicates the error. If `GetScrap` returns the constant `noTypeErr`, then the data in the scrap isn't available in the scrap format type requested by your application. If the Translation Manager is available and `GetScrap` returns the constant `noTypeErr`, this value also indicates that the Translation Manager could not find any translators to convert the data into the scrap format type requested by your application.

```
CONST noTypeErr = -102; {no data of the requested scrap format type}
```

SPECIAL CONSIDERATIONS

The `GetScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

See Listing 2-4 on page 2-21, Listing 2-5 on page 2-24, and Listing 2-7 on page 2-27 for examples that read data from the scrap. If your application uses a private scrap, see “Handling Editing Operations in Dialog Boxes” on page 2-31 for information on maintaining consistency of the scrap when copying and pasting data between document windows and dialog boxes. See *Inside Macintosh: Text* for information on TextEdit. See *Inside Macintosh: Imaging With QuickDraw* for information on the QuickDraw 'PICT' format.

Transferring Data Between the Scrap in Memory and the Scrap on Disk

When system software launches your application, it initially allocates space in your application's heap for the scrap. To write the scrap from memory to the scrap file, use the `UnloadScrap` function. To read data from a scrap file into memory, use the `LoadScrap` function.

UnloadScrap

You can use the `UnloadScrap` function to write the scrap from memory to the scrap file.

```
FUNCTION UnloadScrap: LongInt;
```

DESCRIPTION

The `UnloadScrap` function writes the scrap in memory to the scrap file and releases the memory occupied by the scrap in your application's heap. The scrap file is located in the System Folder of the startup volume and has the filename as indicated by the `scrapName` field of the scrap information record (usually “Clipboard”). If the scrap is already on the disk, the `UnloadScrap` function does nothing.

`UnloadScrap` returns as its function result a long integer corresponding to a result code.

SPECIAL CONSIDERATIONS

The `UnloadScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>dskFulErr</code>	-34	Disk full
<code>ioErr</code>	-36	I/O error

LoadScrap

You can use the `LoadScrap` function to read the scrap from the scrap file into memory.

```
FUNCTION LoadScrap: LongInt;
```

DESCRIPTION

The `LoadScrap` function allocates memory in your application's heap to hold the scrap and then reads the scrap from the scrap file into memory. The scrap file is located in the System Folder of the startup volume and has the filename (usually "Clipboard") as indicated by the `scrapName` field of the scrap information record. If the scrap is already in memory, `LoadScrap` does nothing.

`LoadScrap` returns as its function result a long integer corresponding to a result code.

SPECIAL CONSIDERATIONS

The `LoadScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>memFullErr</code>	-108	Not enough memory in heap zone

Summary of the Scrap Manager

Pascal Summary

Constants

```
gestaltScrapMgrAttr          = 'scra'; {Gestalt selector for }
                                { Scrap Mgr attributes}
gestaltScrapMgrTranslationAware = 0;   {check this bit in the }
                                { response parameter to see }
                                { whether Scrap Mgr supports }
                                { Translation Mgr}
```

Data Types

TYPE

```
ScrapStuff = {scrap information record}
RECORD
  scrapSize:   LongInt;   {size (in bytes) of scrap}
  scrapHandle: Handle;    {handle to scrap}
  scrapCount:  Integer;   {indicates whether the contents }
                                { of the scrap have changed}
  scrapState:  Integer;   {indicates state and location }
                                { of scrap}
  scrapName:   StringPtr; {filename of the scrap}
END;
PScrapStuff = ^ScrapStuff; {pointer to a scrap information record}
```

Routines

Getting Information About the Scrap

```
FUNCTION InfoScrap          : PScrapStuff;
```

Writing Information to the Scrap

```

FUNCTION ZeroScrap          : LongInt;
FUNCTION PutScrap          (length: LongInt; theType: ResType; source: Ptr)
                          : LongInt;

```

Reading Information From the Scrap

```

FUNCTION GetScrap          (hDest: Handle; theType: ResType;
                          VAR offset: LongInt): LongInt;

```

Transferring Data Between the Scrap in Memory and the Scrap on Disk

```

FUNCTION UnloadScrap      : LongInt;
FUNCTION LoadScrap       : LongInt;

```

C Summary

```

enum {
#define gestaltScrapMgrAttr      'scra'      /*Gestalt selector for */
                                      /* Scrap Mgr attributes*/
gestaltScrapMgrTranslationAware = 0        /*check this bit in the */
                                      /* response parameter to see */
                                      /* whether Scrap Mgr supports */
                                      /* Translation Mgr*/
};

```

Data Types

```

struct ScrapStuff {          /*scrap information record*/
    long      scrapSize; /*size (in bytes) of scrap*/
    Handle    scrapHandle; /*handle to scrap*/
    short     scrapCount; /*indicates whether the contents */
                                      /* of the scrap have changed*/
    short     scrapState; /*indicates state and location */
                                      /* of scrap*/
    StringPtr scrapName; /*filename of the scrap*/
};
typedef struct ScrapStuff ScrapStuff;
typedef ScrapStuff *PScrapStuff;

```

Routines

Getting Information About the Scrap

```
pascal PScrapStuff InfoScrap  
                                (void);
```

Writing Information to the Scrap

```
pascal long ZeroScrap          (void);  
pascal long PutScrap           (long length, ResType theType, Ptr source);
```

Reading Information From the Scrap

```
pascal long GetScrap           (Handle hDest, ResType theType, long *offset);
```

Transferring Data Between the Scrap in Memory and the Scrap on Disk

```
pascal long UnloadScrap       (void);  
pascal long LoadScrap         (void);
```

Assembly-Language Summary

Data Structures

Scrap Information Data Structure

0	ScrapSize	long	size (in bytes) of the scrap
4	ScrapHandle	long	handle to scrap
8	ScrapCount	2 bytes	indicates whether the contents of the scrap have changed
10	ScrapState	2 bytes	indicates state and location of scrap
12	ScrapName	long	pointer to the filename of the scrap

Result Codes

noErr	0	No error
dskFulErr	-34	Disk full
ioErr	-36	I/O error
noScrapErr	-100	Scrap does not exist (not initialized)
noTypeErr	-102	No data of the requested scrap format type in scrap
memFullErr	-108	Not enough memory in heap zone

Help Manager

Contents

About the Help Manager	3-6
How the Help Manager Displays Balloons	3-8
Default Help Balloons for Menus, Windows, and Icons	3-13
About BalloonWriter	3-17
Using the Help Manager	3-18
Providing Text or Pictures for Help Balloons	3-18
Defining Help Messages	3-19
Using Clear, Concise Phrases	3-20
Using Active Constructions	3-22
Using Parallel Structure	3-22
Offering Hints	3-22
Using Consistent Terminology	3-23
Defining the Help Balloon Position	3-23
Specifying the Format for Help Messages	3-23
Specifying Options in Help Resources	3-25
Providing Help Balloons for Menus	3-27
Specifying Header Information for the 'hmnu' Resource	3-32
Specifying Help for Menu Items Missing From the Resource	3-33
Specifying Help for Menu Titles and for Items Dimmed by System Software	3-36
Specifying Help for Menu Items	3-39
Specifying Help for a Changing Menu Item	3-43
Specifying Resources by Item Name	3-45
Providing Help Balloons for Menus You Disable for Dialog Boxes	3-47
Providing Help Balloons for Items in Dialog Boxes and Alert Boxes	3-51
Specifying Header Information for the 'hdlg' Resource	3-54
Specifying Missing-Item Information	3-54
Specifying Help for Items in an Alert or Dialog Box	3-56
Adding a Help Item to an Item List Resource	3-62
Using a Help Item Versus Using an 'hwin' Resource	3-63

Providing Help Balloons for Window Content	3-63
Providing Help Balloons for Static Windows	3-65
Specifying Header Information for the 'hrct' Resource	3-67
Specifying Help for Rectangles in Windows	3-67
Associating Help Resources With Static Windows	3-68
Specifying Header Information for the 'hwin' Resource	3-69
Specifying 'hdlg' or 'hrct' Resources in the 'hwin' Resource	3-69
Providing Help Balloons for Dynamic Windows	3-74
Overriding Help Balloons for Non-Document Icons	3-84
Specifying Header Information for the 'hldr' Resource	3-85
Specifying Help for an Icon	3-85
Overriding Other Default Help Balloons	3-87
Specifying Header Information for the 'hovr' Resource	3-88
Overriding Default Help	3-88
Adding Menu Items to the Help Menu	3-90
Writing Your Own Balloon Definition Function	3-93
Help Manager Reference	3-95
Data Structures	3-95
The Help Message Record	3-95
The Help Manager String List Record	3-97
Help Manager Routines	3-97
Determining Balloon Help Status	3-98
Displaying and Removing Help Balloons	3-99
Enabling and Disabling Balloon Help Assistance	3-107
Adding Items to the Help Menu	3-108
Getting and Setting the Font Name and Size	3-110
Setting and Getting Information for Help Resources	3-114
Determining the Size of a Help Balloon	3-119
Getting the Message of a Help Balloon	3-122
Application-Defined Routines	3-128
Resources	3-132
The Menu Help Resource	3-132
The Dialog-Item Help Resource	3-140
The Rectangle Help Resource	3-148
The Window Help Resource	3-154
The Finder Icon Help Resource	3-156
The Default Help Override Resource	3-160
Summary of the Help Manager	3-166
Pascal Summary	3-166
Constants	3-166
Data Types	3-168
Help Manager Routines	3-169
Application-Defined Routines	3-170
C Summary	3-170
Constants	3-170
Data Types	3-173
Help Manager Routines	3-173

CHAPTER 3

Application-Defined Routines	3-175
Assembly-Language Summary	3-176
Data Structures	3-176
Trap Macros	3-176
Result Codes	3-177

Help Manager

This chapter describes how you can use the Help Manager to provide your users with Balloon Help online assistance—information that describes the actions, behaviors, or properties of your application’s features. When the user turns on Balloon Help assistance, the Help Manager displays small help balloons as the user moves the cursor over areas such as controls, menus, and rectangular areas in your windows. **Help balloons** are rounded-rectangle windows that contain explanatory information for the user. (With tips pointing at the objects they annotate, help balloons look like the balloons used for dialog in comic strips.) You provide **help messages** in the form of descriptive text or pictures that appear inside help balloons. Your help messages should be short and pertinent to the object over which the cursor is located.

For example, when a user moves the cursor to a menu command, a help balloon should point to that command and explain its purpose. The help balloon remains displayed until the user moves the cursor away.

The user turns on Balloon Help online assistance for all applications by choosing the Show Balloons command from the Help menu. All normally available features of your application are still active when Balloon Help is enabled. The help balloons only provide information; the actions that the user performs by pressing the mouse button still take effect as they normally would.

The Help Manager is available in System 7. Use the `Gestalt` function to determine whether the Help Manager is present.

Read this chapter if you want to provide help balloons for your application, desk accessory, control panel, Chooser extension, or other software that interacts with the user. If you offer an additional help facility for your users, you should give users access to your information through the Help menu. This chapter explains how you can add your own menu items to the Help menu to provide one convenient and consistent place for users to look for help information.

You can provide help balloons for your menus, dialog boxes, alert boxes, and non-document icons by simply adding resources to your resource file. To provide help for the content area of windows, you can use either resources or Help Manager routines. Both methods are described in this chapter.

You typically provide help balloons for your application by creating resources—such as the `'hmenu'` resource, which the Help Manager uses when displaying help balloons for your menu items. In the `'hmenu'` resource, you specify help balloons for menu titles and menu items in their enabled and disabled (that is, dimmed) states. Menus are described in the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

To provide help balloons for alert boxes and dialog boxes, you typically create an `'hdlg'` resource that specifies help balloons for the various items identified in the item list (`'DITL'`) resource for the alert box or dialog box. If the items include any controls, such as simple buttons, checkboxes, or complex multipart controls, you specify help according to the control’s state—active or inactive (that is, dimmed), and checked or not checked (if applicable). For every item that is not a control, you can provide different help balloons depending on whether the item is enabled or disabled—that is, depending on whether you asked the Dialog Manager to return information regarding events in that item. Dialog boxes and alert boxes are described in the chapter “Dialog Manager” in

Help Manager

Inside Macintosh: Macintosh Toolbox Essentials; controls are described in the chapter “Control Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Depending on whether your windows are static or whether they contain changing or scrolling information, you use Help Manager resources or Help Manager routines to provide the content areas of your windows with help balloons. To provide help balloons for the static windows of your application without modifying its code, you create a resource of type 'hwin' and another resource of type 'hrct' or of type 'hdlg'. The 'hwin' resource identifies windows by the titles or the windowKind values in their window records. To provide help balloons for portions of windows that change or scroll, you must identify, track, and update those portions within your windows, and then use the Help Manager function `HMShowBalloon` to display help balloons for those portions. Windows are described in the chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter provides a brief description of how the Help Manager displays help balloons. It provides information on the default help balloons and then discusses how to

- n use text or a picture for the help message inside a balloon
- n create resources for help balloons for menus, dialog boxes, and alert boxes
- n create resources for help balloons for windows
- n override the default help balloons provided by system software
- n add your own menu items to the Help menu
- n write your own balloon definition function

About the Help Manager

You can use the Help Manager to provide help for these interface features of your application:

- n menu titles and menu items
- n dialog boxes and alert boxes
- n windows, including any object in the frame or content area
- n icons for any desktop objects other than documents
- n other application-defined areas

Providing help balloons for menus, dialog boxes, or alert boxes is quite simple, because you need only to create resources; you don't have to alter any of your existing code. The Help Manager automatically sizes, positions, and draws the help balloon and its help message for you. It is equally simple to provide help balloons for a window whose contents don't change location within its content area.

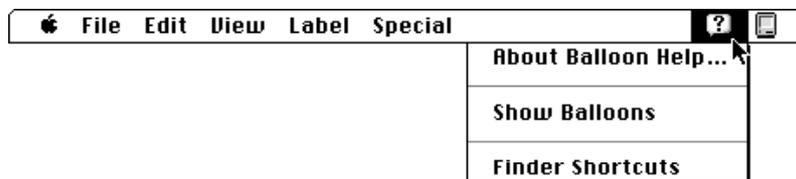
It takes a little more work to provide help balloons for windows in your application that contain objects that are dynamic or that change their position within the content areas of their windows. You provide Balloon Help assistance for these objects by tracking the

Help Manager

cursor yourself and using Help Manager routines to display help balloons. You can let the Help Manager remove help balloons, or your application can determine when to remove help balloons.

The user turns on Balloon Help online assistance by choosing Show Balloons from the Help menu, which is shown in Figure 3-1. Once the user chooses Show Balloons, help is enabled for all applications. The Help menu appears to the right of all your menus and to the left of the Application menu (and to the left of the Keyboard menu if a non-Roman script system is installed). Users can turn on Balloon Help assistance even when your application presents an alert box or a modal dialog box, because the Help menu is always enabled.

Figure 3-1 The Help menu for the Finder



When Balloon Help assistance is enabled, the Help Manager displays any help balloons for the current application whenever the user moves the cursor over a rectangular area that has a help balloon associated with it. For those balloons defined in Help Manager resources, the Help Manager automatically tracks the cursor and generates the shape and calculates the position for the help balloon. The Help Manager removes the help balloon when the cursor is no longer located over the associated area.

The Help Manager provides a default help balloon for inactive windows and displays default help balloons for the title bar and other parts of the active window. The Help Manager also displays default help balloons for other standard features of an application's user interface. "Default Help Balloons for Menus, Windows, and Icons" beginning on page 3-13 describes the default help balloons. (Though you probably won't want or need to change the messages in these default balloons, you have the ability to do so, as described in "Overriding Other Default Help Balloons" on page 3-87.) The Help Manager displays the default help balloons for your application whenever Balloon Help assistance is enabled, even if your application does not explicitly use or create help balloons.

Help balloons do not interfere with your application. Because the Help Manager can display a balloon whether the mouse button is down or up, the user can still click and double-click to use the normal features of your application.

When the user chooses Hide Balloons from the Help menu, the Help Manager removes any visible help balloon and stops displaying help balloons until Balloon Help assistance is enabled again.

How the Help Manager Displays Balloons

The Help Manager performs most of the work involved with rendering help balloons for your application. This section gives an overview of the facilities that the Help Manager uses to display help balloons.

The Help Manager uses the Window Manager to create a special type of window for the help balloon and then draws the help message in the port rectangle of the window. The Help Manager is responsible for

- n calculating the size of the help balloon (based on the help message you provide)
- n determining line breaks for text in a help balloon
- n calculating a position for the help balloon so that it appears onscreen
- n drawing the help balloon and your help message onscreen

A **balloon definition function**, which is an implementation of a window definition function, defines the general appearance of the help balloon. A standard balloon definition function is provided for you, and it is responsible for

- n calculating the help balloon's content region and structure region, which are based on the rectangle calculated by the Help Manager
- n drawing the frame of the help balloon

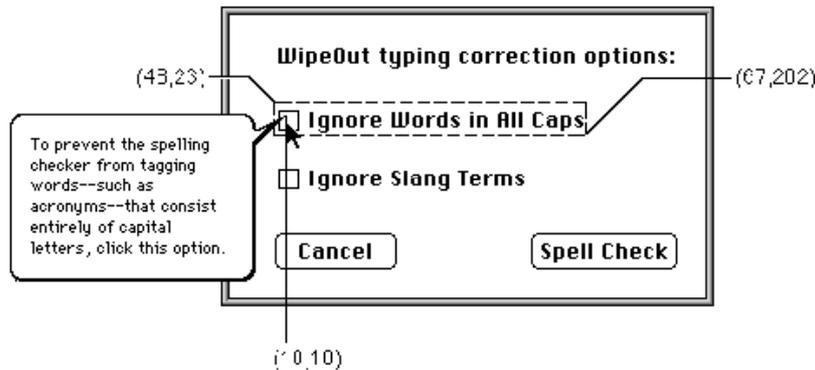
For help balloons, the content region is the area inside the balloon frame; it contains the help message. The structure region is the boundary region of the entire balloon, including the content area and the pointer that extends from one of the help balloon's corners.

The standard balloon definition function is the window definition function (a 'WDEF' resource) with resource ID 126. Figure 3-2 shows the general shape of a help balloon drawn with this standard balloon definition function.

Figure 3-2 A help balloon drawn with the standard balloon definition function



Every help balloon is further defined by its hot rectangle, its tip, and a variation code.

Figure 3-3 The tip and hot rectangle for a help balloon

The **hot rectangle** encloses the area for which you want to provide Balloon Help online assistance. When the user moves the cursor over a hot rectangle, the Help Manager displays the rectangle's help balloon; the Help Manager removes the help balloon when the user moves the cursor away from the hot rectangle. To prevent balloons from flashing excessively, the Help Manager does not display a balloon unless the user leaves the cursor at the same location for a short time (around one-tenth of a second). This length of time is set by the system software and cannot be changed.

In Figure 3-3, the help balloon is displayed for a hot rectangle defined by coordinates (48,23,67,202), which are local to the window. The Help Manager displays and removes the help balloon as the cursor moves in and out of the area defined by the hot rectangle.

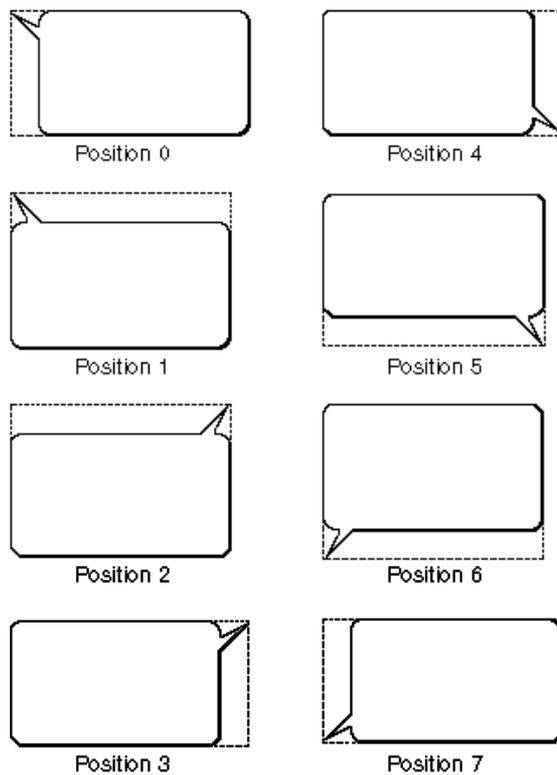
A small pointer extends from a corner of every help balloon, indicating the object or area that is explained in the help balloon. The **tip** is the point at the end of this extension. Figure 3-3 shows an example of a help balloon for a control. The balloon tip is at the coordinates (10,10), which are local to the hot rectangle.

A **variation code** specifies the preferred position of the help balloon relative to the hot rectangle. The balloon definition function draws the frame of the help balloon based on that variation code.

Help Manager

As shown in Figure 3-4, the standard balloon definition function provides eight different positions, which you can specify with a variation code from 0 to 7. The figure also shows the boundary rectangle for each shape. Note that the tip of the help balloon always aligns with an edge of the boundary rectangle. If you write your own balloon definition function, you should support the tip locations defined by the standard variation codes.

Figure 3-4 Standard balloon positions and their variation codes



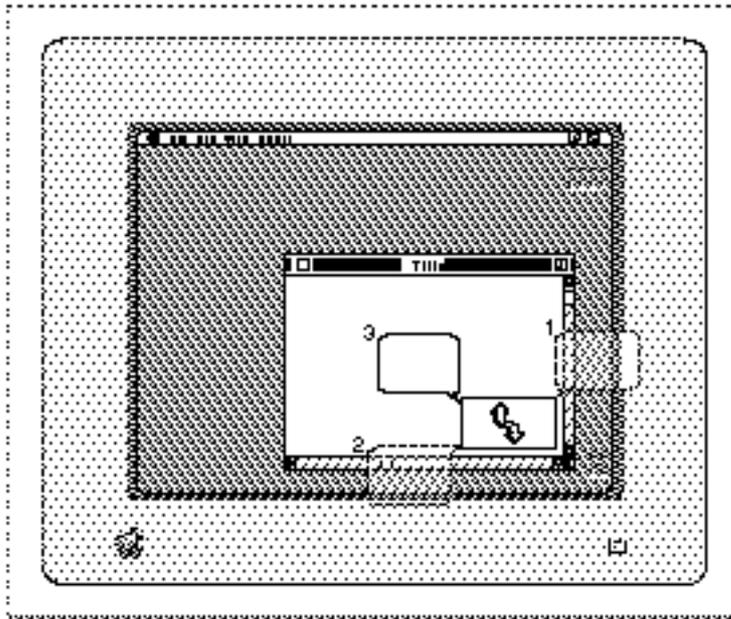
For most of the help balloons it displays, the Finder uses variation code 6. A balloon with variation code 6 has its tip in the lower-left corner and projects up slightly and to the right.

If a help balloon is on the screen and not in the menu bar, the Help Manager uses the specified variation code to display the help balloon. If a help balloon is offscreen or in the menu bar, the Help Manager attempts to display the help balloon by using different variation codes and by moving the tip. Usually, the Help Manager moves the tip by transposing it across the horizontal and vertical planes of the hot rectangle.

Help Manager

Figure 3-5 shows the Help Manager making three attempts to fit a help balloon onscreen by moving the tip to three different sides of the hot rectangle and using an appropriate variation code for each tip.

Figure 3-5 Alternate positions of a help balloon



When positioning a help balloon onscreen, the Help Manager first determines whether the screen has enough horizontal space and then enough vertical space to display the balloon using the specified variation code and tip. If the help balloon is either too wide or too long to fit onscreen at this position, the Help Manager tries a combination of different variation codes and transpositions around the hot rectangle. In Figure 3-5, the Help Manager uses a new variation code, moves the tip to a different side of the hot rectangle, and again tests whether the help balloon fits onscreen. If, after exhausting all possible positions, the Help Manager still cannot fit the entire help balloon onscreen, the Help Manager displays a help balloon at the position that best fits onscreen and clips the help message to fit the balloon at this position.

When you use `dialog-item help ('hdlg')` resources or the `HMShowBalloon` and `HMShowMenuBalloon` functions, the Help Manager allows you to specify **alternate rectangles**, which give you additional flexibility in positioning your help balloons onscreen. The Help Manager uses the alternate rectangle instead of the hot rectangle for transposing help balloons to make them fit onscreen. If you make your alternate rectangle smaller than your hot rectangle, for example, you have greater assurance that the Help Manager will be able to fit the help balloon onscreen; if you specify an alternate rectangle that is larger than your hot rectangle, you have greater assurance that the help balloon will not obscure some object explained by the balloon.

Help Manager

To provide help balloons under most circumstances, you create **help resources**, which specify the help messages, the balloon definition functions, the variation codes, and, when necessary, the tips and the hot rectangles or alternate rectangles for the Help Manager to use in drawing help balloons. These help resources are

- n the menu help ('hmenu') resource, which provides help balloons for menus and menu items
- n the dialog-item help ('hdlg') resource, which provides help balloons for items in dialog boxes and alert boxes
- n the rectangle help ('hrct') resource, which associates a help balloon with a hot rectangle in a static window
- n the window help ('hwin') resource, which associates an 'hrct' or 'hdlg' resource with a hot rectangle in a window or with an item in a dialog box or alert box
- n the Finder icon help ('hldr') resource, which provides a custom help balloon message for your application icon
- n the default help override ('hovr') resource, which overrides the help messages of default help balloons provided in system software

To put help balloons in your application, you have a number of responsibilities:

- n You must create any necessary help resources for your application.
- n You must provide the help messages that appear in the balloons. Although you can store these messages in the help resources themselves or in data structures, localizing your help messages is much easier if you store them in other resources—such as 'PICT', 'STR#', 'STR ', 'TEXT', and 'styl' resources—that are easier to edit.
- n In your help resources you must specify a balloon definition function for your help balloons. Typically, you should use the standard balloon definition function that draws shapes similar to that shown in Figure 3-2 on page 3-8. This helps maintain a consistent look across all help balloons used by the Finder and other applications. However, if you feel absolutely compelled to change the shape of help balloons in your application, you can write your own balloon definition function as described in “Writing Your Own Balloon Definition Function” on page 3-93. Be aware, however, that a different help balloon shape may initially confuse your users.
- n In your help resources you must specify a variation code. The variation code positions your balloons onscreen according to the general shape described by their balloon definition function. If you use the standard balloon definition function, you'll use variation codes 0 to 7 to display the help balloons shown in Figure 3-4 on page 3-10. The preferred variation code is 0. If you are unsure of which variation code to use, specify 0; the Help Manager will use a different variant if another is more appropriate. If you write your own balloon definition function, you must define your own variation codes.

Help Manager

For objects other than menu items, you have these additional responsibilities:

- n In your help resources you must specify coordinates for the balloon's tip. For menu items, the Help Manager automatically places the tip just inside the right edge of the menu item.
- n You must specify rectangles in your help resources. (The hot rectangles for items in menus, alert boxes, and dialog boxes are automatically defined for you by their display rectangles.) For 'hdlg' resources, you specify alternate rectangles for moving the help balloon. For 'hrcr' resources, you specify hot rectangles, which define the areas onscreen for association with help balloons.
- n You must track the cursor in dynamic windows, and, when the cursor moves over a hot rectangle in your window, you must call Help Manager routines (such as `HMShowBalloon`) to display your help balloons. You can let your application or the Help Manager remove the help balloon when the user moves the cursor out of the hot rectangle.

In summary, the Help Manager automatically displays help balloons in the following manner. The user turns Balloon Help assistance on, then moves the cursor to an area described by a hot rectangle. The Help Manager calculates the size of the help balloon based on its help message. The Help Manager uses `TextEdit` to determine word breaks and line breaks of text in the help balloon. The Help Manager then determines the size of the help balloon and uses the Window Manager to create a new help balloon. The Window Manager calls the balloon definition function to determine the help balloon's general shape and position. (If the variation code places the help balloon offscreen or in the menu bar, the Help Manager tries a different variation code or moves the tip of the help balloon to another side of the hot rectangle or the alternate rectangle.) The window definition function draws the frame for the help balloon, and the Help Manager draws the help message of the help balloon.

For most interface features that you want to provide help for, you create the help message (preferably in a separate, easily edited resource) and, in the help resources themselves, you specify the standard balloon definition function, one of the eight variation codes, the tip's coordinates, and (often) a hot rectangle.

The Help Manager does not automatically display help balloons for dynamic windows or for menus using custom menu definition procedures. If you want to provide help balloons for either of these types of objects, or if you want more control over help balloons, you must identify hot rectangles, create your own data structures to store their locations, track the cursor yourself, and call `HMShowBalloon` when the cursor moves to your hot rectangles. If you wish, you can also write your own balloon definition function and tip function.

Default Help Balloons for Menus, Windows, and Icons

The Help Manager displays many default help balloons for an application when help is enabled and the user moves the cursor to certain standard areas of the user interface. These areas include the standard window frame and the menu titles and menu items in

Help Manager

the Apple menu, Help menu, Keyboard menu, and Application menu. You don't need to create any resources or use any Help Manager routines to take advantage of the default help balloons.

The following list summarizes the items that have default help balloons.

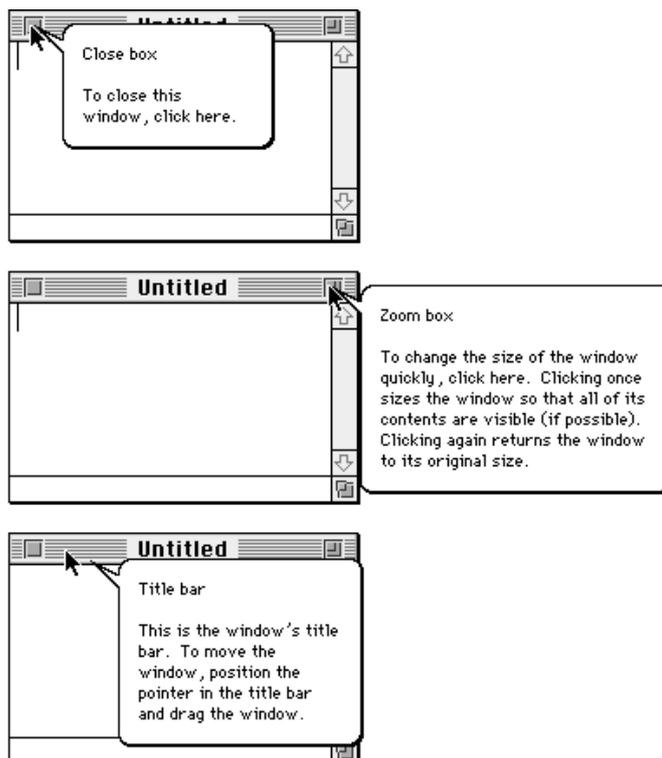
- n Application icon in the Finder. Default help balloons are also provided for desk accessory, system extension, and control panel icons. You can override these help messages.
- n Document icon in the Finder. You cannot override the help message for this icon.
- n Standard file dialog boxes. You supply balloons for items that you add to these dialog boxes; you cannot override the help messages for the other items.
- n Window title bar. A default help balloon is provided for the title bars of windows created with both standard and custom window definition functions (WDEFs). You can override the default help message.
- n Window close box. A default help balloon is provided for the close boxes of windows created with both standard and customized WDEFs. You can override the default help message.
- n Window zoom box. A default help balloon is provided for the zoom boxes of windows created with both standard and customized WDEFs. You can override the default help message.
- n Inactive window. You can override the default help message for inactive windows.
- n Apple menu title. The default help balloon for the title of the Apple menu is available only if your application uses the standard menu definition procedure. You cannot override the default help message for this title.
- n Apple menu items. Default balloons are provided for items that the user moves to the Apple Menu Items folder, but there is no default balloon for the About command or other items that your application adds to this menu; you must provide help balloons for such items.
- n Help menu title. The default help balloon for the title of the Help menu is available only if your application uses the standard menu definition procedure. You cannot override the default help message for this title.
- n Help menu items. Default balloons are provided only for the About Balloon Help and Hide/Show Balloons commands; you must provide help balloons for items you add to this menu. You cannot override the default help messages.
- n Application menu title and items. Default help balloons for the title and items of the Application menu are available only if your application uses the standard menu definition procedure. You cannot override these default help messages.
- n Keyboard menu title. The default help balloon for the title of the Keyboard menu is available only if your application uses the standard menu definition procedure. You cannot override the default help message.

Help Manager

System software uses the Help Manager to display help balloons for most of its dialog boxes and alert boxes. (For example, the Standard File Package provides help balloons for its standard file dialog boxes.) If your application uses a system software routine (such as the `StandardPutFile` procedure) that provides help balloons, and the user has enabled Balloon Help assistance, the Help Manager displays each help balloon as the user moves the cursor to each hot rectangle. If you've added your own buttons, checkboxes, or other controls to such a dialog box or alert box, you can also provide these controls with help balloons.

The Help Manager uses the window definition function of a window to determine whether the cursor is in the window frame and, if so, which region of the window (title bar, close box, or zoom box) the cursor is in. If the cursor is in any of these regions, the Help Manager displays the associated help balloon. Figure 3-6 shows the default help balloons for the active window of an application that uses the standard window definition function. If you use a custom window definition function, the Help Manager also displays these default help balloons for the corresponding regions of your windows.

Figure 3-6 Default help balloons for the window frame



Help Manager

The Help Manager also provides these default help balloons for the title bars, close boxes, and zoom boxes of windows in the Finder. The Finder specifies additional help for other window regions—for example, the scroll bar and size box—although the Help Manager does not automatically provide your window with this help.

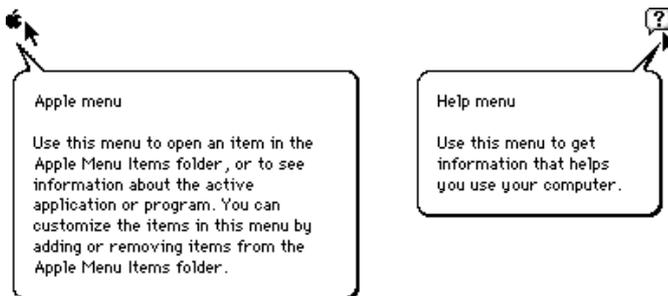
The Help Manager displays help balloons for the standard window frame and other standard areas named in the 'hovr' resource. You can override any of the default help balloons defined in the 'hovr' resource by providing your own resource of type 'hovr'. See “Overriding Other Default Help Balloons” on page 3-87 for more information.

The Help Manager displays default help balloons for the Apple menu, Help menu, and Application menu. The Menu Manager uses the Help Manager to display help balloons for these menus regardless of whether you supply help balloons for the rest of your menus. The Help Manager also provides default help balloons for the Keyboard menu when a non-Roman script system is installed. Figure 3-7 shows the default help balloons for the Apple menu and Help menu titles.

Note

For all menus and menu items, the Help Manager displays help balloons only for applications that use the standard menu definition procedure. If you use your own menu definition procedure, your application must track the cursor and use Help Manager routines to display and remove help balloons, as described in “Displaying and Removing Help Balloons” on page 3-99. u

Figure 3-7 Default help balloons for the Apple and Help menus

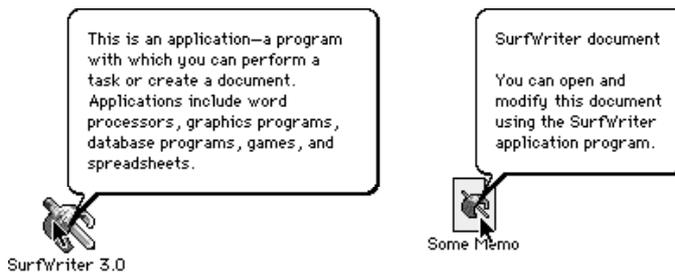


The Help Manager does not provide default help balloons for items you put at the top of your application's Apple menu or items you add to the Help menu. You typically put one item at the top of the Apple menu: the About command for your application. If you have additional user help facilities, list them in the Help menu—not in the Apple menu. You have control only over those items that you add to the Apple and Help menus.

Help Manager

The Finder provides default help balloons for your application icon and any documents created by your application. Figure 3-8 shows the default help balloon for the SurfWriter application and a document created by this application. You can customize the help balloon for your application icon by providing an 'hfd_r' resource; however, you can't customize the default help balloon for the documents created by your application.

Figure 3-8 Default help balloons for application and document icons



About BalloonWriter

Apple Computer, Inc., makes available a tool that greatly facilitates the creation of help balloons. Called BalloonWriter, this tool gives nonprogrammers an easy, intuitive way to create help balloons. Writers who have no programming experience can use BalloonWriter to provide your application with fully functional resource code for menus, dialog and alert boxes, static windows, and non-document Finder icons. In its user's guide, BalloonWriter refers to help balloons for these interface features as *standard balloons*. For these types of help balloons, BalloonWriter creates 'hmnu', 'hdlg', 'hwin', 'hrct', and 'hfd_r' resources, as appropriate, and places them in the resource file of your application. BalloonWriter likewise creates and stores 'STR ', 'STR# ', and 'TEXT' resources that contain the help messages authored by your nonprogramming writers.

For dynamic windows and for menus that use custom menu definition procedures, your application must track the cursor and use the `HMShowBalloon` function to display help balloons. The BalloonWriter documentation refers to these balloons as *custom balloons*. BalloonWriter does not create the necessary resources or code that automatically displays these types of help balloons. However, nonprogrammers can use BalloonWriter to provide you with conveniently delimited ASCII text that you can then use in conjunction with `HMShowBalloon` to display the desired help balloons.

BalloonWriter is available from APDA.

Using the Help Manager

You can use the Help Manager to provide information to the user that describes the action, behavior, or properties of your application's features. For example, you can create a help balloon for each menu item to describe what it does.

To determine whether the Help Manager is available, use the `Gestalt` function with the `gestaltHelpMgrAttr` selector. Test the bit field indicated by the `gestaltHelpMgrPresent` constant in the response parameter. If the bit is set, then the Help Manager is present.

```
CONST gestaltHelpMgrPresent    = 0;        {if this bit is set, then }
                                     { Help Manager is present }
```

The Help Manager is initialized at startup time. The user controls whether help is enabled by choosing the Show Balloons or Hide Balloons command from the Help menu.

The Help menu is specific to each application, just as the File and Edit menus are specific to each application. The Help menu items that are defined by the Help Manager are common to all applications, but you can add your own menu items for help-related information.

The Help Manager automatically appends the Help menu when your application inserts an Apple menu into its menu bar. The Menu Manager automatically appends the Help menu to the right of all your menus and to the left of the Application menu (and to the left of the Keyboard menu if a non-Roman script system is installed).

You can create help balloons for the menus, dialog boxes, alert boxes, or content area of windows belonging to your application. You can also override some of the default help balloons—such as the default help balloon for the title bar of a window.

You can specify the help message by using plain text, styled text, or pictures. Although you should always strive for brevity in your help messages, plain text strings can contain up to 255 characters. Styled text can contain up to 32 KB of information. The Help Manager determines the actual size of the help balloon and, for text strings, uses `TextEdit` to determine word breaks and line breaks.

The Help Manager automatically tracks the cursor and generates help balloons defined in standard help resources. Your application can also track the cursor and use Help Manager routines to display and remove help balloons.

Providing Text or Pictures for Help Balloons

Use help balloons to provide the user with information that describes or explains interface features of your application. The information you supply in help balloons should follow a few general guidelines in order to provide the most useful information to the user. This section describes these guidelines.

For examples of how your application should use help balloons, observe the help balloons that the Finder, the TeachText application, and system software use.

Defining Help Messages

Use help balloons to explain parts of your application's interface that might confuse a new user or features that could help a user become an expert user. The information you provide in help balloons should identify interface features in your application or describe how to use them.

The help balloon for an item appears when the user moves the cursor to that item. Because the user knows exactly what the text is referring to, this is a powerful method of providing information. But the method has some limitations. There are some kinds of information that help balloons cannot display effectively.

- n Help balloons *can* show users what they will accomplish by using onscreen objects, including menu commands, dialog boxes, and tool palettes.
- n Help balloons *can* help experienced Macintosh users who prefer to learn programs by using them, rather than by reading manuals.
- n Help balloons *can't* help users who don't know what they want to do or users who don't know where to look.
- n Help balloons *can't* teach your program by themselves. They can't substitute for task-oriented paper or electronic documentation or training.
- n Help balloons *can't* teach novice Macintosh users the concepts they need to know in order to use the Macintosh computer.

Help balloons work best when you keep your audience in mind as you write. Ask yourself these questions when you are planning balloons for your program:

- n Who will be using your program?
- n What aspects of your program are users unfamiliar with?
- n What terminology are your users likely to know?

Unless your application has a specialized audience, it's best to write for users who already know something about using the Macintosh (although they may not be experts) but who don't know much about your application.

Each help balloon should answer at least one of these questions:

- n **What is this?** For example, when the user moves the cursor to the item count in the upper-right corner of a Finder window, the Finder displays a help balloon that reads "This is the number of files or folders in this window."
- n **What does this do?** For example, when the user moves the cursor to the Find command in the Finder's File menu, the Finder displays a help balloon that reads "Finds and selects items with the characteristics you specify."
- n **What happens when I click this?** For example, when the user moves the cursor to the close box of a window, the Window Manager displays a help balloon that first names the object ("Close box") and then explains, "To close this window, click here."

Help Manager

Help messages should be short and easy to understand. You should not include lengthy instructions or numbered steps in help balloons. Use help balloons to clarify the meaning of objects in your application—for example, tool icons in palettes.

Use simple, clear language in the information you provide. Include definitions in help balloons when appropriate.

You can use graphics or styled text in help balloons to illustrate the effects of a command. For example, to demonstrate the effect of the Bold command in a word-processing application, you might use styled text to show a word in boldface.

You can provide separate help balloons for two display states—enabled and dimmed (disabled)—of a menu item. You can also provide separate help balloons for two display states—active and dimmed (inactive)—of a control. The help balloon that you provide for an enabled menu item should explain the effect of choosing the item. The help balloon that you provide for a dimmed menu item should explain why it isn't currently available, or, if more appropriate, how to make it available. Similarly, the help balloon that you provide for an active control should explain the effect of clicking or selecting the control, and the help balloon that you provide for a dimmed control should explain why it isn't currently available, or, if more appropriate, how to make it available.

Complicated dialog boxes can often benefit from help balloons that explain what's essential about the dialog box. You can use help balloons to describe groups of controls rather than individual controls. For example, if a dialog box has several distinct regions that contain radio buttons or checkboxes, you could provide a help balloon for each set of radio buttons, rather than providing a separate balloon for each button.

If you use a function to customize standard dialog boxes, use as many of the existing help balloons as possible. For example, if your application uses any of the standard file dialog boxes and provides an extra button, you can create a help balloon for the extra button, and the Help Manager continues to use the default help balloons for other items in the dialog box.

To make localization easier, you should store your help messages in resources separate from the help resources. To avoid problems with grammar and sentence structure when you localize your application, never combine separately stored phrases into one help message.

Using Clear, Concise Phrases

You can provide up to 255 characters of information using text strings in help balloons. (You can use up to 32 KB if you use styled text.) However, you should include only the most relevant information in the help balloon. To determine what to provide, decide what information would be most useful to a user. This information usually omits the object's name, which normally doesn't matter to the user, and instead tells what the object is for and what the object does, which does matter to the user.

You might eventually translate your help messages into other languages, so try to keep the messages as short as possible. When translated, your help messages may require more words or longer words—and therefore larger balloons and more screen space. Expect English text to expand 20–30 percent after translation. To keep the translated text

Help Manager

within the Help Manager's 255-character limit for text strings, limit English text to approximately 180 characters.

If an item already has a commonly used name, or if it's a special case of a larger category of objects, name it in the balloon. The Finder, for example, displays the message "Drag the title bar to move the window," since title bars and windows are commonly used names. However, you don't need to name everything in your application just so that you can refer to it in a help balloon. For example, because the tip of the help balloon points to the subject of the help balloon, you can easily say "To apply the style, click here," rather than "The Apply button activates the Styles command. Click the button to activate the command."

Many of the items onscreen don't need names. An item needs a name only if the name helps the user remember how to use the application. The following items are likely to need names:

- n icons that don't already have names on the screen
- n tools in a palette
- n controls on a ruler
- n controls in a paint program
- n Finder icons whose names can be changed

If you decide to name an item, make sure that the name you use in the balloon matches the name used in other documentation.

For balloons that describe menu items, you can use sentence fragments; the grammatical subject is obvious from the context. For example, the help balloon for the Open command could read "Opens the selected file" rather than "This command opens the selected file"; the grammatical subject is obvious from the context. Using sentence fragments lets users assimilate the message more quickly because they have fewer words to read.

When you describe a menu item or a button, try to use a word that's different from the one that appears onscreen. Using a synonym in this way helps users who aren't sure what the item's name means. For example, the help balloon for a Paste command in the Edit menu might say something like "Inserts the contents of the Clipboard into the document."

Help balloons are usually inappropriate for describing multiple-step procedures, because a help balloon does not stay on the screen while the user performs the various steps. The user may begin a procedure described in a help balloon and then become confused when the information disappears.

You can, however, describe a very simple two-step procedure in a balloon. This is probably most appropriate for a tool in a palette. For example, the balloon for an eraser tool might first define the tool as an eraser and then explain, "To remove parts of your drawing, click this icon, then drag to erase those parts you want to remove."

Using Active Constructions

Try to use short, active phrases in help balloons. Avoid passive constructions. An active construction is more forceful because it communicates how the grammatical subject (usually the user in this context) performs an action. In the sentence “To turn the page, click here,” the implied “you” (that is, the user) is the subject, and “click” is the action that the subject performs. Passive constructions show subjects being acted upon rather than performing an action. For example, in the sentence “The page will be turned when this button is clicked,” both “page” and “button” are acted upon.

Research suggests that instructional materials are more effective when they present the goal clause before the action clause, helping readers quickly recognize how the information meets their needs. A goal might be “To turn the page,” “To calculate the result,” or “To apply the style.” For example, the message “To turn the page, click here” starts with a goal statement and then describes the action necessary to fulfill it; users find this more helpful than a purely descriptive message like “This button turns the page.”

If there is more than one way for the user to achieve a goal, mention only the method that involves the item to which the user is pointing. In other words, if the user is pointing to a button, the balloon should tell the user how to use the button, not how to use a keyboard shortcut for that button. For example, a help balloon for a Save button might state, “To save the changes you have made to the settings in the dialog box, click this button”—but the help balloon should *not* add “or press the Return key.”

If there is more than one method for using the item to which the user is pointing, describe the method that’s simplest to explain and understand.

Using Parallel Structure

Use similar syntax for help balloons that describe similar objects. For example, all help balloons that describe buttons should have the same structure. In a style dialog box, you might provide these messages for the buttons: “To see the style, click Apply,” “To implement the style, click OK,” and “To do nothing to change the previous style, click Cancel.”

Users see help balloons provided by many different applications, so a consistent approach within your application helps them to identify types of balloons quickly and to develop realistic expectations about their help messages.

Offering Hints

If there are just a few interesting features in your application that would be difficult to discover, then it’s appropriate to use balloons to call those features to users’ attention.

But if you want to give a hint or shortcut in a balloon, ask yourself these questions:

- n Is the balloon reasonably short, even with the hint?
- n How often will users need the information? If a feature is very obscure and few people will need it, the balloon probably shouldn’t describe it.

Help Manager

- n Are hints and shortcuts available somewhere else—for example, in a “shortcuts” dialog box or a quick-reference card? Not all users will look at balloons. If your program includes many shortcuts and tricks, be sure to list them in other documentation as well.
- n Does the need for hints indicate the need for a different design? If your application contains many hidden shortcuts and features, then you may need to redesign your application to make these features more easily accessible to users.

If you include a hint or shortcut, put the hint at the bottom of the balloon and separate it from the rest of the message by a blank line. For example, the Clean Up Window command in the Finder’s Special menu initially describes the command’s effect: “Neatly arranges the icons in the active window.” Then there is a blank line followed by a hint: “Tip: for other cleanup commands, hold down the Shift or Option key while choosing this command.”

Using Consistent Terminology

You should employ consistent terminology in all your help balloons. Use language that users understand; avoid introducing technical jargon or computer terminology into help balloons. Follow the style and usage standardized by Apple Computer, Inc., in the *Apple Publications Style Guide* (available through APDA) to make the most effective use of the information and vocabulary with which users are already familiar. A supplement to the *Apple Publications Style Guide*, titled “How to Write Balloons,” spells out the guidelines that Apple writers use for the wording and phrasing of help messages. This supplement also provides many examples of clear and useful help messages as well as counterexamples of types of messages to avoid.

Defining the Help Balloon Position

When you provide a help balloon, you specify its help message, the tip of the help balloon, and the variation code for its preferred position. The tip of the help balloon should point to the object that the help balloon describes. You should specify the tip and the variation code so that the help balloon doesn’t obscure the object for which you’re providing help. In most cases, the tip of the help balloon should point to an edge of the object.

You should also consider how the Help Manager repositions the balloon if the variation code places it offscreen. “How the Help Manager Displays Balloons” on page 3-8 describes how the Help Manager repositions the help balloon if necessary.

Specifying the Format for Help Messages

You specify the format for your help messages as text strings within help resources, as text strings within 'STR' resources, as lists of text strings within 'STR#' resources, as styled text using 'TEXT' and 'styl' resources, or as pictures described in 'PICT' resources.

Help Manager

Later sections in this chapter describe all the help resources in detail. Common to all the help resources are the following identifiers, by which you identify the format of your help messages.

Identifier	Help message format
HMStringItem	A text string (a Pascal string stored in the help resource)
HMPictItem	A picture (stored in a 'PICT' resource)
HMStringResItem	A text string (stored in a list of strings as an 'STR#' resource)
HMTEResItem	Styled text (stored in both a 'TEXT' and an 'styl' resource)
HMSTRResItem	A text string (stored in an 'STR ' resource)
HMSkipItem	No help message—skip this item

You specify the identifiers within the help resources; the Help Manager reads these identifiers to determine where and how your help messages are stored. You can use the `HMStringItem` identifier to store Pascal strings directly in a help resource. However, you can make it much easier to localize your product by storing your help messages in separate resources—namely, in 'STR#', 'PICT', 'STR ', and 'TEXT' resources—that can be modified by nonprogrammers using tools like `BalloonWriter` and the `ResEdit` resource editor.

To display a diagram or illustration in 'PICT' format, use the `HMPictItem` identifier. You provide a help message by specifying the resource ID of the 'PICT' resource that contains the diagram or illustration, and the Help Manager displays the picture in a help balloon.

To display a text string stored in a string list ('STR#') resource, use the `HMStringResItem` identifier. You provide a help message by specifying two items in your help resource: the resource ID of an 'STR#' resource, and the index to a particular text string from within that list. For more information on these items, see “Providing Help Balloons for Menus” beginning on page 3-27.

To display styled text, use the `HMTEResItem` identifier. You provide a help message by specifying a resource ID that is common to both a style scrap ('styl') resource and a 'TEXT' resource, and the Help Manager employs `TextEdit` routines to display your text with your prescribed styles. For example, you might create a 'TEXT' resource with resource ID 1000 that contains the words “Displays your text in boldface print” and a 'styl' resource with resource ID 1000 that applies boldface style to the message. (See the chapter “TextEdit” in *Inside Macintosh: Text* for a description of the style scrap.)

To display text from a simple text string ('STR ') resource, use the `HMSTRResItem` identifier. You provide a help message by specifying the resource ID of an 'STR ' resource, and the Help Manager displays the text from that resource in a help balloon. With 'STR ' resources, each text string must be stored in a separate resource. It is usually more convenient to group related help messages in a single 'STR#' resource and use the `HMStringResItem` identifier as previously described.

You can use the `HMSkipItem` identifier for items for which you don't want to provide a help balloon. For example, you specify `HMSkipItem` for the divider lines that appear in menus. (Divider lines cannot have help balloons.)

Specifying Options in Help Resources

Each help resource contains an element that allows you to specify certain options. Notice the options element in the following header component for an 'hmenu' resource.

```
resource 'hmenu' (130, "Edit", purgeable) {
    HelpMgrVersion,      /*version of Help Manager*/
    hmDefaultOptions,   /*options*/
    0,                  /*balloon definition function*/
    0,                  /*variation code*/
```

You should normally use the `hmDefaultOptions` constant, as shown in the preceding example, to get the standard behavior for help balloons. However, you can also use the constants listed here for the options element. (Note that not all options are available for every help resource.)

```
CONST hmDefaultOptions    = 0;  {use defaults}
      hmUseSubID           = 1;  {use subrange resource IDs }
                               { for owned resources}
      hmAbsoluteCoords    = 2;  {ignore coords of window }
                               { origin and treat upper-left }
                               { corner of window as 0,0}
      hmSaveBitsNoWindow  = 4;  {don't create window; save }
                               { bits; no update event}
      hmSaveBitsWindow    = 8;  {save bits behind window }
                               { and generate update event}
      hmMatchInTitle      = 16; {match window by string }
                               { anywhere in title string}
```

If you're providing help balloons for a desk accessory or a driver that uses owned resources, use the `hmUseSubID` constant in the options element. Otherwise, the Help Manager treats the resource IDs specified in the rest of your help resource as standard resource IDs. (See the chapter "Resource Manager" in this book for a discussion of owned resources and their resource IDs.)

As described later in this chapter, you often specify tip and rectangle coordinates in your help resources. When specifying these coordinates within a scrolling window or whenever the window origin is offset from the origin of the port rectangle, you may want to use the `hmAbsoluteCoords` constant. This causes the Help Manager to ignore the local coordinates of the port rectangle when tracking the cursor and instead to track the mouse location relative to the window origin. When you specify the `hmAbsoluteCoords` constant as an option in a help resource, the Help Manager subtracts the coordinates of the window origin from the coordinates of the mouse location and uses the results for the current mouse location, as shown here:

```
mousepoint.h := mousepoint.h - portRect.left;
mousepoint.v := mousepoint.v - portRect.top;
```

Help Manager

With the `hmAbsoluteCoords` constant specified, the Help Manager always assigns coordinates (0,0) to the point in the upper-left corner of the window. So, for example, if the cursor is positioned at point (4,5) in a port rectangle and the window origin is at (3,4), the Help Manager calculates the cursor at (1,1). If this option is not specified, the Help Manager uses the port rectangle's local coordinates when tracking the cursor—for example, when using the `GetMouse` procedure.

The Help Manager draws and removes help balloons in three different ways. For all help resources except 'hmenu' resources, the Help Manager by default draws and removes help balloons as if they were windows. That is, when drawing a balloon, the Help Manager does not save bits behind the balloon, and, when removing the balloon, the Help Manager generates an update event. By specifying the `hmDefaultOptions` constant in your help resources, you always get the standard behavior of help balloons. However, you can often specify two options that change the way balloons are drawn and removed from the screen.

If you specify the `hmSaveBitsNoWindow` constant for the options element, the Help Manager does not create a window for displaying the balloon. Instead, the Help Manager creates a help balloon that is more like a menu than a window. The Help Manager saves the bits behind the balloon when it creates the balloon. When it removes the balloon, the Help Manager restores the bits without generating an update event. You should use this option only in a modal environment where the bits behind the balloon cannot change from the time the balloon is drawn to the time it is removed. For example, you might choose the `hmSaveBitsNoWindow` option in a modal environment when providing help balloons that overlay complex graphics, which might take a long time to redraw with an update event. Note that the Help Manager always uses this behavior when drawing and removing help balloons specified in your 'hmenu' resources. That is, when you specify the `hmDefaultOptions` constant in an 'hmenu' resource, the Help Manager provides this sort of balloon instead of drawing a window for a balloon. (In an 'hmenu' resource, you cannot even specify options for drawing a window for a balloon.)

If you specify the `hmSaveBitsWindow` constant, the Help Manager treats the help balloon as a hybrid having properties of both a menu and a window. That is, the Help Manager saves the bits behind the balloon when it creates the balloon and, when it removes the balloon, it both restores the bits and generates an update event. You'll rarely need this option. It is necessary only in a modal environment that might immediately change to a nonmodal environment—that is, where the bits behind the help balloon are static when the balloon is drawn, but can possibly change before the help balloon is removed. For example, if you use an 'hmenu' resource to provide help balloons for menu titles and menu items, you'll notice that the Help Manager automatically provides this sort of behavior (even when you don't specify the `hmSaveBitsWindow` option) when creating help balloons for menu titles.

In the preceding list of constants, the values for the constants represent bit positions that are set to 1. To override more than one default, add the values of the bit positions for the desired options and specify this sum, instead of a constant, for the options element. For example, to use subrange IDs, ignore the window port origin coordinates, and save bits behind the help balloon without generating an update event, you should add the values

Help Manager

of the bit positions of these options (1, 2, and 4) and specify their sum (7) for the options element.

If you supply the `hmDefaultOptions` constant, the Help Manager treats the resource IDs in this resource as regular resource IDs and not as subrange IDs; it uses the port rectangle's local coordinates when tracking the cursor; and it generally creates windows when drawing balloons and then generates update events without saving or restoring bits when removing balloons.

The `hmMatchInTitle` constant is used only in window help ('hwin') resources to match windows containing a specified number of characters in their titles. This constant is explained in more detail in "Providing Help Balloons for Static Windows" on page 3-65.

The next sections describe how to create help resources that provide help balloons for the standard user interface features of your application.

Providing Help Balloons for Menus

If your application uses the standard menu definition procedure, you'll find that it's easier to provide help balloons for menus than for any of your other interface features. This section is relatively lengthy compared to the sections describing dialog boxes, alert boxes, and windows because it explains in detail much of the work you'll also perform when supplying help balloons for those items.

This section assumes that your application uses the standard menu definition procedure. If your application uses its own menu definition procedure, you must use Help Manager routines to display and remove help balloons. These routines are described in "Displaying and Removing Help Balloons" on page 3-99. Even if you use these routines, you should read this section so that your balloons emulate the behavior that the Help Manager provides for menus using the standard menu definition function.

To create help balloons for a menu—pull-down, pop-up, or hierarchical—that uses the standard menu definition procedure, create a resource of type 'hmenu' in which you specify help balloons for the menu title and for each menu item. You create a separate 'hmenu' resource for each menu.

Note

BalloonWriter, available from APDA, is a tool that gives nonprogrammers an easy, intuitive way to create help balloons for menus. BalloonWriter creates 'hmenu' resources as appropriate and places them in the resource file of your application; BalloonWriter likewise creates and stores 'STR', 'STR#', and 'TEXT' resources that contain the help messages authored by nonprogramming writers. For menus that use custom menu definition procedures, nonprogrammers can use BalloonWriter to provide you with delimited ASCII text that you can then use in conjunction with `HMShowBalloon` to display the desired help balloons. u

Help Manager

The Help Manager can display different help balloons for the various states of a menu item. Each menu item can have up to four help balloons associated with it, one for each state:

- n enabled
- n disabled (that is, dimmed)
- n enabled and checked
- n enabled and marked (that is, marked by a symbol other than a checkmark—for example, a bullet or a diamond)

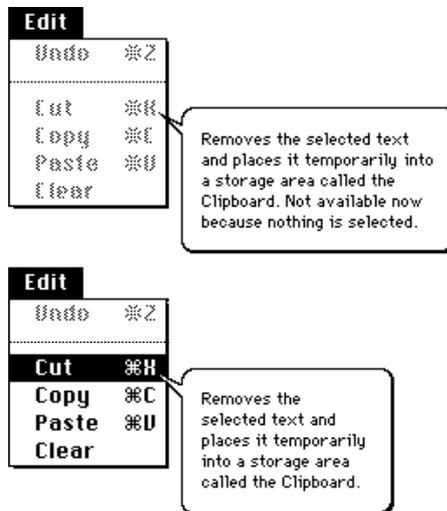
For example, you can define a help balloon that the Help Manager displays when the Cut command is enabled and another help balloon for display when the Cut command is dimmed. Remember that the help balloon you provide for a dimmed menu item should explain why it isn't currently available or, if more appropriate, how to make it available.

Note

Although `enabled` and `disabled` are the constants you use in a resource file to display or dim menus and menu items, you shouldn't use these terms in your help balloons or user guides. Rather, use the term *menus*, *menu commands*, or *menu items* for those that are enabled, and use the term *not available* or *dimmed* to distinguish those that have been disabled. u

When your application calls the Menu Manager function `MenuSelect` or `MenuKey`, the Menu Manager automatically tracks the cursor, highlights enabled menu items, and displays any additional hierarchical or pop-up menus as the user moves the mouse. As the user drags the cursor across or through a menu, the Menu Manager uses the Help Manager to display any help balloons associated with the current state of the menu title or menu item.

If there is sufficient memory, the standard menu definition procedure saves the bits behind the help balloon and restores these bits for quick updating of the screen. If there isn't sufficient memory to save the bits behind the help balloon, then—as with menus—the procedure generates appropriate update events. Figure 3-9 shows help balloons for two instances of a menu, one with the Cut command dimmed, the other with the Cut command enabled.

Figure 3-9 Help balloons for different states of the Cut command

You don't specify hot rectangles or tip coordinates for menus. The rectangles defined by the Menu Manager for menu titles and menu items are used as hot rectangles. The Help Manager initially tries to draw a help balloon for a menu item using variation code 0 (shown in Figure 3-4 on page 3-10) with the tip placed 8 pixels inside the right edge and halfway between the top and bottom edges of the menu item's rectangle. If the balloon's initial position lies wholly or partially offscreen, the Help Manager tries to redraw the balloon by moving its tip to the left edge of the item's rectangle and using variation code 3. The Help Manager uses variation codes 1 and 2 in its attempts to draw help balloons for menu titles. The Help Manager never moves the tip for menu titles; instead, the tip is always located just below the bottom of the menu bar at the midpoint of the menu title's text.

The resource ID of each 'hmenu' resource should match the corresponding menu ID. For example, to provide help balloons for a menu with ID 130, create an 'hmenu' resource with resource ID 130.

The 'hmenu' resource contains four types of components, listed below. Each component consists of several elements that contain information used by the Help Manager.

- n The **header component** is where you specify information that applies to all help balloons specified in this resource—information such as the version number of the Help Manager, the balloon definition function, and the variation code.
- n The **missing-items component** is where you specify help messages for any menu items missing from or unspecified in the rest of the resource. This is useful, for example, for allowing several menu items to share the same help message.

Help Manager

- n The **menu-title component** is where you specify help messages for the menu title.
- n A **menu-item component** is where you specify the help messages for a particular menu item. You can include any number of menu-item components; however, the menu-item components in the 'hmenu' resource must appear in the order in which their corresponding menu items appear in the menu. If you do not provide menu-item components for any items at the bottom of a menu, a help message from the missing-items component is used.

Here is the general Rez input format of an 'hmenu' resource. (Rez is the resource compiler provided with Apple's Macintosh Programming Workshop [MPW], available from APDA.)

Component	Element
Header	Help Manager version
	Options
	Balloon definition function
	Variation code
Missing item	Identifier
	Help message for missing enabled items
	Help message for missing items dimmed by application
	Help message for missing enabled-and-checked items
	Help message for missing enabled-and-marked items
Menu title	Identifier
	Help message for enabled menu title
	Help message for menu title dimmed by application
	Help message for menu title dimmed by system software
	Help message for menu items dimmed by system software
First menu item	Identifier
	Help message for enabled menu item
	Help message for menu item dimmed by your application
	Help message for enabled-and-checked menu item
	Help message for enabled-and-marked menu item
Next menu item	(Same as for first menu item)
	.
	.
	.
Last menu item	(Same as for first menu item)

Listing 3-1 shows Rez input code for the 'hmenu' resource for an Edit menu.

Listing 3-1 Rez input for a partial 'hmenu' resource

```

resource 'hmenu' (130, "Edit", purgeable) {
    /*header component*/
    HelpMgrVersion,
    hmDefaultOptions,          /*options*/
    0,                          /*balloon definition function*/
    0,                          /*variation code*/
    /*missing-items component*/
    HMSkipItem {
        /*no missing items, so skip to menu-title component*/
    },
    { /*menu-title component*/
        HMStringItem { /*use following P-strings*/
            /*use string below when menu is enabled*/
            "Edit menu\n\nUse this menu to manipulate text.",
            /*use string below when app dims menu*/
            "Edit menu\n\nUse this menu to manipulate text. "
                "Not available because you do not have permission "
                "to alter this file.",
            /*use string below for title dimmed by system */
            /* software for an alert or modal dialog box*/
            "Edit menu\n\nUse this menu to manipulate text. "
                "Not available because a dialog box is on "
                "the screen.",
            /*use string below for all items when system */
            /* software dims them for an alert or modal */
            /* dialog box*/
            "This item is not available because a dialog box "
                "is on the screen.",
        },

        /*first menu-item component: Undo command*/
        HMStringItem { /*use following P-strings*/
            /*use string below when command is enabled*/
            "Cancels your last edit.",
            /*use string below when app dims the command*/
            "Cancels your last edit. Not available because "
                "you haven't performed an editing action yet.",
            /*can't check the item, so empty string goes below*/
            "",
            /*can't mark the item, so empty string goes below*/
            "",
        },
    },
}

```

Help Manager

```

    /*second menu-item component: divider line*/
    HMSkipItem { /*no help balloons for divider lines*/
    },
    /*third menu-item component: Cut command*/
    HMStringItem { /*use following P-strings*/
        /*use string below when command is enabled*/
        "Cuts the selected text to the Clipboard.",
        /*use string below when app dims the command*/
        "Cuts the selected text to the Clipboard. "
        "Not available now because no text is selected.",
        /*can't check item, so empty string goes below*/
        "",
        /*can't mark item, so empty string goes below*/
        "",
    }
    /*menu-item components for Copy, Paste, and Clear */
    /* commands go here*/
}
};

```

Specifying Header Information for the 'hmenu' Resource

The header component of an 'hmenu' resource consists of these elements:

1. Help Manager version.
2. Options.
3. Balloon definition function.
4. Variation code.

Always specify the `HelpMgrVersion` constant for the Help Manager version element.

For the options element, you must specify the constant `hmDefaultOptions`.

The third element in the header component specifies the resource ID of the window definition function that is used to draw the frame of the help balloon. To use the standard balloon definition function, specify 0 for this element; this is the suggested default. If you use your own balloon definition function (as described in “Writing Your Own Balloon Definition Function” on page 3-93), specify its resource ID for this element.

Help Manager

The fourth element in the header component specifies the preferred position of the help balloon. For example, the standard balloon definition function displays help balloons according to eight different positions. If you specified the standard balloon definition for the preceding element, supply a variation code from 0 to 7 to display the balloon according to one of the eight positions shown in Figure 3-4 on page 3-10. The preferred variation code is 0. If you are unsure of which variation code to use, specify 0; the Help Manager will use a different variant if another is more appropriate. If you use your own balloon definition function, you specify its variation code for this element of the header component.

Specifying Help for Menu Items Missing From the Resource

After the header component, you specify the format and help messages for help balloons for missing items, for the menu title, and for the menu items.

Use the missing-items component of the 'hmenu' resource to specify how the Help Manager handles menu items that are not described in this resource. You can also use the missing-items component to supply help messages for menu items that are described in the 'hmenu' resource but that lack help messages for any particular states.

The missing-items component of this resource is useful when you have menu items with similar characteristics or when the number of menu items is variable. For example, if the help message for a dimmed item applies to all dimmed menu items, you can specify a help message once in the third element of the missing-items component instead of repeating it in every third element of the various menu-item components.

The missing-items component consists of the following five elements:

1. An identifier (either `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`) for the format of the help messages.
2. The help message when a menu item is enabled. This message is displayed either when the item itself is not specified in a menu-item component of this 'hmenu' resource or when its help message is specified in a menu-item component, but specified with either an empty string or a resource ID of 0.
3. The help message when your application dims the menu item. This message is displayed either when the item itself is not specified in a menu-item component of this 'hmenu' resource or when its help message is specified in a menu-item component, but specified with either an empty string or a resource ID of 0.
4. The help message when a menu item is enabled and checked. This message is displayed either when the item itself is not specified in a menu-item component of this 'hmenu' resource or when its help message is specified in a menu-item component, but specified with either an empty string or a resource ID of 0.
5. The help message when a menu item is enabled and marked (with a character other than a checkmark). This message is displayed either when the item itself is not specified in a menu-item component of this 'hmenu' resource or when its help message is specified in a menu-item component, but specified with either an empty string or a resource ID of 0.

Help Manager

For missing items (as for the rest of the items listed in an 'hmenu' resource), you store the help messages in text strings within this resource or in separate 'STR', 'STR#', 'PICT', or 'TEXT' and 'styl' resources. For the first element in the missing-items component, use one of the identifiers described in “Specifying the Format for Help Messages” on page 3-23. These identifiers indicate how and where you store your help messages. Then, depending on the identifier you specify, for the next four elements supply either text strings for help messages or resource IDs of resources that contain help messages.

There are two additional identifiers that you can specify for menu items in 'hmenu' resources. These identifiers are explained in “Specifying Help for a Changing Menu Item” on page 3-43 and in “Specifying Resources by Item Name” on page 3-45.

Identifier	Purpose
HMCompareItem	The Help Manager displays help for the current menu item only when it matches a specified string.
HMNamedResourceItem	The Help Manager displays the help message from the resource that has the same name as the current menu item.

Listing 3-2 on page 3-35 illustrates the help resource for a menu titled Colors. Notice in the missing-items component that the element describing dimmed states for menu items has the message “Not available; either you have not selected text to color, or your monitor does not support color.” Because this resource doesn’t specify a message for any individual command’s dimmed state, this message appears in help balloons for the Blue, Green, and Red commands whenever the application disables them. If there are many reasons why your application may have dimmed an item, don’t name them all. Instead, describe one or two of the most likely reasons.

Note

As described in the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*, system software automatically dims your application’s menus as appropriate whenever an alert box or a fixed-position modal dialog box appears on the screen. You supply help messages for menu titles and menu items dimmed by system software in the third and fourth elements of the menu-title component of the 'hmenu' resource, as described in the next section. If your application uses *movable* modal dialog boxes or modeless dialog boxes, your application must dim its menus as appropriate and provide an alternate 'hmenu' resource for this state, as described in “Providing Help Balloons for Menus You Disable for Dialog Boxes” beginning on page 3-47. u

Listing 3-2 Rez input for the missing-items component of an 'hmenu' resource

```

resource 'hmenu' (132, "Colors", purgeable) {
    /*header component*/
    HelpMgrVersion, hmDefaultOptions, 0, 0,
    /*missing-items component*/
    HMStringItem {
        "", /*no missing enabled items*/
        /*help messages for all items that app dims are below*/
        "Not available; either you have not selected "
            "text to color, or your monitor does "
            "not support color.",
        "", /*no missing enabled-and-checked items*/
        "", /*no missing enabled-and-marked items*/
    },
    { /*menu-title component*/
        HMStringItem { /*use these P-strings for help messages*/
            /*use string below when menu is enabled*/
            "Colors menu\n\nUse this menu to display text in color.",
            /*use string below when app dims menu title*/
            "Colors menu\n\nUse this menu to display text in color."
                "Not available because this monitor does not
                support color.",
            /*use string below when system software dims menu */
            /* title for an alert or modal dialog box*/
            "Colors menu\n\nUse this menu to display text in color. "
                "Not available because a dialog box is on the "
                "screen.",
            /*use string below for all items when system dims */
            /* them for alert and modal dialog boxes*/
            "Colors your selected text. This item is not "
                "available because a dialog box is on the screen.",
        },
        /*first menu-item component: Blue command*/
        HMStringItem { /*use these P-strings for help messages*/
            /*use string below when command is enabled*/
            "Displays the selected text in blue.",
            "", /*use missing-items help when app dims menu*/
            "", /*can't check command, so use empty string here*/
            "", /*can't mark command, so use empty string here*/
        },
    },
}

```

Help Manager

```

    /*second menu-item component: Green command*/
    HMStringItem { /*use these P-strings for help messages*/
        /*use string below when command is enabled*/
        "Displays the selected text in green.",
        "", /*use missing-items help when app dims menu*/
        "", /*can't check command, so use empty string here*/
        "", /*can't mark command, so use empty string here*/
    },
    /*third menu-item component: Red command*/
    HMStringItem { /*use these P-strings for help messages*/
        /*use string below when command is enabled*/
        "Displays the selected text in red.",
        "", /*use missing-items help when app dims menu*/
        "", /*can't check command, so use empty string here*/
        "", /*can't mark command, so use empty string here*/
    }
}
};

```

Specifying Help for Menu Titles and for Items Dimmed by System Software

After the missing-items component, use the menu-title component to specify the help messages for the menu title and for menu items dimmed by system software. The menu-title component consists of the following five elements:

1. An identifier (either `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`) that indicates the format of the help messages.
2. The help message for the menu title when the menu title is enabled.
3. The help message for the menu title when your application dims the menu title.
4. The help message for the menu title when system software dims the menu title at the appearance of an alert box or a modal dialog box.
5. The help message for all menu items when system software dims them at the appearance of an alert box or a modal dialog box.

As in the missing-items component, use the first element in the menu-title component to specify an identifier that describes the format for the help messages. Depending on the identifier you specify, for the other elements you supply either text strings for the help messages or the resource IDs of resources that contain the help messages. The second, third, and fourth elements correspond to states of the menu title; the fifth element corresponds to a state of all the menu items.

Help Manager

Use the second element of the menu-title component to specify a help message that describes the purpose of the menu when it's enabled. For menus in the menu bar, the beginning of the message should name the menu and then concisely describe what kinds of commands are in the menu, as shown in Figure 3-10. (Listing 3-2 on page 3-35 specifies the menu title—"Colors menu"—and then includes the special characters `\n\n` to specify two new lines in a Rez input file before specifying a description of the menu itself.)

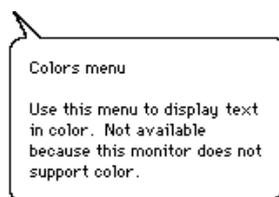
Figure 3-10 A help balloon for an enabled menu title



Because some pull-down menus in the menu bar are identified by icons, not words, offer additional clarification in your help balloon by always providing a name for pull-down menus. For pop-up menus, simply describe what the user does with the menu; don't give the menu a name.

Use the third element of the menu-title component to specify a help message that identifies the menu, describes what it does, and then describes why your application has dimmed the menu title. As much as possible, repeat the text that you use for the title's enabled state, then describe why it is not enabled. See Figure 3-11 for an example and see Listing 3-2 on page 3-35 for the Rez input that specifies the help message for the help balloon shown in the figure.

Figure 3-11 A help balloon for a dimmed menu title

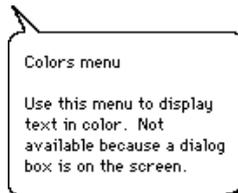


In general, you should use the phrase "Not available because" to introduce your explanation of a dimmed title. If there are several reasons why a menu title might be dimmed, don't name them all. Instead, describe one or two of the most likely reasons.

Help Manager

Use the fourth element of the menu-title component to specify a help message that describes why system software has dimmed the menu title—that is, because the user must respond to an alert box or modal dialog box on the screen. Figure 3-12 illustrates an appropriate help balloon for this situation.

Figure 3-12 A help balloon for a menu title dimmed by the Dialog Manager



Starting with system software version 7.0, users have been able to use selected menus while the screen displays an alert box or a modal dialog box. For example, the Show Balloons (or Hide Balloons) command is always available from the Help menu so that users can see your help balloons for the modal dialog box or alert box. While some menus are accessible (in particular, the Help, Keyboard, and—when appropriate—Edit menus), others aren't. The chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* describes the circumstances under which menus are enabled or disabled when alert boxes and dialog boxes are displayed.

Note

If your application uses *movable* modal dialog boxes, you must dim your menus and provide an alternate 'hmenu' resource for this state, as described in “Providing Help Balloons for Menus You Disable for Dialog Boxes” beginning on page 3-47. u

Use the fifth element to specify the help message for all menu items whenever system software dims them because of an alert box or a modal dialog box. Because this message is used for all items in the menu, it needs to be somewhat general, as shown in Figure 3-13.

Figure 3-13 A help balloon for menu items dimmed by the Dialog Manager



Specifying Help for Menu Items

After you create the header component, the missing-items component, and the menu-title component, you specify help messages in a menu-item component for each menu item. The menu-item components in the 'hmenu' resource must appear in the order in which their corresponding menu items appear in the menu. (Because of this, if you have menu items that your application can add while it is running, you should add these dynamic items to the end of the menu to simplify your implementation of help balloons for your nondynamic items.)

The menu-item component consists of the following five elements:

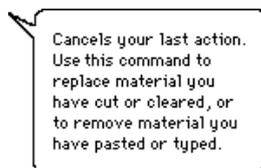
1. An identifier (either `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`) that indicates the format of the help messages.
2. The help message for the menu item when the item is enabled.
3. The help message for the menu item when your application dims the item.
4. The help message for the menu item when the item is enabled and checked.
5. The help message for the menu item when the item is enabled and marked with a character other than a checkmark.

For the first element of each menu-item component, supply an identifier to describe the format of the help messages in that component. Then, depending on the identifier you specify, use the other elements of the menu-item component to supply either text strings for the help messages or the resource IDs of resources that contain the help messages.

You can use the `HMSkipItem` identifier for items that appear in your menu but for which you don't provide a help balloon. For example, you can specify `HMSkipItem` for divider lines that appear in menus. (Divider lines cannot have help balloons.) If you specify `HMSkipItem`, the Help Manager does not display help balloons for that menu item, even if the missing-items component specifies a help message.

For the second element, specify a help message that describes what the item usually does. Don't name the menu item. Begin with a verb describing what happens when the user chooses the item. For example, the help balloon for the Undo command in an Edit menu should contain something similar to the information in Figure 3-14.

Figure 3-14 A help balloon for a menu item

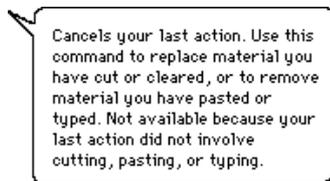


Help Manager

For menu items that display a dialog box, it is usually unnecessary to state that a dialog box will appear. The fact that a menu item displays a dialog box is not what the user wants to know; the user wants to know what choosing the menu item ultimately accomplishes.

For the third element, specify a help message that describes what the menu item does and why your application has dimmed the item. As much as possible, repeat the text that you use for the item's enabled state, and then describe why it is not enabled. In general, you should use the phrase "Not available because" to introduce your explanation of the dimmed item. If there are multiple reasons why an item might be dimmed, don't name them all. Instead, describe one or two of the most likely reasons. Figure 3-15 gives an example of a help message for a dimmed Undo command.

Figure 3-15 A help balloon for a dimmed menu item



If your application checks or otherwise marks a menu item, use the fourth and fifth elements of that item's component in the 'hmenu' resource to describe the special condition indicated by that state. As with dimmed states, try to repeat the text that you use for the title's enabled state, and then describe why it is checked or marked. If there are multiple reasons why an item might be checked or marked, don't name them all. Instead, describe one or two of the most likely reasons.

Note that, for any component in the resource, you can specify only one format for all of its help messages. For example, if you specify the `HMSTRResItem` identifier in a menu-item component for the Undo command, you must store all help messages specified in that component in 'STR' resources. (However, if you specify a resource ID of 0 or an empty string as the help message of any item in order to use the help message from the missing-items component, the help message follows the format specified in the missing-items component.)

You do not have to provide a help message for every state of a menu item. If you do not provide a help message for a particular state, the Help Manager uses the help message specified in the missing-items component. If the missing-items component does not specify a help message for that state either, the Help Manager does not display a help balloon.

Help Manager

Listing 3-3 shows a sample 'hmenu' resource for another Edit menu.

Although Listing 3-1 and Listing 3-2 illustrate 'hmenu' resources that contain their own Pascal-string help messages, you should keep your help messages in separate, more easily localized resources. The 'hmenu' resource in Listing 3-3 stores its help messages in a separate 'STR#' resource (which is given a corresponding resource ID of 130 for easier maintenance).

Listing 3-3 Rez input for corresponding 'hmenu' and 'STR#' resources

```
resource 'hmenu' (130, "Edit menu help", purgeable) {
    HelpMgrVersion, 0, 0, 0, /*standard header component*/
    HMSkipItem { /*missing-items component*/
        /*no missing items, so skip to menu-title component*/
    },
    { /*menu title and items below*/
        /*menu-title component*/
        HMStringResItem { /*use an 'STR#' for help messages*/
            130,1, /*'STR#' res ID, index when menu is enabled*/
            130,2, /*'STR#' res ID, index when app dims menu*/
            130,3, /*'STR#', index for title that system */
                /* software dims for all alert and modal */
                /* dialog boxes*/
            130,4 /*'STR#', index for items that system */
                /* software dims for all alert and modal */
                /* dialog boxes*/
        },
        /*first menu-item component: Undo command*/
        HMStringResItem { /*use 'STR#' resource for help messages*/
            130,5, /*'STR#' res ID, index when item is enabled*/
            130,6, /*'STR#' res ID, index when item is dimmed*/
            0,0, /*can't check command*/
            0,0 /*can't mark command*/
        },
        /*second menu-item component: divider line*/
        HMSkipItem { /*no balloon help for divider lines*/
        },
        /*third menu-item component: Cut command*/
        HMStringResItem { /*use an 'STR#' for help messages*/
            130,7, /*'STR#' res ID, index when item is enabled*/
            130,8, /*'STR#' res ID, index when app dims item*/
            0,0, /*can't check command*/
            0,0 /*can't mark command*/
        },
    },
}
```

Help Manager

```

        /*menu-item component for Copy command goes here*/
    }
};

resource 'STR#' (130, "Edit menu help strings") {
    /*help text for Edit menu*/
    { /*array StringArray: 17 elements*/
        /*[1] help text for enabled Edit menu title*/
        "Use this menu to cancel your last action, to manipulate "
            "text, to select the entire content of a document, "
            "and to show what's on the Clipboard.";
        /*[2] help text for Edit menu title dimmed by app*/
        "Use this menu to cancel your last action, to manipulate "
            "text, to select the entire content of a document, "
            "and to show what's on the Clipboard. Not "
            "available now.";
        /*[3] help text for Edit menu title dimmed by system */
        /* software for all alert and modal dialog boxes */
        /* that don't contain editable text items*/
        "Use this menu to cancel your last action, to manipulate "
            "text, to select the entire content of a document, and "
            "to show what's on the Clipboard. Not available "
            "because a dialog box is on the screen.";
        /*[4] help for Edit menu items that system software dims */
        /* for all alert and modal dialog boxes */
        /* that don't contain editable text items*/
        "Not available because a dialog box is on the screen.";
        /*[5] help text for enabled Undo command*/
        "Cancels your last action. Use this command to replace "
            "material you have cut or cleared, or to remove material "
            "you have pasted or typed.";
        /*[6] help text for Undo command dimmed by app*/
        "Cancels your last action. Use this command to replace "
            "material you have cut or cleared, or to remove material "
            "you have pasted or typed. Not available because your "
            "last action did not involve cutting, pasting, "
            "or typing.";
        /*help text for all other commands goes here*/
    }
};

```

Help Manager

The 'hmnu' resource in Listing 3-3 specifies the standard balloon definition function and variation code in the third and fourth elements of the header component. The missing-items component is specified using the `HMSkipItem` identifier, meaning that this 'hmnu' resource does not provide any help balloons for menu items that are missing from this resource or that do not have help messages specified for any states.

Following the menu-title component, the menu-item components for the menu items are listed in the order in which the items appear in the menu. For menu-item components that do not specify information for a particular state, the Help Manager normally uses the information from the missing-items component. However, in Listing 3-3 the 'hmnu' resource does not specify a help message in the missing-items component. Instead, all help messages are specified in each menu-item component in this resource. Because there are no enabled-and-checked or enabled-and-marked states for the Undo and Copy commands, these states are specified with resource IDs of 0.

As described in the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*, system software does not dim your application’s Edit menu when you display a dialog box that contains editable text items. Listing 3-3 nevertheless provides help messages for a dimmed Edit menu for those instances when the application displays alert boxes and dialog boxes that do not contain editable text items.

Specifying Help for a Changing Menu Item

If you have a menu item that changes names, you can use the `HMCompareItem` identifier to compare a string against the current menu item in that position. If the string specified after the `HMCompareItem` identifier matches the name of the current menu item, the Help Manager displays the help messages specified in the next four elements of the 'hmnu' resource. Because of performance considerations, the `HMCompareItem` identifier shouldn't be used unless necessary.

A menu-item component that uses the `HMCompareItem` identifier uses a different format than other menu-item components do. Here are the seven elements you use for specifying help in an 'hmnu' resource for a changing menu item.

1. The `HMCompareItem` identifier.
2. The string to compare against current menu item.
3. The identifier (either `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`) that indicates the format of the help messages.
4. The help message for the menu item when the item is enabled.
5. The help message for the menu item when the item is dimmed.
6. The help message for the menu item when the item is enabled and checked.
7. The help message for the menu item when the item is enabled and marked.

Help Manager

Create a component that uses the `HMCompareItem` identifier for every item name that can appear in a particular menu position. For example, Listing 3-4 shows an `'hmenu'` resource for a menu command that toggles between Show Colors and Hide Colors.

Note

It is important to provide components for all possible strings that can appear in the changing item. Also note that if you use the `HMCompareItem` identifier for a menu item, you should ensure that the following menu item has a text string that is different from the one that the `HMCompareItem` menu-item component compares against. u

Listing 3-4 Rez input for an `'hmenu'` resource that uses `HMCompareItem` for a changing menu item

```
resource 'hmenu' (132, "Colors menu help", purgeable) {
    /*see Listing 3-2 for missing-items example*/
    /*see Listing 3-2 for Colors menu title's help example*/
    HMCompareItem { /*help message if first command is */
        /* called Show Colors*/
        "Show Colors",
        HMStringResItem {
            132, 1, /*enabled*/
            0, 0, /*use missing items*/
            0, 0, /*item can't be checked*/
            0, 0 /*no marked state*/
        },
    },
    HMCompareItem { /*help if the first command is */
        /* called Hide Colors*/
        "Hide Colors",
        HMStringResItem {
            132, 2, /*enabled*/
            0, 0, /*use missing items*/
            0, 0, /*item can't be checked*/
            0, 0 /*no marked state*/
        },
    },
    /*Blue command's help messages*/
    HMStringItem { /*use these P-strings for help messages*/
        /*use string below when command is enabled*/
        "Displays the selected text in blue.",
        "", /*use missing-items help when menu is dimmed*/
        "", /*can't check command--use empty string here*/
        "", /*can't mark command--use empty string here*/
    }
}
```

Help Manager

```

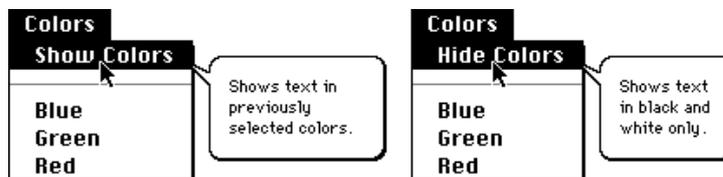
    },
    /*see Listing 3-2 for other commands' examples*/
}
};

resource 'STR#' (132, "Hide & Show Colors commands help text") {
    {
        /*[1] help text for enabled Show command*/
        "Shows text in previously selected colors.";
        /*[2] help text for enabled Hide command*/
        "Shows text in black and white only.";
    }
};

```

As illustrated in Figure 3-16, when the menu command is Show Colors, the Help Manager displays the help message described by the first `HMCompareItem` component. When the menu command is Hide Colors, the Help Manager displays the help message described by the second `HMCompareItem` identifier.

Figure 3-16 Help balloons for a changing menu item



Specifying Resources by Item Name

You can also specify help messages in a component with the `HMNamedResourceItem` identifier, which causes the Help Manager to use a resource whose name matches the name and state of the current menu item. A menu-item component that uses the `HMNamedResourceItem` identifier uses a different format than the other menu-item components do. Here are the two elements you use for specifying named resources as help messages in an 'hmenu' resource.

1. The `HMNamedResourceItem` identifier.
2. A resource type (either 'STR', 'PICT', or, for text, 'TEXT') that contains the help messages for the current menu item. If you specify 'TEXT', you also get style information for the 'TEXT' resource by creating a resource of type 'styl' with the same name.

Help Manager

To provide help for a menu item when it is enabled, create a resource with the same name as the menu item, then put the help message for the enabled menu item in this resource. The Help Manager uses the `GetNamedResource` function to find the resource—of the type specified in the second element of the menu-item component—that has the same name as the current menu item.

To provide help for a menu item when it is dim, create a resource with a name consisting of the menu item and an exclamation point (!), then put the help message for the dimmed menu item in this resource. When the menu item is dimmed, the Help Manager appends an exclamation point to the name of the menu item and searches for a resource by that name. Similarly, if a menu item is enabled and marked with a checkmark or other mark, the Help Manager appends the mark to the name of the current menu item and looks for a resource by that name.

For example, the 'hmnu' resource in Listing 3-5 specifies that the Help Manager extracts the help message from a resource named `Red` of type 'STR' when displaying a help balloon for an enabled menu command named `Red`. If the menu item is dimmed, the Help Manager gets the 'STR' resource with the name `Red!` and uses its text string for the help message. If the `Red` command could be marked with an asterisk (*), the Help Manager would search for the resource with the name `Red*` of type 'STR'.

Listing 3-5 Rez input for specifying help messages with named resources

```
resource 'hmnu' (132, "Colors menu help", purgeable) {
    /*see Listing 3-2 for header, missing-items, */
    /* menu-title, and menu-item components*/
    HMNamedResourceItem { /*Red command's help message*/
        'STR' /*use the 'STR' resource named "Red"*/
    }
}
};

resource 'STR' (333, "Red") { /*help text for enabled */
    /* Red command*/
    "Displays the selected text in red."
};

resource 'STR' (334, "Red!") { /*help text for dimmed */
    /* Red command*/
    "Not available; either you have not selected text to "
    "color, or your monitor does not support color.",
};
```

Providing Help Balloons for Menus You Disable for 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 dim your menu titles and menu items as appropriate. As described in “Specifying Help for Menu Titles and for Items Dimmed by System Software” beginning on page 3-36, you specify help balloons for these dimmed menu titles and menu items in the fourth and fifth elements of your 'hmenu' resources' menu-title components.

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:

- n Use the Menu Manager function `DisableItem` to 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.
- n Use the Menu Manager function `DisableItem` to disable all of your application's menus except the Edit menu, as well as any inappropriate commands in the Edit menu.
- n Use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items.
- n Provide your own code for supporting the Undo command.
- n Use the Menu Manager function `EnableItem` to 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.

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.

Help Manager

When creating a modeless dialog box, your application should perform the following tasks:

- n Use the Menu Manager function `DisableItem` to disable only those menus whose commands are invalid in the current context.
- n 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.
- n Enable your application's items in the Help menu, as appropriate. (System software disables all items except the Hide Balloons/Show Balloons command.)

When your application creates a movable modal dialog box, it should perform the following tasks:

- n Leave the Apple menu enabled so that the user can open other applications with it.
- n 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.
- n Use the Menu Manager function `DisableItem` to disable all of your other menus.

See the chapters "Menu Manager" and "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about menus, alert boxes, and dialog boxes.

When you use the Menu Manager function `DisableItem` to dim your menus, the Help Manager does not know that you have dimmed them because you're displaying a dialog box; instead, the Help Manager assumes that you've dimmed them for some other reason. Whenever you dim your own menus—for whatever reason—the Help Manager always uses the second element of your menu-title component and the second elements of your menu-item components.

To provide help messages that explain that the menu and its items are dim because your application dimmed them to display a dialog box, you must use an alternate 'hmnu' resource.

For example, if an application were to allow only one document at a time to be opened, it would dim the New command in the File menu whenever a document were open. The second element of that item's component specifies a help message similar to this.

```
/*help message for dimmed New command*/
"Opens a new SurfWriter document called \"Untitled\". "
"Not available now because a SurfWriter document is "
"already open. (SurfWriter can open only one document "
"at a time.)";
```

This is not an appropriate message for the item's help balloon when the application displays a modal dialog box that contains an editable text item—but unless the application changes the 'hmnu' resource for its File menu, this is the message that the Help Manager displays.

Help Manager

To handle those menus that you dim for dialog boxes, your application must use alternate 'hmenu' resources. In an alternate 'hmenu' resource, use the second element of the missing-items component and the second element of the menu-title component to specify help balloons for the menu's dimmed title and all of its dimmed items, as shown in Listing 3-6.

Listing 3-6 Specifying an alternate 'hmenu' resource for a menu that your application disables when it displays movable modal dialog boxes

```
resource 'hmenu' (kFileHelpID, purgeable)
{ /*use this when my application dims the menu to display */
  /* a modal dialog box with editable text items*/
  /*header component*/
  HelpMgrVersion, hmDefaultOptions, 0, 0,
  /*missing-items component*/
  HMStringResItem {
    0, 0,          /*missing enabled items: not applicable */
                  /* because they're all dim*/
    256, 1,        /*use this help string for all dimmed */
                  /* menu items--they're all missing from */
                  /* this resource*/
    0, 0,          /*missing enabled-and-checked items: not */
                  /* applicable because they'd be dimmed*/
    0, 0,          /*missing enabled-and-marked items: not */
                  /* applicable because they'd be dimmed*/
  },
  /*menu-title component*/
  { /*File menu title help when dimmed for a movable modal*/
    HMStringResItem {
      0, 0,          /*no enabled title: it's dimmed*/
      256, 2,        /*use this help string for menu title */
                  /* dimmed for a movable modal dialog*/
      0, 0,          /*Help Manager doesn't look here for */
                  /* movable modal dialogs*/
      0, 0,          /*Help Manager doesn't look here for*/
                  /* movable modal dialogs*/
    },
  } /*use missing-items info for all dimmed menu items*/
};

resource 'STR#' (256, "help messages for dimmed menus") {
  /*use these when my application dims menus to show a */
  /* modal dialog box*/
  {
    /*[1] Dimmed items help text*/
```

Help Manager

```

    "Not available now because a dialog box is on "
    "the screen.".
    /*[2] help message for dimmed File menu title*/
    "File menu\n\nUse this menu to open, close, save, and print "
    "SurfWriter documents, and to quit SurfWriter. "
    "Not available because a dialog box is on the screen.";
    /*[3] help message for dimmed Tools menu title*/
    "Tools menu\n\nUse this menu to ... " /*more text goes here*/
    "Not available because a dialog box is on the screen.";
    /*help messages for other dimmed menu titles go here*/
};

```

Use the `HMSetMenuResID` function to associate alternate 'hmenu' resources with your menus whenever your application displays a movable modal dialog box. Listing 3-7 illustrates how an application disables its menus and then reassigns them appropriately, using alternate 'hmenu' resources before displaying a dialog box.

Listing 3-7 Reassigning 'hmenu' resources before displaying a movable modal dialog box

```

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);
            END
    END

```

Help Manager

```

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: }
    { diable all menus except Apple, }
    { call MyAdjustEditMenuForModalDialogs for editable text items}
END; {of CASE}
END;

```

The `MyAdjustMenusForDialogs` routine in Listing 3-7 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 adjust the items in the Edit menu, `MyAdjustMenusForDialogs` calls another application-defined routine, `MyAdjustEditMenuForModalDialogs`. The `MyAdjustEditMenuForModalDialogs` routine, which is not shown in this volume, 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.

After removing a dialog box from the screen, enable the appropriate menus again and use the `HMSetMenuResID` function to reassociate your original help balloons with the reenabled menus. (You can pass `-1` to the `HMSetMenuResID` function to remap menus to their previous 'hmnu' resources.)

Providing Help Balloons for Items in Dialog Boxes and Alert Boxes

For dialog boxes and alert boxes defined with item list ('DITL') resources, you can provide help balloons for individual items in the dialog box or alert box by supplying a resource of type 'hdlg' (dialog-item help). When an item has a help balloon associated with it, the Help Manager automatically displays and removes the help balloon as the user moves the cursor into and out of the item's display rectangle. The Help Manager

Help Manager

can display different help balloons for various states of an item—by highlight value if the item is a control, and by enabled and disabled states for items that are not controls.

Note

BalloonWriter, available from APDA, gives nonprogrammers an easy, intuitive way to create help balloons for dialog and alert boxes. **BalloonWriter** creates 'hdlg' resources as appropriate and places them in the resource file of your application; **BalloonWriter** likewise creates and stores 'STR ', 'STR#', and 'TEXT' resources that contain the help messages authored by nonprogramming writers. u

You can also provide help balloons for other areas of a dialog box or alert box using the 'hwin' (window help) resource as described in “Providing Help Balloons for Static Windows” on page 3-65.

To create help balloons for items in dialog boxes or alert boxes, create an 'hdlg' resource that corresponds to an item list resource. You associate the information defined in the 'hdlg' resource to the alert or dialog box in one of three ways:

- n by adding an item of type `HelpItem` to the item list resource
- n by supplying a resource of type 'hwin'
- n by calling the `HMScanTemplateItems` function from your application

The 'hdlg' resource specifies the tip, the alternate rectangle, and help messages for items in a dialog box or alert box. The item list resource describes the items, and, if it includes an item of type `HelpItem`, it can contain the resource ID of a corresponding 'hdlg' resource. The Help Manager uses the display rectangles defined in the item list resource as the hot rectangles for the items. The Help Manager uses the alternate rectangles specified in the 'hdlg' resource for transposing help balloons' tips when trying to fit the balloons onscreen.

For those items designated in the 'hdlg' resource, the Help Manager automatically tracks the cursor and displays help balloons when the following conditions are met: the dialog or alert box has an item of type `HelpItem` in its item list resource; your application calls the Dialog Manager routine `ModalDialog`, `IsDialogEvent`, `NoteAlert`, `StopAlert`, `CautionAlert`, or `Alert`; and help is enabled.

If the cursor passes over any active windows, including dialog or alert boxes, the Help Manager searches the current resource file for resources of type 'hwin' (described in “Providing Help Balloons for Static Windows” on page 3-65). The Help Manager attempts to match either the title of the window or the `windowKind` value in its window record with the title or `windowKind` value specified in an 'hwin' resource. The matched 'hwin' resource, in turn, specifies the resource ID of an 'hdlg' or 'hrct' (rectangle help) resource that contains the relevant help message. (The 'hrct' resource is described in “Providing Help Balloons for Static Windows” on page 3-65.) As described in “Providing Help Balloons for Window Content” on page 3-63, the 'hwin' resource can provide help for various other interface features across the entire window as well as for items in a dialog box or an alert box.

If you prefer, you can track and display help balloons for modal dialog boxes and alert boxes yourself by using an event-filter function and calling the `HMScanTemplateItems`

Help Manager

function. Using `HMScanTemplateItems` requires you to modify your code. For further information on `HMScanTemplateItems`, see “Setting and Getting Information for Help Resources” beginning on page 3-114.

As shown here, a Rez input file for an 'hdlg' resource contains a header component, a missing-items component, and dialog-item components.

Component	Element
Header	Help Manager version
	Index number of starting item (first item is number 0)
	Options
	Balloon definition function
	Variation code
Missing items	Tip's coordinates
	Alternate rectangle
	Identifier for help messages
	Help message for missing, unselected active controls (that is, those with highlight values of 0), or for missing enabled items that are not controls
	Help message for missing dimmed controls (that is, those with highlight values of 255), or for missing disabled items that are not controls
	Help message for missing active controls that are “on” (that is, those with highlight values of 1)
First dialog item	Help message for missing active controls with highlight values other than 0, 1, and 255
	Tip's coordinates
	Alternate rectangle
	Identifier for help messages
	Help message for an active, unselected control (that is, one with a highlight value of 0), or for an enabled item that is not a control
	Help message for a dimmed control (that is, one with a highlight values of 255), or for a disabled item that is not a control
Next dialog item	Help message for an active control that is “on” (that is, one with a highlight value of 1)
	Help message for an active control with a highlight value other than 0, 1, and 255
	(Same as for first dialog item)
Last dialog item	.
	.
	.

Help Manager

As described in greater detail later, the way the Help Manager interprets many of the elements depends on whether the item it describes is a control, such as a checkbox or radio button, or something else, such as static text or an icon.

Specifying Header Information for the 'hdlg' Resource

Use the header component to specify the Help Manager version number, the starting index, options, the balloon definition function, and the variation code. As in the other help resources, specify the `HelpMgrVersion` constant for the first element of the header component of the 'hdlg' resource.

You use the second element to associate the help messages beginning at any item number and then continuing sequentially through the item list ('DITL') resource. To derive an item number to start from, the Help Manager adds the index number you specify for this element to the number of the first item in the item list resource. Thus, index number 0 starts with the item number 1 in the item list resource (because 0 plus 1 equals 1). For example, to describe help messages for only the fifth through seventh items, specify 4 as the starting index in the header component and, because 4 plus 1 equals 5, provide help messages that start with the fifth and proceed through the sixth and seventh items.

For the options element, specify a constant (normally, `hmDefaultOptions`) or the sum of several constants' values from this list. (These options are described in "Specifying Options in Help Resources" beginning on page 3-25.)

```
CONST hmDefaultOptions    = 0;  {use defaults}
      hmUseSubID           = 1;  {use subrange resource IDs }
                               { for owned resources}
      hmAbsoluteCoords     = 2;  {ignore coords of window }
                               { origin and treat upper-left }
                               { corner of window as 0,0}
      hmSaveBitsNoWindow  = 4;  {don't create window; save }
                               { bits; no update event}
      hmSaveBitsWindow    = 8;  {save bits behind window }
                               { and generate update event}
```

Specify the balloon definition function and variation code (both typically 0) in the fourth and fifth elements of the header component. (These are described in detail in "Specifying Header Information for the 'hmnu' Resource" beginning on page 3-32.)

Specifying Missing-Item Information

Following the header component, you can specify the help message for items that are missing from the 'hdlg' resource or that are present but have no help messages defined for a particular state. (The function of the missing-items component of the 'hdlg' resource is similar to that of the missing-items component of the 'hmnu' resource. For details, see "Specifying Help for Menu Items Missing From the Resource" beginning on page 3-33.)

Help Manager

In the missing-items component, use the first element to specify a set of tip coordinates and use the second element to specify an alternate rectangle. Both specifications apply to the help messages specified in the other elements of this component.

The tip's coordinates are always relative to the item's position in the dialog box. If you specify the point (0,0) as a default tip, then it is placed 10 pixels from the right and 10 pixels from the bottom of the item's rectangle (as specified in the item list resource) for all missing items. To move the missing item's tip relative to this default location, you can specify positive or negative integers in place of the coordinates (0,0).

If you want an alternate rectangle that is either larger or smaller than a display rectangle, use the missing item's alternate rectangle to specify offsets that apply to the display rectangles for all items in the dialog box. (Remember that the alternate rectangle is used by the Help Manager for transposing the tip if a help balloon does not fit onscreen.) The Help Manager adds the top, left, bottom, and right offsets to the coordinates of an item's display rectangle. For example, if you specify (0,0,0,0) as the missing item's alternate rectangle offsets, the Help Manager uses the display rectangles as alternate rectangles for all missing items. You can specify positive or negative integers for these offsets to move an alternate rectangle's coordinates relative to a display rectangle's coordinates.

Use the third element of the missing-items component to supply one of these identifiers: `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`, described in “Specifying the Format for Help Messages” on page 3-23. In the remaining four elements of this component, supply the help messages for items in the item list resource that do not otherwise have help messages specified in this 'hdlg' resource. You can supply either text strings for the help messages or the resource IDs of resources that contain the help messages.

When displaying help balloons for a control, the Help Manager examines the highlight value in the `ctrlHilite` field of the control record. An active control that is not selected by the user has a highlight value of 0. Specify a help message for all missing highlighted controls in the fourth element of the missing-items component of the 'hdlg' resource.

An inactive—that is, dimmed—control has a highlight value of 255. Specify a help message for all missing dimmed controls in the fifth element of the missing-items component.

Note

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`. When you don't want the Dialog Manager to report events involving an item in a dialog box, you mark it disabled in the item list resource. The Dialog Manager makes no visual distinction between disabled and enabled items. See the chapters “Control Manager” and “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for more information. \cup

Help Manager

When, as with checkboxes and radio buttons, the user turns on an on-and-off control, the control has a highlight value of 1. Specify a help message for all missing, active, “on” controls in the sixth element of the missing-items component.

In addition to the values 0, 1, and 255, multipart controls—such as scroll bars—can also take highlight values between 2 and 253, signifying the part code for the part of the control that has been selected by the user. However, you can specify only one message for all possible highlight values that a control might have other than 0, 1, and 255. You can use the seventh element of the missing-items component to specify this message for missing controls.

The following section offers guidelines about what sorts of messages to provide for different types of controls according to their states. For more detailed information about controls, see the chapter “Control Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

When displaying help for items that are not controls, the Help Manager examines only whether the item is enabled or disabled, as specified in the item list resource. For an enabled item (other than a control), you specify a help message in the fourth element of its component in the 'hdlg' resource. In the fifth element, you specify the help message for the item when it is disabled. The sixth and seventh elements apply only to controls. You should supply these elements with either empty strings or resource IDs of 0, depending on the format indicated by the identifier you specified in the third element of the component.

Specifying Help for Items in an Alert or Dialog Box

After the missing-items component, create **dialog-item components** that specify help messages for the individual items. The first dialog-item component must relate to the item number indexed in the header component; list the remaining dialog-item components in the same order in which they appear in the item list resource. (See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the item list resource.)

Use the first element of a dialog-item component to specify the coordinates of the help balloon’s tip for that item. Use coordinates local to the item’s display rectangle (which is specified in the item list resource) to specify the tip. You can specify (0,0) to place the tip 10 pixels from the right and 10 pixels from the bottom of the item’s display rectangle.

Use the second element of a dialog-item component to specify an alternate rectangle for the item. Note that you cannot specify hot rectangles—only alternate rectangles—in an 'hdlg' resource. This is because the Help Manager uses the display rectangles specified in the item list resource as the hot rectangles for help balloons. (If you must specify hot rectangles that are different from the items’ rectangles, use the 'hrct' resource as described in “Specifying Help for Rectangles in Windows” on page 3-67.) You can, however, specify alternate rectangles in 'hdlg' resources that are different from the display rectangles defined in the item list resource. Alternate rectangles give you additional flexibility in positioning your help balloons onscreen. If you make your alternate rectangle smaller than the display rectangle, for example, you have greater assurance that the Help Manager will be able to fit the help balloon onscreen; if you

Help Manager

specify an alternate rectangle that is larger than the display rectangle, you have greater assurance that the help balloon will not obscure some important portion within the display rectangle.

Specify offsets from the item's display rectangle if you want an alternate rectangle that is different from the display rectangle. The Help Manager adds the top, left, bottom, and right offsets that you specify to the coordinates of the item's display rectangle. For example, if you specify (0,0,0,0) as the alternate rectangle's offsets, the Help Manager uses the item's display rectangle as its alternate rectangle. You can specify positive or negative integers for these offsets to move the alternate rectangle's coordinates relative to the display rectangle's coordinates.

Use the third element of a dialog-item component to supply one of these identifiers: `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`, described in "Specifying the Format for Help Messages" on page 3-23. Note that in any one dialog-item component in the resource, you can specify only one format for all help messages.

The remaining elements in a dialog-item component specify help messages for the related item. As previously described in "Specifying Missing-Item Information" on page 3-54, the Help Manager uses these elements differently according to whether the item is or is not a control. For elements four through seven in a dialog-item component, supply either text strings for the help messages or the resource IDs of resources that contain the help messages.

You do not have to provide a help message for every state of an item. If you do not provide a help message for a particular state, the Help Manager uses the help message specified in the missing-items component. If the missing-items component does not specify a help message for that state, then the Help Manager does not display a help balloon for that state of that item.

In your help balloons for buttons, use the construction "To [perform action], click this button." For example, the help message for the OK button in a Spell Check dialog box should state something similar to "To check the spelling of this document with the options you've chosen, click this button."

For an unselected radio button or checkbox, use the fourth element of the corresponding dialog-item component to describe what happens when the user selects the button or checkbox. For example, an unselected radio button titled "Left" might have a help balloon that states, "To align all text along the left margin of the document, click this button."

For a selected radio button (that is, one that is "on"), use the sixth element of the corresponding dialog-item component to describe what the selected button does, beginning with a verb. At the end of the message, state that the button is selected. For example, a selected radio button titled "Left" might have a help balloon that states, "Aligns all text along the left margin of the document. This option is selected."

For a selected checkbox (that is, one that is "on"), use the sixth element of the corresponding dialog-item component to describe what the selected checkbox does, then describe how to turn off the option. For example, a selected checkbox titled "On Line"

Help Manager

might have a help balloon that states, “Your Macintosh is connected to a remote computer. To disconnect, click this box.”

For a radio button or a checkbox that is dimmed, use the fifth element of the corresponding dialog-item component to describe what it does when it is selected. Use a sentence fragment that begins with a verb. Then explain why the radio button or checkbox is not available. For example, a dimmed radio button titled “Left” might have a help balloon that states, “Aligns all text along the left margin of a document. Not available because no documents are open.”

For an editable text item in a dialog box, use the word “here” in your help message to describe the item. Explain what type of information the user should enter, but don’t describe standard Macintosh editing procedures. For example, an editable text item identified by static text reading “Name” might have a help balloon that states, “Type your name here.” Since an editable text item is typically disabled, you’ll use the fifth element of that item’s component to specify a help balloon.

Since users typically don’t interact with your static text items, you generally shouldn’t provide them with help balloons.

You can use the `HMSkipItem` identifier for an item for which you do not want to provide help. If you specify `HMSkipItem`, the Help Manager does not display help balloons for that item, even if the missing-items component specifies a help message.

In most cases, you should try to describe only the item the balloon is pointing to. It may be tempting to discuss the relationships among items, but this much information can become complex and difficult to read. Remember that the user can point at other items to find out what they are. For example, a button titled “Print” might have a help balloon that states, “To print the document with the options you’ve chosen, click this button.” Do not complicate the message with information like “To print the number of copies of the document that you’ve selected to the left, using the printer named at the top of this dialog box,” and so on.

Listing 3-8 shows a sample dialog-item help resource along with its associated item list and string list resources.

Listing 3-8 Rez input for an item list resource and an 'hdlg' resource

```

resource 'DITL' (145, "Spelling options", purgeable) {
    { {124, 194, 144, 254},
      Button {
          enabled,
          "OK"
      },
      {48, 23, 67, 202},
      CheckBox {
          enabled,
          "Ignore Words in All Caps"
      },
      {83, 23, 101, 196},
      CheckBox {
          enabled,
          "Ignore Slang Terms"
      },
      {13, 23, 33, 254},
      StaticText {
          disabled,
          "WipeOut typing correction options:"
      },
      /*item for Cancel button goes here*/
      {0,0,0,0},          /*for help balloon: scan 'hdlg' with */
                        /* res ID 145*/

      HelpItem {
          disabled,
          HMSCanhdlg     /*scan resource type 'hdlg'*/
          {145}         /*get the resource with ID 145*/
      }
    }
};

resource 'hdlg' (145, "Spell options help", purgeable) {
    /*header component*/
    HelpMgrVersion,    /*version of Help Manager*/
    0,                  /*start help with first item in 'DITL'*/
    hmDefaultOptions,  /*options*/
    0,                  /*balloon definition ID*/
    3,                  /*variation code: hang left of items*/
    /*missing-items component*/
    HMSkipItem {       /*no missing-items help messages*/
    },
    {                  /*help messages for items*/

```

Help Manager

```

/*first dialog-item component: OK button*/
    HMStringResItem { /*store help messages in 'STR#' 145*/
        {10, 10},      /*place tip inside left edge of button*/
        {0,0,0,0},    /*default alternate rectangle: use */
                    /* display rectangle*/
        145, 1,       /*enabled OK button*/
        0, 0,         /*OK button is never dimmed*/
        0, 0,         /*no enabled-and-checked state for */
                    /* button*/
        0, 0          /*no other marked states for button*/
    },
/*second dialog-item component: All Caps checkbox*/
    HMStringResItem { /*store help messages in 'STR#' 145*/
        {6, 6},      /*place tip in checkbox*/
        {0,0,0,0},   /*default alternate rectangle: use */
                    /* display rectangle*/
        145, 2,      /*highlighted state of checkbox*/
        145, 3,      /*dimmed state of checkbox*/
        145, 4,      /*checkbox is checked*/
        0, 0         /*not applicable to this control*/
    },
/*third dialog-item component: Slang Terms checkbox*/
    HMStringResItem { /*store help messages in 'STR#' 145*/
        {6, 6},      /*place tip in checkbox*/
        {0,0,0,0},   /*default alternate rectangle: use */
                    /* display rectangle*/
        145, 5,      /*highlighted state of checkbox*/
        145, 6,      /*dimmed state of checkbox*/
        145, 7,      /*checkbox is checked*/
        0, 0         /*not applicable to this control*/
    }
    /*dialog-item component for Cancel button goes here*/
}
};
resource 'STR#' (145, "Spell options help text") {
    {
        /*[1]*/
        "To check the spelling of this document with the "
        "options you've chosen, click this button.";
        /*[2]*/
        "To prevent the spelling checker from tagging "
        "words--such as acronyms--that consist entirely "
        "of capital letters, click this option.";
    }
}

```

Help Manager

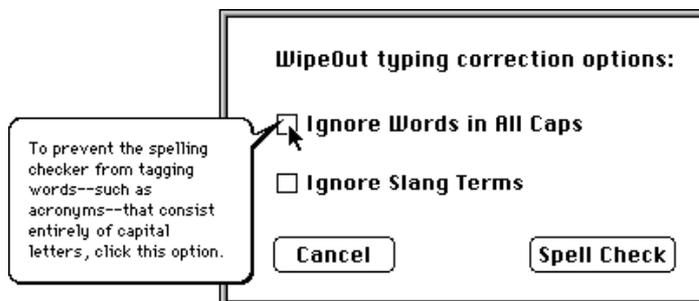
```

/*[3]*/
"Prevents the spelling checker from tagging "
  "words--such as acronyms--that consist entirely "
  "of capital letters. Not available until ";
  "you install the main dictionary.";
/*[4]*/
"The spelling checker is not tagging "
  "words--such as acronyms--that consist entirely "
  "of capital letters. Click here to make the ";
  "spelling checker tag such words.";
/*[5]*/
"To prevent the spelling checker from tagging words "
  "considered to be slang, click this option.";
/*[6]*/
"Prevents the spelling checker from tagging "
  "words considered to be slang. "
  "Not available until you install the slang dictionary.";
/*[7]*/
"The spelling checker is not tagging "
  "words considered to be slang. "
  "Click here to make the spelling checker tag such words.";
/*help strings for Cancel button go here*/
}
};

```

The 'hdlg' resource in Listing 3-8 specifies help messages for the first three items in the item list resource. Figure 3-17 shows the Help Manager displaying a help balloon for the second item.

Figure 3-17 A help balloon in a modal dialog box



Adding a Help Item to an Item List Resource

In Listing 3-8 on page 3-59, an item of type `HelpItem` is included in the item list (`'DITL'`) resource. This item isn't visible to the user; it's provided so that the Help Manager can find the `'hdlg'` or `'hrct'` resource in which you've specified the help messages for your dialog box or alert box.

When creating a help item in an item list resource, specify an empty rectangle—that is, one with coordinates (0,0,0,0)—for the item's display rectangle. Specify `HelpItem` for the item's type, and specify `disabled` for the item's state. Then, specify one of these three identifiers:

Identifier	Purpose
<code>HMScanhdlg</code>	For the items in an item list resource, the Help Manager displays the help messages specified in an <code>'hdlg'</code> resource.
<code>HMScanAppendhdlg</code>	For the items in one item list resource that are appended to those in another item list resource, the Help Manager displays help messages specified in an <code>'hdlg'</code> resource.
<code>HMScanhrct</code>	For rectangular areas in the dialog box or alert box, the Help Manager displays help messages specified in an <code>'hrct'</code> resource.

If you specify help messages for your dialog box or alert box in an `'hdlg'` resource, use either the `HMScanhdlg` or the `HMScanAppendhdlg` identifier. Use the `HMScanAppendhdlg` identifier, however, only when you use the Dialog Manager procedure `AppendDITL` to append the item list resource to another item list resource. The `AppendDITL` procedure is useful, for example, when adding your own items to the standard file dialog box or print dialog box. When you use the `AppendDITL` procedure to add items to an existing dialog box or alert box, the `HMScanAppendhdlg` identifier allows you to provide help balloons for the new items in addition to those balloons already provided for the dialog box or alert box. See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for more information on the `AppendDITL` procedure.

As described in “Specifying Help for Rectangles in Windows” on page 3-67, you can also use the `'hrct'` resource to specify help balloons for areas of your dialog box or alert box. If you specify help messages for a dialog box or alert box in an `'hrct'` resource, you can use the `HMScanhrct` identifier in the help item of the box's item list resource.

Conclude a help item by specifying the resource ID of the `'hdlg'` or `'hrct'` resource that provides the help messages for the dialog box or alert box.

Using a Help Item Versus Using an 'hwin' Resource

Adding an item of type `HelpItem` to an item list resource is the simplest method of associating the help balloons defined in your 'hdlg' (or 'hrct') resource with the item list resource. A slightly more involved method requires you to create an 'hwin' (window help) resource. The advantages and disadvantages of the two methods are listed here.

The advantages of adding an item for help to the item list resource are that

- n it's simple (you have to create only one resource, the 'hdlg' or 'hrct' resource)
- n it works for dialog boxes or alert boxes that have no titles and for those whose `windowKind` values do not adequately differentiate them from other windows (the `windowKind` field of window records is described in the chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*)

The disadvantage of adding an item for help to the item list resource is that it allows you to associate help balloons *only* with items listed in the item list resource.

The advantages of using 'hwin' (window help) resources are that

- n you can provide a single help balloon for a group of related items (rather than having separate help balloons for all the items)
- n you can provide help balloons for areas instead of items inside the dialog box or alert box

The disadvantages of using 'hwin' resources are that

- n it's slightly more complex, because you must create an 'hwin' resource in addition to either an 'hdlg' or an 'hrct' resource
- n it works only for dialog boxes and alert boxes that have titles or `windowKind` values that differentiate them from other windows in your application

Using the 'hwin' resource requires treating the dialog box or alert box as a static window. When the cursor passes over an active window, the Help Manager attempts to match either the title of the window or the `windowKind` value (from its window record) with a title or `windowKind` value you specify in an 'hwin' resource. “Associating Help Resources With Static Windows” beginning on page 3-68 describes how to use 'hwin' resources for dialog boxes, alert boxes, and other kinds of static windows you may wish to define.

Providing Help Balloons for Window Content

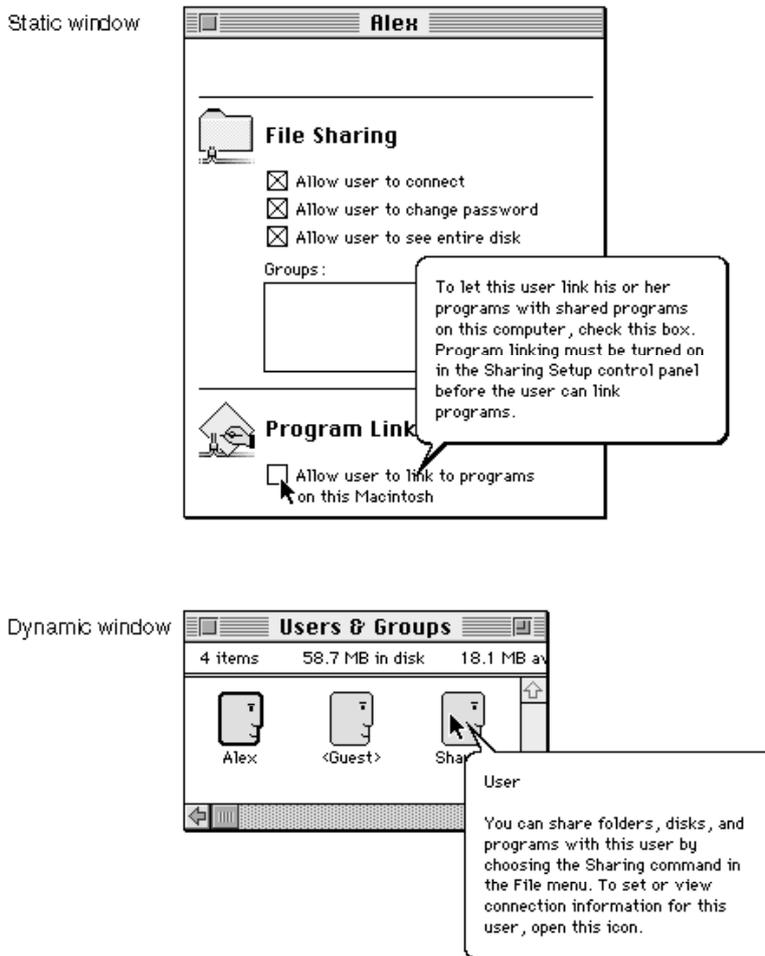
You can create help balloons for objects within the content area of your windows. How you choose to provide help balloons for the content area of your windows depends mainly on whether your windows are static or dynamic.

A **static window** doesn't change its title or reposition any of the objects within its content area. A **dynamic window** can reposition any of its objects within the content area, or its title may change.

Help Manager

For example, any window that scrolls past areas of interest to the user is a dynamic window, because the objects with associated help balloons can change location as the user scrolls. A window that displays only a picture that cannot be resized or scrolled is an example of a static window. Figure 3-18 shows examples of static and dynamic windows. “Providing Help Balloons for Static Windows” beginning on page 3-65, “Associating Help Resources With Static Windows” beginning on page 3-68, and “Providing Help Balloons for Dynamic Windows” beginning on page 3-74 describe how to provide help balloons for these types of windows.

Figure 3-18 Static and dynamic windows



Providing Help Balloons for Static Windows

To provide help balloons for the static windows of your application without modifying its code, create a resource of type 'hwin' (window help) and additional resources of type 'hrct' (rectangle help) or 'hdlg' (dialog-item help). With these resources, the Help Manager automatically tracks the cursor and displays and removes help balloons as the cursor moves into and out of the hot rectangles associated with these resources.

The 'hwin' resource allows you to associate 'hrct' and 'hdlg' resources with your static windows. You use the 'hrct' and 'hdlg' resources to define help balloons for the individual objects within your windows. While the Help Manager uses the display rectangles defined in the item list resource as the hot rectangles for 'hdlg' resources, you can specify your own hot rectangles for alert boxes and dialog boxes and other static windows by using 'hrct' resources.

Note

BalloonWriter gives nonprogrammers an easy, intuitive way to create help balloons for static windows and dialog and alert boxes.

BalloonWriter creates 'hdlg', 'hwin', and 'hrct' resources as appropriate and places them in the resource file of your application; BalloonWriter likewise creates and stores 'STR', 'STR#', and 'TEXT' resources that contain the help messages authored by nonprogramming writers. u

You use an 'hrct' resource to specify tip coordinates, hot rectangles, balloon definition functions, variation codes, and help messages for areas within a static window.

As explained in “Providing Help Balloons for Items in Dialog Boxes and Alert Boxes” on page 3-51, you use the 'hdlg' resource to specify the tip, alternate rectangle, and help messages for items in an alert box or dialog box. “Using a Help Item Versus Using an 'hwin' Resource” on page 3-63 describes how to associate either an 'hdlg' or an 'hrct' resource with an alert box or a dialog box by adding an item of type `HelpItem` to the box's item list resource. This section describes how you can instead treat your dialog boxes or alert boxes as static windows and use an 'hwin' resource instead of `HelpItem` items to associate 'hdlg' and 'hrct' resources with the boxes.

The 'hwin' resource identifies windows by their titles or by their `windowKind` values. You can list all of your windows within one 'hwin' resource, or you can create separate 'hwin' resources for your separate windows. (You'll probably find it easier to maintain your window help if you create only one 'hwin' resource, but, as described later in this section, you must create separate 'hwin' resources for windows that require different options. For example, one window may be matched to its 'hwin' resource by a string *anywhere* in the window's title, and another window may be matched to its 'hwin' resource only by the *exact* string of the window's title.) An 'hwin' resource contains the resource ID (or IDs) of one or more 'hrct' or 'hdlg' resources. With an 'hwin' resource, you can use both 'hrct' and 'hdlg' resources for various parts of the same window.

Help Manager

To use an 'hwin' resource, the window's window record must specify either a title or a windowKind value that adequately distinguishes it from other windows. Within an 'hwin' resource, you could identify the Verb Tenses window shown in Figure 3-20 on page 3-72 by its title, and you could identify the palette window in Figure 3-19 on page 3-70 by its windowKind value.

The chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* describes the windowKind field of the window record. Note that windowKind values of 0, 1, and 3 through 7 are reserved by system software and that dialog boxes or alert boxes must have a value of 2. Because your dialog boxes and alert boxes must have a windowKind value of 2, you can use this value to define only one 'hwin' resource for all untitled dialog boxes and alert boxes. You may find it difficult—using 'hwin', 'hrct', and 'hdlg' resources alone—to provide help balloons for untitled dialog and alert boxes. However, you can use an 'hwin' resource to associate generic help for the common objects of all your untitled dialog boxes and alert boxes, and you can use the HMsSetDialogResID function to provide help for the unique objects among them. The HMsSetDialogResID function is explained on page 3-117.

You describe the tip, a rectangle, and help messages for each object in static windows using either 'hrct' or 'hdlg' resources. As shown here, an 'hrct' resource consists of two types of components: a header component and a hot-rectangle component. You use **hot-rectangle components** to specify the hot rectangles within the window and the help messages for each hot rectangle. (For a description of 'hdlg' resources, see “Providing Help Balloons for Items in Dialog Boxes and Alert Boxes” beginning on page 3-51.)

Component	Element
Header	Help Manager version
	Options
	Balloon definition function
	Variation code
	First hot rectangle
First hot rectangle	Identifier for help message
	Tip's coordinates
	Hot rectangle coordinates
	Help message for hot rectangle
Next hot rectangle	(Same as for first hot rectangle)
	.
	.
	.
Last hot rectangle	(Same as for first hot rectangle)

Specifying Header Information for the 'hrct' Resource

As with the other help resources, specify the `HelpMgrVersion` constant for the first element of the header component of the 'hrct' resource. For the second element, specify a constant (normally, `hmDefaultOptions`) or the sum of several constants' values from the following list. (“Specifying Options in Help Resources” beginning on page 3-25 describes these options.)

```
CONST hmDefaultOptions    = 0;  {use defaults}
      hmUseSubID          = 1;  {use subrange resource IDs }
                               { for owned resources }
      hmAbsoluteCoords    = 2;  {ignore coords of window }
                               { origin and treat upper-left }
                               { corner of window as 0,0 }
      hmSaveBitsNoWindow  = 4;  {don't create window; save }
                               { bits; no update event }
      hmSaveBitsWindow    = 8;  {save bits behind window }
                               { and generate update event }
```

Specify the balloon definition function and the variation code (both typically 0) in the third and fourth elements, respectively, of the header component. (The balloon definition function and the variation code are described in detail in “Specifying Header Information for the 'hmnv' Resource” on page 3-32.)

Specifying Help for Rectangles in Windows

Following the header component, use hot-rectangle components to specify tip coordinates, hot rectangles, and help messages for all the areas in the window that would benefit from having help balloons.

For the first element of each hot-rectangle component, specify the format that the help messages take. As with the other help resources, specify the format using one of these identifiers: `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`, described in “Specifying the Format for Help Messages” on page 3-23.

Use the second element of the hot-rectangle component to specify the coordinates (local to the window) of the balloon tip. Use the third element to specify the coordinates (local to the window) of the hot rectangle. Use the fourth element to specify your help message, as either a text string or a resource ID.

In a hot-rectangle component, you specify the tip, hot rectangle, and help message for every applicable area in the window. As explained in “Using a Help Item Versus Using an 'hwin' Resource” on page 3-63, you can associate an 'hrct' resource with an alert box or a dialog box by adding an item of type `HelpItem` to the box's item list resource. For windows, you create an 'hwin' resource that contains the resource ID of this 'hrct' resource and that associates the 'hrct' resource with the window. In either

Help Manager

case, the Help Manager automatically tracks the cursor and displays and removes help balloons as the user moves the cursor into and out of the hot rectangles defined in this 'hrct' resource.

If you need to supply a help balloon for an area within a larger area that needs a different help balloon, create 'hrct' resources for both the inner and outer areas and specify their areas as hot rectangles. In your resource file, list the 'hrct' resource for the inner area before the 'hrct' resource for the outer area. Then, when the cursor is in the inner hot rectangle, the Help Manager scans its 'hrct' resource first and displays its help balloon instead of the help balloon for the outer hot rectangle. When the cursor moves from the inner hot rectangle to the outer, the Help Manager removes the inner area's help balloon and displays the balloon for the outer hot rectangle.

As previously explained, you can create an 'hdlg' resource to specify the tips, alternate rectangles, balloon definitions, variation codes, and help messages for items in an item list resource, and you can use an 'hwin' resource to associate that 'hdlg' resource with a dialog box or alert box. When help is enabled and your application calls the Dialog Manager routine `ModalDialog`, `IsDialogEvent`, `Alert`, `NoteAlert`, `CautionAlert`, or `StopAlert`, the Help Manager automatically tracks the cursor and displays and removes help balloons for items specified in the 'hdlg' resource.

Associating Help Resources With Static Windows

To associate 'hrct' and 'hdlg' resources with static windows, create an 'hwin' resource. As shown here, an 'hwin' resource consists of two types of components: a header component and a window component. Use a **window component** to associate an 'hrct' or 'hdlg' resource with a particular window.

Component	Element
Header	Help Manager version Options
First window	Resource ID of an 'hrct' or 'hdlg' resource Resource type ('hdlg' or 'hrct') Length used to compare title strings—or, if flagged by a minus sign (-), the <code>windowKind</code> value of an untitled window Window title string—or empty string if window is untitled
Next window	(Same as the first window component) . . .
Last window	(Same as the first window component)

Specifying Header Information for the 'hwin' Resource

Specify the `HelpMgrVersion` constant for the first element of the header component. For the second element, specify a constant (normally, `hmDefaultOptions`) or the sum of several constants' values from this list.

```
CONST hmDefaultOptions = 0      {use defaults}
      hmUseSubID       = 1;    {use subrange resource IDs }
                                { for owned resources}
      hmMatchInTitle   = 16;   {match window by string }
                                { anywhere in title string}
```

Notice that options regarding local coordinates and bits behind the balloon are not applicable to the 'hwin' resource, but, compared to the other resources related to the Help Manager, the 'hwin' resource has a unique option: `hmMatchInTitle`.

If you're providing help balloons for a desk accessory or a driver that owns other resources, use the `hmUseSubID` constant in the second element. (See the chapter "Resource Manager" in this book for a discussion of owned resources and their resource IDs.)

You can specify the `hmMatchInTitle` constant to match windows containing a specified number of sequential characters starting with any character position in the window title. If you do not specify the `hmMatchInTitle` constant for the second element of the header component, the Help Manager matches characters starting with the first character of the window title.

For example, if an 'hwin' resource specifies the `hmMatchInTitle` constant in the header component, specifies in the window component that four characters should be matched, and specifies the character string `Test` as the window's title string, the Help Manager uses this 'hwin' resource when the cursor is located in any active window that is titled `Test`, `Window Test`, or `Test Case` or is given a title with any other string that contains the characters `Test`.

If you supply the `hmDefaultOptions` constant, the Help Manager treats the resource IDs in this resource as regular resource IDs and not as subrange IDs, and it begins matching characters at the first character of the window strings specified in each window component. As long as the window components all use the same options, you can list help for all your windows in a single 'hwin' resource. You must create separate 'hwin' resources for window components that require different options.

Specifying 'hdlg' or 'hrct' Resources in the 'hwin' Resource

You can specify multiple window components after the header component.

Within the 'hwin' resource you identify 'hrct' resources and 'hdlg' resources by their resource IDs and by their types. Use the first element of a window component to specify the resource ID of either an 'hrct' or an 'hdlg' resource. Use the second element to specify that resource's type—either 'hrct' or 'hdlg'. Use the next two elements to specify the window with which you want to associate the 'hrct' or 'hdlg' resource identified in the first two elements.

Help Manager

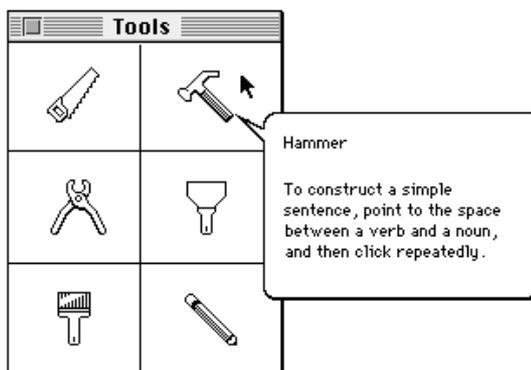
You specify windows in one of these two ways:

- n by specifying the number of characters used for matching a window title in the third element of the window component, and by specifying a string consisting of this number of sequential characters from the window's title in the fourth element
- n by flagging the third element of the component with a minus sign (-), specifying the `windowKind` value from the window's window record in the third element, and leaving an empty string in the fourth element

When an active window has a title or `windowKind` value that matches an `'hwin'` resource, the Help Manager provides help balloons for the hot rectangles associated with the specified `'hrct'` and `'hdlg'` resources.

Figure 3-19 shows a sample palette an application might use and the help balloon displayed for the hammer tool.

Figure 3-19 A tool palette with a help balloon



Note that the help message in Figure 3-19 names the tool. It's a good idea to name tools, because the name of a tool often helps the user determine the purpose of the tool. After naming the tool, describe one or two likely ways to use it. Don't describe every shortcut or trick you can do with the modifier keys.

For dialog boxes and alert boxes, you can use `'hrct'` resources to define hot rectangles in addition to or instead of those associated with the items. For example, you might want to use an `'hwin'` and an `'hrct'` resource in a dialog box to associate a single help balloon with a group of related items rather than provide separate help balloons for all the individual items. (To provide help balloons for individual items by using `'hdlg'` resources alone, see “Providing Help Balloons for Items in Dialog Boxes and Alert Boxes” beginning on page 3-51.)

When providing one help balloon for a group of options in a dialog box, describe first how to implement the options, and then describe how to tell whether an option is selected. If, for example, radio buttons titled Left, Right, and Middle appear in a dialog box grouped under the heading Alignment, a single help balloon explaining this group might state, “To line up the selected text along the left margin, right margin, or middle of

Help Manager

the page, click one of these buttons. A dot indicates the selected option.” A help balloon for several checkboxes grouped under the heading Style might state, “To apply design elements to the selected text, click the styles you want. To remove design elements, click the styles you want to remove. An X means a style has been applied.”

Listing 3-9 shows the 'hwin' resource and the 'hrct' resource for the palette in Figure 3-19.

Listing 3-9 Rez input for corresponding 'hwin' and 'hrct' resources

```
resource 'hwin' (128, "Window help resource", purgeable) {
    HelpMgrVersion, hmDefaultOptions, /*header component*/
    {
        /*window component*/
        128,          /*resource ID of type specified on next line*/
        'hrct',      /*resource type for defining help*/
        5,           /*length to use when comparing strings*/
        "Tools"     /*window's title string*/
    }
};

resource 'hrct' (128, "Tools palette help") {
    /*header component*/
    HelpMgrVersion,
    hmDefaultOptions,
    0,                /*balloon definition function*/
    0,                /*variation code*/
    {
        /*hot-rectangle component for saw tool goes here*/
        /*hot-rectangle component for hammer tool*/
        HMStringResItem {
            {50, 127},          /*tip's coordinates*/
            {22,99,54,131},    /*hot rectangle*/
            147,2              /*'STR#' resource ID and index*/
        }
        /*hot-rectangle components for other tools go here*/
    }
};

resource 'STR#' (147, "Tools palette help text") {
    {
        /*[1] saw tool*/
        /*help text for saw tool goes here*/
        /*[2] hammer tool*/
        "Hammer\n\nTo construct a simple sentence, point to the "
        "space between a verb and a noun, and then click "
```

Help Manager

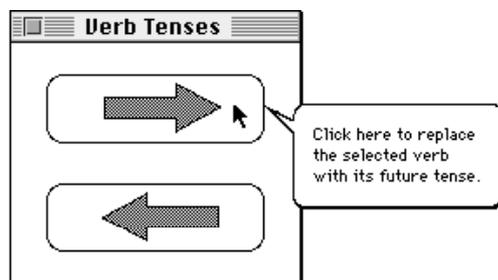
```

    "repeatedly.";
    /*help for other tools goes here*/
  }
};

```

You can also use the 'hwin' resource to associate help for items in an alert box or a dialog box. Figure 3-20 shows the Help Manager displaying a help balloon for an item in the dialog box titled Verb Tenses.

Figure 3-20 A help balloon for a dialog box with a title



Listing 3-10 shows how the 'hwin' resource associates an 'hdlg' resource with the dialog box illustrated in Figure 3-20. This 'hwin' resource associates help with three different windows: the first is the window titled Tools, the second is an untitled window with a windowKind value of 10, and the third is the dialog box titled Verb Tenses.

Listing 3-10 Rez input for specifying help for titled and untitled windows

```

resource 'hwin' (128, "Window help resource", purgeable) {
  /*header component*/
  HelpMgrVersion, hmDefaultOptions,
  { /*first window component*/
    128,          /*help resource ID for Tools window*/
    'hrct',      /*resource type for defining help*/
    5,           /*length to use when comparing strings*/
    "Tools",     /*window's title string*/
    /*second window component*/
    129,          /*help resource ID for untitled window*/
    'hdlg',      /*resource type for defining help*/
    -10,         /*match on windowKind values of 10*/
    "",          /*matching on windowKind, so empty */
                /* string goes here*/
    /*third window component*/
    130,          /*help res ID for Verb Tenses window*/

```

Help Manager

```

        'hdlg',          /*resource type for defining help*/
        11,             /*length to use when comparing strings*/
        "Verb Tenses", /*dialog box's title string*/
    }
};
resource 'hdlg' (130, "Help for Verb Tense control", purgeable) {
    /*header component*/
    HelpMgrVersion,    /*version of Help Manager*/
    0,                 /*start with first item in item list*/
    hmDefaultOptions, /*options*/
    0,                 /*balloon definition ID*/
    0,                 /*variation code*/
    /*missing-items component*/
    HMSkipItem { /*no missing-item help message*/
    },
    { /*first dialog-item component*/
        HMStringResItem {
            {20, 130}, /*tip--local to item's display rectangle*/
            {0,0,0,0}, /*default alternate rectangle: use */
                        /* item's display rectangle*/
            131, 1,    /*highlighted control for future tense*/
            131, 2,    /*dimmed control for future tense*/
            0, 0,      /*no checked state for control*/
            0, 0      /*no other states for control*/
        },
        /*second dialog-item component*/
        HMStringResItem {
            {20, 130}, /*tip--local to item's display rectangle*/
            {0,0,0,0}, /*default alternate rectangle: use */
                        /* item's display rectangle*/
            131, 3,    /*highlighted control for past tense*/
            131, 4,    /*dimmed control for past tense*/
            0, 0,      /*no enabled-and-checked control*/
            0, 0      /*no other marks for control*/
        }
    }
};
resource 'STR#' (131, "Verb tense help strings") {
    {
        /*[1] highlighted control for future tense: help text*/
        "Click here to replace the selected verb with its "
        "future tense.";
        /*[2] dimmed control for future tense: help text*/
    }
};

```

Help Manager

```

    "Click here to replace a verb with its future tense. "
    "Not available now because you have not selected a verb.";
/*[3] /*highlighted control for past tense: help text*/
"Click here to replace the selected verb with its past tense.";
/*[4] dimmed control for past tense: help text*/
"Click here to replace a verb with its past tense. "
    "Not available now because you have not selected a verb.";
}
};

```

Providing Help Balloons for Dynamic Windows

To create help balloons for objects whose location in the content area of windows may vary, your application needs to use Help Manager routines to display and remove balloons as the user moves the cursor.

Note

Nonprogrammers can use the BalloonWriter tool to provide you with delimited ASCII text that you can then use in conjunction with Help Manager routines to display balloons for dynamic windows. However, BalloonWriter does not create the resources or routines necessary to automatically display help balloons for these types of windows. u

You should display or remove help balloons for dynamic windows at the same time that you normally check the mouse location to display or change the cursor. For example, if you provide your own `DoIdle` procedure (as described in the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*), you can also check the mouse location and, if the cursor is located in a hot rectangle, you should display the associated help balloon.

To create help balloons for the content area of a dynamic window, you need to

- n identify the hot rectangles for each area or object
- n create data structures to store the locations of the hot rectangles
- n determine how to calculate their changing locations
- n track and update the hot rectangles
- n use the `HMShowBalloon` function to display a help balloon when the cursor is located in a hot rectangle

After defining all the hot rectangles within your content region, create separate 'STR', 'STR#', 'PICT', or 'TEXT' and 'styl' resources for the help balloons' messages.

You don't have to store the help messages in these resources when using `HMShowBalloon`, but doing so makes your application easier to localize.

Help Manager

When you use the `HMShowBalloon` function, your application is responsible for tracking the cursor and determining when to display the help balloon. If you use the `HMShowBalloon` function, you can let the Help Manager track the cursor and determine when to remove the help balloon, or your application can remove the balloon when necessary by calling the `HMRemoveBalloon` function. If you display your own help balloons using the `HMShowBalloon` function, you should use the `HMGetBalloons` function to determine whether help is enabled before displaying a help balloon. If help is not enabled, you don't need to call any Help Manager routines that display balloons, because they won't do anything unless `HMGetBalloons` returns `TRUE`.

The `HMShowBalloon` function is useful for

- n windows whose content changes
- n windows that can be resized
- n windows that contain hot rectangles with variable locations
- n situations in which you want your application to have more control over the display and removal of the help balloon

For example, windows with scrolling icons (such as the Users & Groups dynamic window shown in Figure 3-18 on page 3-64) require you to use `HMShowBalloon` to display help balloons for the icons. Likewise, if you have tools—such as rulers that users configure for tab stops in a word-processing document—that scroll with a document, you'll need to use `HMShowBalloon` to display help balloons for the scrolling tools.

When using `HMShowBalloon`, you specify the help message, the balloon tip's coordinates, an alternate rectangle to use if the Help Manager needs to move the tip, an optional pointer to a function that can modify the tip and alternate rectangle coordinates, the balloon definition function, and the variation code. In the final parameter to the `HMShowBalloon` function, provide a constant that tells the Help Manager whether to save the bits behind the balloon.

```
myErr := HMShowBalloon(aHelpMsg, tip, alternateRect, tipProc,
                      theProc, variant, method);
```

Specify the help message in a help message record, which you pass in the `aHelpMsg` parameter to the `HMShowBalloon` function. You can specify the help message for each hot rectangle using text strings, 'STR' resources, 'STR#' resources, styled text resources, 'PICT' resources, handles to styled text records, or handles to pictures.

Help Manager

The `HMMessageRecord` data type defines the help message record.

```

TYPE HMMessageRecord =
RECORD
    hmmHelpType:      Integer;          {type of next field}
    CASE Integer OF
        khmmString:   (hmmString: Str255); {Pascal string}
        khmmPict:     (hmmPict: Integer);  {'PICT' resource ID}
        khmmStringRes: (hmmStringRes: HMStringResType);
                                {'STR#' resource
                                { ID and index}
        khmmTEHandle: (hmmTEHandle: TEHandle);
                                {TextEdit handle}
        khmmPictHandle: (hmmPictHandle: PicHandle);
                                {picture handle}
        khmmTERes:    (hmmTERes: Integer); {'TEXT'/'styl' }
                                { resource ID}
        khmmSTRRes:   (hmmSTRRes: Integer); {'STR ' resource ID}
    END;

```

The `hmmHelpType` field specifies the data type of the second field of the help message record. You specify one of these constants for the `hmmHelpType` field.

```

CONST khmmString      = 1;      {Pascal string}
      khmmPict        = 2;      {'PICT' resource ID}
      khmmStringRes   = 3;      {'STR#' resource ID and index}
      khmmTEHandle    = 4;      {TextEdit handle}
      khmmPictHandle  = 5;      {picture handle}
      khmmTERes       = 6;      {'TEXT' and 'styl' resource ID}
      khmmSTRRes      = 7;      {'STR ' resource ID}

```

You specify the help message itself in the second field of the help message record.

You can specify the help message by using a text string, a text string stored in a resource of type 'STR ', or a text string stored as an 'STR#' resource. You can also provide the information using styled text resources, or you can provide a handle to a styled text record. If you want to provide a picture for the help message, you can use a resource of type 'PICT' or provide a handle to a picture.

Listing 3-11 illustrates how to specify a Pascal string using the `khmmString` constant in the help message record. (Although you can specify a string from within your code, storing the strings in resources and then accessing them through the Resource Manager makes localization easier.)

Listing 3-11 Using a string resource as the help message for `HMShowBalloon`

```

PROCEDURE DoTextStringBalloon;
VAR
    aHelpMsg:      HMMessageRecord;
    tip:           Point;
    alternateRect: RectPtr;
    err:           OSErr;
BEGIN
    aHelpMsg.hmmHelpType := khmmString;
    aHelpMsg.hmmString := 'To turn the page, click here.';
    MySetTipAndAltRect(tip, alternateRect); {initialize values}
    err := HMShowBalloon(aHelpMsg, tip, alternateRect,
                        NIL, 0, 0, kHMRegularWindow);
END;

```

To use a picture, you can either store the picture as a 'PICT' resource or create the 'PICT' graphic from within your application and provide a handle to it. Because the Help Manager uses the resource itself or the actual handle that you pass to `HMShowBalloon`, your 'PICT' resource should be purgeable, or, when using a handle to a 'PICT' resource, you should release the handle or dispose of it when you are finished with it.

Listing 3-12 illustrates how to use the `khmmPict` constant for specifying a 'PICT' resource ID in a help message record. The help message record is then passed in the `aHelpMsg` parameter of the `HMShowBalloon` function.

Listing 3-12 Using a picture resource as the help message for `HMShowBalloon`

```

PROCEDURE DoPictBalloon;
VAR
    aHelpMsg:      HMMessageRecord;
    tip:           Point;
    alternateRect: RectPtr;
    err:           OSErr;
BEGIN
    aHelpMsg.hmmHelpType := khmmPict;
    aHelpMsg.hmmPict := 128;          {resource ID of 'PICT' resource}
    MySetTipAndAltRect(tip, alternateRect); {initialize values}
    err := HMShowBalloon(aHelpMsg, tip, alternateRect,
                        NIL, 0, 0, kHMRegularWindow);
END;

```

Help Manager

Listing 3-13 illustrates how to specify a handle to a 'PICT' resource using the `khmmPictHandle` constant in the help message record. The help message record is then passed to the `HMShowBalloon` function in the `aHelpMsg` parameter.

Listing 3-13 Using a handle to a picture resource as the help message for `HMShowBalloon`

```
PROCEDURE DoPictBalloon2;
VAR
    pict:          PicHandle;
    aHelpMsg:      HMMessageRecord;
    tip:           Point;
    pictFrame:     Rect;
    alternateRect: RectPtr;
    err:           OSErr;
BEGIN
    MySetPictFrame(pictFrame); {initialize pictFrame}
    pict := OpenPicture(pictFrame);
    DrawString('Test Balloon');
    ClosePicture;
    aHelpMsg.hmmHelpType := khmmPictHandle;
    aHelpMsg.hmmPictHandle := pict;
    MySetTipAndAltRect(tip, alternateRect); {initialize values}
    err := HMShowBalloon(aHelpMsg, tip, alternateRect,
                        NIL, 0, 0, kHMRegularWindow);
    KillPicture(pict);
END;
```

To specify a help message stored in a string list ('STR#' resource) in a help message record, you must first create a Help Manager string list record. The `HMStringResType` data type defines a Help Manager string list record.

```
TYPE HMStringResType =
RECORD
    hmmResID:   Integer;   {resource ID of 'STR#' resource}
    hmmIndex:   Integer;   {index of string}
END;
```

Help Manager

The `hmmResID` field specifies the resource ID of the 'STR#' resource, and the `hmmIndex` field specifies the index of a string within that resource.

To use a string stored in an 'STR#' resource with the `HMShowBalloon` function, use the `khmmStringRes` constant in the `hmmHelpType` field of the help message record, and supply the `hmmStringRes` field with a Help Manager string list record, as shown in Listing 3-14.

Listing 3-14 Using a string list resource as the help message for `HMShowBalloon`

```
PROCEDURE DoStringListBalloon;
VAR
    aHelpMsg:           HMMessageRecord;
    tip:                Point;
    alternateRect:      RectPtr;
    khmmStringRes:      HMStringResType;
    err:                OSErr;
BEGIN
    aHelpMsg.hmmHelpType := khmmStringRes;
    aHelpMsg.hmmStringRes.hmmResID := 1000;
    aHelpMsg.hmmStringRes.hmmIndex := 1;
    MySetTipAndAltRect(tip, alternateRect); {initialize values}
    err := HMShowBalloon(aHelpMsg, tip, alternateRect,
                        NIL, 0, 0, kHMRegularWindow);
END;
```

To use styled text resources with the `HMShowBalloon` function, use the `khmmTERes` constant in the `hmmHelpType` field of the help message record. In the next field, supply a resource ID that is common to both a 'TEXT' resource and a style scrap ('styl') resource. For example, you might create a 'TEXT' resource that contains the words "Displays your text in boldface print." You would also create a 'styl' resource (with the same resource ID as the 'TEXT' resource) that applies boldface style to the word "boldface." When you specify the `HMTEResItem` constant and the resource ID number of the 'TEXT' and 'styl' resources, the Help Manager employs `TextEdit` routines to display your text with your prescribed styles.

Help Manager

To use a handle to a styled text record, supply the `khmmTEHandle` constant in the `hmmHelpType` field, as illustrated in Listing 3-15.

Listing 3-15 Using styled text resources as the help message for `HMShowBalloon`

```
PROCEDURE DoStyledTextBalloon;
VAR
    aHelpMsg:      HMMMessageRecord;
    tip:           Point;
    alternateRect: RectPtr;
    hTE:           TEHandle;
    err:           OSErr;
BEGIN
    hTE := TStyleNew(destRect, viewRect);    {or, use TENew}
    {be sure to fill in data in handle here}
    aHelpMsg.hmmHelpType := khmmTEHandle;
    aHelpMsg.hmmTEHandle := hTE;
    MySetTipAndAltRect(tip, alternateRect); {initialize values}
    err := HMShowBalloon(aHelpMsg, tip, alternateRect,
                        NIL, 0, 0, kHMRegularWindow);
END;
```

When using the `HMShowBalloon` function, you specify the tip in the `tip` parameter and the rectangle pointed to in the `alternateRect` parameter in global coordinates. The Help Manager calculates the location and size of the help balloon. If the help balloon fits onscreen, the Help Manager displays the help balloon using the specified tip.

If you use the previously described help resources to define help balloons, the Help Manager uses the hot rectangles you specify in the help resources for two purposes: first, to associate areas of the screen with help balloons and, second, to move the tip if the help balloon doesn't fit onscreen.

However, if you use the `HMShowBalloon` function to display help balloons, you must identify hot rectangles, create your own data structures to store their locations, track the cursor yourself, and call `HMShowBalloon` when the cursor moves to your hot rectangles. The Help Manager does not know the locations of your hot rectangles, so it cannot use them for moving the tip if the help balloon is placed offscreen. Instead, the Help Manager uses the alternate rectangle that you point to with the `alternateRect` parameter. Often, you specify the same coordinates for the alternate rectangle that you specify for your hot rectangle. However, you may choose to make your alternate rectangle smaller or larger than your hot rectangle. If you make your alternate rectangle smaller than your hot rectangle, you have greater assurance that the Help Manager will be able to fit the help balloon onscreen; if you specify an alternate rectangle that is larger than your hot rectangle, you have greater assurance that the help balloon will not obscure some object explained by the balloon.

Help Manager

By specifying a rectangle in the `alternateRect` parameter, you tell the Help Manager to call `HMRemoveBalloon` to automatically remove the balloon when the cursor leaves the area bounded by the rectangle. However, if you specify `NIL` for the `alternateRect` parameter, your application is responsible for tracking the cursor and determining when to remove the help balloon. When you specify `NIL`, the Help Manager also does not attempt to calculate a new tip position if the help balloon is offscreen.

Use the `tipProc` parameter (the fourth parameter to `HMShowBalloon`) to specify the tip function called by the Help Manager before displaying the balloon. Specify `NIL` to use the Help Manager's default tip function, or supply your own tip function and point to it in this parameter. Writing your own tip function is described in "Application-Defined Routines" beginning on page 3-128.

Note

When you call the `HMShowBalloon` function, the Help Manager does not display the help balloon or attempt to move the tip under either of these conditions:

The help balloon's tip is offscreen or in the menu bar, and you don't specify an alternate rectangle.

Both the help balloon's tip and the alternate rectangle are offscreen. \cup

Use the parameter `theProc` (the fifth parameter) to specify a balloon definition function. To use the standard balloon definition function, specify 0 in this parameter. To use your own balloon definition function, specify the resource ID of the 'WDEF' resource containing your balloon definition function. Writing your own balloon definition function is described in "Writing Your Own Balloon Definition Function" on page 3-93.

Supply a variation code for the balloon definition function in the `variant` parameter (the sixth parameter to `HMShowBalloon`). Specify 0 in the `variant` parameter to use the default help balloon shape, specify a code from 1 to 7 to use one of the other positions provided by the standard balloon definition function, or specify a code to use one of the positions provided by your own balloon definition function.

Use the `method` parameter (the last parameter to `HMShowBalloon`) to specify how the Help Manager should draw and remove the balloon. Use the following constants for the parameter.

```
CONST kHMRegularWindow      = 0;  {don't save bits; just update}
      kHMSaveBitsNoWindow   = 1;  {save bits; don't do update}
      kHMSaveBitsWindow     = 2;  {save bits; do update event}
```

If you specify `kHMRegularWindow`, the Help Manager draws and removes the help balloon as if it were a window. That is, when drawing the balloon, the Help Manager does not save bits behind the balloon, and when removing the balloon, the Help Manager generates an update event. This is the standard behavior of help balloons; it is the behavior you should normally use.

Help Manager

If you specify `kHMSaveBitsNoWindow` in the method parameter, the Help Manager does not create a window for displaying the balloon. Instead, the Help Manager creates a help balloon that is more like a menu than a window. The Help Manager saves the bits behind the balloon when it creates the balloon. When it removes the balloon, the Help Manager restores the bits without generating an update event. You should use this technique only in a modal environment where the bits behind the balloon cannot change from the time the balloon is drawn to the time it is removed. For example, you might specify the `kHMSaveBitsNoWindow` constant when providing help balloons for pop-up menus that overlay complex graphics, which might take a long time to redraw with an update event.

If you specify `kHMSaveBitsWindow`, the Help Manager treats the help balloon as a hybrid having properties of both a menu and a window. That is, the Help Manager saves the bits behind the balloon when it creates the balloon, and when it removes the balloon, it both restores the bits and generates an update event. You'll rarely need this option. It is necessary only in a modal environment that might immediately change to a nonmodal environment—that is, where the bits behind the help balloon are static when the balloon is drawn, but can possibly change before the help balloon is removed.

Listing 3-16 shows a sample routine that displays help balloons for hot rectangles within the content area of a window.

Listing 3-16 Using `HMShowBalloon` to display help balloons

```
PROCEDURE FindAndShowBalloon (window: WindowPtr);
VAR
    i:           Integer;
    mouse:      Point;
    savePort:   GrafPtr;
    helpMsg:    HMMMessageRecord;
    inRect:     Boolean;
    hotRect:    Rect;
    result:     OSErr;
BEGIN
    IF (window = FrontWindow) THEN          {only do frontmost windows}
    BEGIN
        GetPort(savePort);   {save the old port for later}
        SetPort(window);     {set the port to the front window}
        GetMouse(mouse);     {get the mouse location in local }
                               { coords}
        inRect := FALSE;     {clear flag saying mouse location }
                               { wasn't in any hot rectangle}
        IF PtInRect(mouse, window^.portRect) THEN
            {if the cursor is in the window}
```

Help Manager

```

FOR i := 1 TO 10 DO {check all ten predefined hot }
                    { rectangles in the window}
IF PtInRect(mouse, MyPredefinedRects[i]) THEN
BEGIN {the cursor is in a hot rectangle}
  IF (i <> gLastBalloon) THEN
    {user moved cursor to a different hot rectangle}
  BEGIN
    hotRect := MyPredefinedRects[i];
    LocalToGlobal(hotRect.topLeft);
    {converting rect to global}
    LocalToGlobal(hotRect.botRight);
    WITH hotRect DO {put the tip in the middle}
      SetPt(mouse, (right + left) div 2,
            (bottom + top) div 2);
    helpMsg.hmmHelpType := khmmStringRes;
    {get help message from an 'STR#' resource}
    helpMsg.hmmStringRes.hmmResID := kHelpMsgsID;
    helpMsg.hmmStringRes.hmmIndex := i;
    result := HMShowBalloon
                (helpMsg, {use just-made help msg}
                mouse,    {pointing to this tip}
                @MyPredefinedRects[i], {use hot }
                { rect for alt rect}
                NIL,      {no special tip proc}
                0,0,      {using default balloon}
                kHMRegularWindow);{don't save bits behind}
    IF (result = noErr) THEN {then remember balloon}
      gLastBalloon := i;
  END;
  inRect := TRUE; {remember when the }
                { cursor is in any hot rect}
END;
IF not inRect THEN
  gLastBalloon := -1; {clear last balloon global for }
                    { no hit}
  SetPort(savePort); {restore the port}
END;
END; {FindAndShowBalloon}

```

The `FindAndShowBalloon` procedure in Listing 3-16 tracks the cursor, and, if the cursor is located in a predefined hot rectangle, it displays a help balloon for that rectangle. In this example there are ten predefined rectangles (in the `MyPredefinedRects` array) and ten corresponding help messages in an 'STR#'

Help Manager

resource (of ID `kHelpMsgsID`)—one message for each hot rectangle. Other supporting routines can update the coordinates of the hot rectangles as their locations change.

You can also use the `HMShowBalloon` function from the event filter function of a modal dialog box or an alert box. See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on event filter functions.

Overriding Help Balloons for Non-Document Icons

The Finder displays default help balloons for all icon types. By specifying an `'hfdr'` resource in your application's resource fork, you can provide your own help balloon for the Finder to display when the user moves the cursor over your non-document icons.

Note

BalloonWriter, available from APDA, is a tool that gives nonprogrammers an easy way to create help balloons for most of the icons that the Finder displays for your software. BalloonWriter creates an `'hfdr'` resource and places it in the resource fork of the file represented by the icon; BalloonWriter likewise creates and stores an `'STR'` resource that contains the help message. [u](#)

To override the Finder's default help balloons for your application icon, desk accessory icon, system extension icon, or control panel icon, create an `'hfdr'` resource in your resource file. As shown here, an `'hfdr'` resource consists of two components: a header component and an icon component. Use the **icon component** to specify a help message for your application's Finder icon.

Component	Element
Header	Help Manager version
	Options
	Balloon definition function
	Variation code
Icon	Identifier for help message
	Help message for application icon

Note

You cannot override the default help balloon that the Finder uses for document icons. [u](#)

Use resource ID `-5696` for your `'hfdr'` resource. If an `'hfdr'` resource with that ID exists for an application, the Help Manager uses it instead of the default help balloon supplied by the Finder.

Specifying Header Information for the 'hldr' Resource

As with the other help resources, specify the `HelpMgrVersion` constant for the first element of the header component of the 'hldr' resource. For the second element, specify a constant (normally, `hmDefaultOptions`) or the sum of several constants' values from the following list. ("Specifying Options in Help Resources" beginning on page 3-25 describes these options.)

```

CONST hmDefaultOptions    = 0;  {use defaults}
      hmUseSubID          = 1;  {use subrange resource IDs }
                              { for owned resources }
      hmAbsoluteCoords    = 2;  {ignore coords of window }
                              { origin and treat upper-left }
                              { corner of window as 0,0 }
      hmSaveBitsNoWindow  = 4;  {don't create window; save }
                              { bits; no update event }
      hmSaveBitsWindow    = 8;  {save bits behind window }
                              { and generate update event }

```

Specify the balloon definition function and variation code (both typically 0) in the third and fourth elements, respectively, of the header component. (These are described in detail earlier in "Specifying Header Information for the 'hmnu' Resource" on page 3-32.)

Specifying Help for an Icon

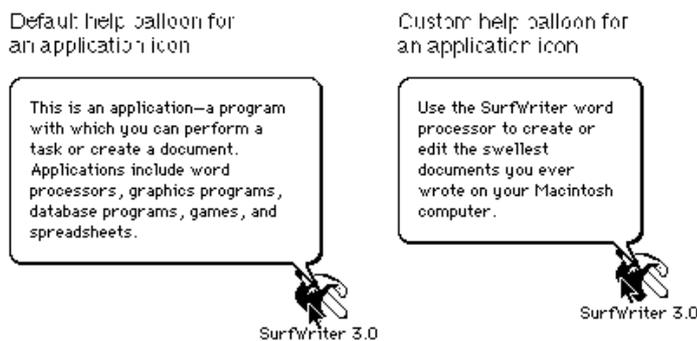
In the icon component, use the first element to specify the format that the help message takes. As with the other help resources, specify the format using one of these identifiers: `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`. These identifiers are described in "Specifying the Format for Help Messages" on page 3-23. (If you specify `HMSkipItem`, no help balloon appears.)

Help Manager

In the second element of the icon component, specify the help message. Your help message doesn't have to describe how to open icons; you can assume that users know how.

Figure 3-21 shows the default help balloon for application icons on the left. A custom help balloon for the same icon appears on the right.

Figure 3-21 Default and custom help balloons for an application icon



The custom help balloon on the right side of Figure 3-21 is supplied with the resources shown in Listing 3-17.

Listing 3-17 Rez input for creating an 'hfdR' resource for an application icon

```
resource 'hfdR' (-5696) {           /*help for SurfWriter icon*/
    /*header component*/
    HelpMgrVersion, hmDefaultOptions, 0, 0,
    { /*icon component*/
        HMSTRResItem {             /*use 'STR ' resource 1001*/
            1001
        }
    }
};
resource 'STR ' (1001) {           /*help message for SurfWriter icon*/
    "Use the SurfWriter word processor to wrote or edit the "
    "swellest documents you ever wrote on "
    "your Macintosh computer."
};
```

Overriding Other Default Help Balloons

The Help Manager also provides default help balloons for the title bar and the close and zoom boxes of an active window, for the windows of inactive applications, for inactive windows of an active application, and for the area outside a modal dialog box.

Apple Computer, Inc., has researched and tested these help messages to ensure that they are as effective as possible for users. Normally, you shouldn't need to override them. However, you can override one or more of these defaults if you feel you absolutely must by creating a resource of type 'hovr'.

Using an 'hovr' resource sets the default help balloons for your application only. It does not affect the default help balloons used by other applications.

An 'hovr' resource consists of exactly nine components: a header component, a missing-items component, and seven components that specify help messages for seven standard user interface features.

Component	Element
Header	Help Manager version
	Options
	Balloon definition function
	Variation code
Missing-items help	Identifier for help message Help message for items missing from this resource or lacking help messages
Title bar help	Identifier for help message Help message for title bar of active window
Reserved	HMSkipItem identifier (always used here) No help message; reserved for future use
Close box help	Identifier for help message Help message for close box of active window
Zoom box help	Identifier for help message Help message for zoom box of active window
Help for active application's inactive windows	Identifier for help message Help message for inactive window of active application
Help for inactive application's windows	Identifier for help message Help message for window of inactive application
Help for area outside a modal dialog box or alert box	Identifier for help message Help message for area outside a modal dialog box or an alert box

Specifying Header Information for the 'hovr' Resource

As with the other help resources, specify the `HelpMgrVersion` constant for the first element of the header component of the 'hovr' resource. For the second element, specify a constant (normally, `hmDefaultOptions`) or the sum of several constants' values from the following list. (“Specifying Options in Help Resources” beginning on page 3-25 describes these options.)

```
CONST hmDefaultOptions    = 0;  {use defaults}
      hmUseSubID           = 1;  {use subrange resource IDs }
                               { for owned resources}
      hmAbsoluteCoords     = 2;  {ignore coords of window }
                               { origin and treat upper-left }
                               { corner of window as 0,0}
      hmSaveBitsNoWindow   = 4;  {don't create window; save }
                               { bits; no update event}
      hmSaveBitsWindow     = 8;  {save bits behind window }
                               { and generate update event}
```

Specify the balloon definition function and variation code (both typically 0) in the third and fourth elements, respectively, of the header component. (The balloon definition function and variation code are described in detail earlier in “Specifying Header Information for the 'hmnu' Resource” on page 3-32.)

Overriding Default Help

In the first element of the missing-items component, supply an identifier. As with the other help resources, use one of these identifiers: `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`. These identifiers are described in “Specifying the Format for Help Messages” on page 3-23. For the second element, supply either a text string for the help message or the resource ID of the resource that contains the help message.

The Help Manager expects the remaining components of an 'hovr' resource to be listed in the order previously shown. If you specify fewer than seven components in the Rez input file, the Help Manager adds components to the end of your list until there are seven. Each component that the Help Manager adds uses the message specified in the missing-items component. The Help Manager also uses the missing-items component's help message if the Rez input file specifies an empty string or a resource ID of 0 for any other component's help message.

For the first element of each of the remaining components, specify one of these identifiers: `HMStringItem`, `HMSTRResItem`, `HMStringResItem`, `HMPictItem`, `HMTEResItem`, or `HMSkipItem`. To use any of the default help balloons, use the `HMSkipItem` identifier. For the second element of each of the remaining components, supply either a text string for the help message or the resource ID of the resource that contains the help message.

Listing 3-18 shows a resource of type 'hovr' that overrides all of the default help balloons.

Listing 3-18 Rez input for an 'hovr' resource

```
resource 'hovr' (1000) {
    /*header component*/
    HelpMgrVersion,
    hmDefaultOptions, /*options*/
    0, /*the balloon definition ID*/
    0, /*variation code*/
    /*missing-items component*/
    HMStringItem { /*missing items in case this resource is */
        /* short of components*/
        "Missing override message"
    },
    {
    /*remaining components: for overriding default messages*/
        HMSkipItem { /*title bar help*/
            /*HMSkipItem means use default help balloon for this element*/
        },
        HMSkipItem { /*reserved; always specify HMSkipItem*/
        },
        HMStringItem { /*close box help*/
            "" /*empty string means use missing-items help*/
        },
        HMStringItem { /*zoom box help*/
            "Get this message if in Zoom In or Zoom Out box."
        },
        HMStringItem { /*help for active app's inactive window*/
            "Get this message if in inactive window of "
            "active application."
        },
        HMStringItem { /*help for inactive app's window*/
            "Get this message if in window of inactive application."
        },
        HMStringItem { /*help when outside of modal dialog box*/
            "Get this message if outside modal dialog box."
        }
    }
};
```

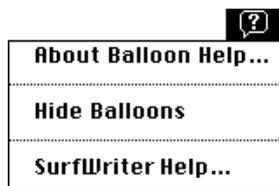
Adding Menu Items to the Help Menu

The Help menu is specific to each application, just as the File and Edit menus are. The Help menu items defined by the Help Manager should be common to all applications, but you can add your own menu items for help-related information.

If you provide your users with help information in addition to help balloons, you should append a command to the Help menu for this information. The Help menu 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 command relates to. For example, Figure 3-22 shows the Help menu with an item appended to it by the active application.

Figure 3-22 The Help menu with an appended menu item



You add items to the Help menu by using the `HMGetHelpMenuHandle` function and by providing an `'hmnu'` resource and specifying the `kHMHelpMenuID` constant as the resource ID.

The `HMGetHelpMenuHandle` function returns a copy of the handle to the Help menu. Do not use the Menu Manager function `GetMenuHandle` 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, this code adds the menu item displayed in Figure 3-22.

```
VAR
    mh:   MenuHandle;
    err:  OSErr;

BEGIN
    err := HMGetHelpMenuHandle(mh);
    IF err = noErr THEN
        IF mh <> NIL THEN
            BEGIN
                AppendMenu(mh, 'SurfWriter Help...');
            END;
        END;
    END;
```

Help Manager

```
    DrawMenuBar;
END;
```

Be sure to use an 'hmenu' resource to provide help balloons for items you've added to the Help menu. Use the `kHMHelpMenuID` constant (-16490) to specify the 'hmenu' resource ID. After the header component of the 'hmenu' resource, provide a missing-items component and then the components for your appended items. You don't provide a menu-title component here; instead, the Help Manager automatically creates the help balloons for the Help menu title and the standard Help menu items. The Help Manager also automatically adds a divider line between the end of the standard Help menu items and your appended items.

Listing 3-19 shows an 'hmenu' resource for the appended menu item shown in Figure 3-22.

Listing 3-19 Rez input for specifying help balloons for items in the Help menu

```
resource 'hmenu' (kHMHelpMenuID, "Help", purgeable) {
    HelpMgrVersion, 0, 0, 0, /*header component*/
    HMSkipItem { /*missing-items component*/
        /*no missing items, so skip to first appended menu-item */
        /* component*/
    },
    { /*first menu-item component: SurfWriter Help command*/
        HMStringResItem { /*use an 'STR#' for help messages*/
            146, 1, /*'STR#' res ID, index when item is enabled*/
            146, 2, /*'STR#' res ID, index when item is dimmed*/
            146, 3, /*'STR#' res ID, index when item is checked*/
            0, 0 /*item can't be marked*/
        },
    }
};

resource 'STR#' (146, "My help menu items' strings") {
    { /*array StringArray: six elements*/
        /*[1] enabled SurfWriter Help command help text*/
        "Offers tutorial help for the SurfWriter text processor.";
        /*[2] dimmed SurfWriter Help command help text*/
        "Offers tutorial help for the SurfWriter text processor. "
        "Not available until you open a SurfWriter document.";
        /*[3] checked SurfWriter Help command help text*/
        "Closes tutorial help for the SurfWriter text processor.";
    }
};
```

Help Manager

As previously explained in “Providing Help Balloons for Menus” beginning on page 3-27, the 'hmenu' resource allows you to specify help balloons for four states of a menu item: enabled, dimmed, enabled and checked, and enabled and marked with a symbol other than a checkmark. You cannot specify a help balloon for a Help menu item that system software dims when an alert box or a modal dialog box appears, because you don't have access to the missing-items component of the Help menu. When an alert box or a modal dialog box appears, the Help Manager displays a default help balloon for all dimmed Help menu items.

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 help is global and affects all applications.

The `MenuSelect` and `MenuKey` functions return a result with the menu ID in the high word and the menu item in the low word. Both functions return the `kHMHelpMenuID` constant (-16490) in the high word when the user chooses an appended item from the Help menu. The menu item number of the appended item is returned in the low word of the function result. The `DoMenuCommand` procedure shown in Listing 3-20 handles mouse clicks for those items defined by the application to appear in the Help menu.

Listing 3-20 Responding to the user's choice in a menu command

```
PROCEDURE DoMenuCommand (menuResult: LongInt);
VAR
    menuID, menuItem: Integer;
BEGIN
    menuID := HiWrd(menuResult);    {get menu ID}
    menuItem := LoWrd(menuResult);  {get menu item number}
    CASE menuID OF
        mApple:        DoAppleMenuCommand(menuItem);
        mFile:         DoFileMenuCommand(menuItem);
        mEdit:         DoEditMenuCommand(menuItem);
        mFont:         DoFontMenuCommand(menuItem);
        kHMHelpMenuID: DoHelpMenuCommand(menuItem);
    END;
    HiliteMenu(0);
END;
```

Help Manager

In the future, Apple may choose to add other items to the Help menu. To determine the number of items in the Help menu, call the `CountMItems` function, which is described in the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Writing Your Own Balloon Definition Function

The Help Manager takes care of positioning, sizing, and drawing your help balloons, and the standard balloon definition function provides a consistent and attractive shape to balloons across all applications.

Although it takes extra work on your part, and your balloons will not share the consistent appearance of help balloons used by the Finder and by other applications, you can create your own balloon definition function. The balloon definition function defines the appearance of the help balloon, which is a special type of window. You implement a balloon definition function by writing a window definition function that performs the tasks described in this section. The standard balloon definition function is of type `'WDEF'` with resource ID 126.

A balloon definition function is responsible for calculating the content region and structure region of the help balloon window and drawing the frame of the help balloon. The content region is the area inside the balloon frame; it contains the help message. The structure region is the boundary region of the entire balloon, including the content area and the pointer that extends from one of the help balloon’s corners. (Figure 3-4 on page 3-10 illustrates the structure regions of the eight standard help balloons.)

The Help Manager first calculates the size of the rectangle that can enclose the help message and determines where to display the help balloon. The Help Manager uses `TextEdit` to determine any word and line breaks in the help message. The Help Manager determines where to display the help balloon based on the tip and alternate rectangle.

The Help Manager then adds a system-defined distance to the size of the rectangle. This distance allows for the tip of the help balloon. Note that the tip must always align with an edge of the boundary rectangle. The Help Manager uses the resulting rectangle as the boundary rectangle for the help balloon window.

To create the help balloon, the Help Manager uses the Window Manager function `NewWindow`. The Help Manager specifies the calculated rectangle and the window definition ID as parameters to `NewWindow`.

The `NewWindow` function calls the balloon definition function in the same manner as a window definition function. See the chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for more information on writing a window definition function.

Help Manager

The `NewWindow` function calls your balloon definition function with four parameters: the variation code that specifies the shape and relative tip position of the help balloon, a pointer to the window, the action to perform, and a parameter that has variable contents depending on the action to perform.

Here's an example that shows the declaration for a balloon definition function called `MyBalloonDef`.

```
FUNCTION MyBalloonDef (variant: Integer; theBalloon: WindowPtr;
                      message: Integer;
                      param: LongInt): LongInt;
```

The `variant` parameter is the variation code used to specify the shape and position of the help balloon. You should use the same relative position for the tip of the help balloon that the standard variation codes 0 through 7 specify (see Figure 3-4 on page 3-10). This ensures that the tip of the help balloon points to the object that the help balloon describes.

The parameter `theBalloon` is a pointer to the window of the help balloon.

The `message` parameter identifies the action your balloon definition function should perform. Your balloon definition function can be sent the same messages as a window definition function, but the only ones your balloon definition function needs to process are the `wDraw` and `wCalcRgns` messages.

When your balloon definition function receives the `wCalcRgns` message, your function should calculate the content region and structure region of the help balloon. When your balloon definition function receives the `wDraw` message, your function should draw the frame of the help balloon. If you want to process other messages in your balloon definition function (for example, performing any additional initialization), you can also process the other standard 'WDEF' messages. These messages, along with the `wDraw` and `wCalcRgns` messages, are described in the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

The value of the `param` parameter depends on the value of the `message` parameter. The `wCalcRgns` and `wDraw` messages do not use this parameter.

If you want the Help Manager to use your balloon definition function, you specify its resource ID and the desired variation code either in the `HMShowBalloon` function or in the appropriate elements of the 'hmenu', 'hdlg', or 'hrct' resource. The Help Manager derives your balloon's window definition ID from its resource ID.

Help Manager Reference

This section describes the data structures, routines, and resources that are specific to the Help Manager.

The “Data Structures” section shows the data structures for the help message record and the Help Manager string list record. The “Help Manager Routines” section describes routines for determining Balloon Help status, displaying and removing help balloons, adding items to the Help menu, getting and setting the name and size of the font for help messages, setting and getting information for your application’s help resources, determining the size of a help balloon, and getting the message of a help balloon. Should you want to replace the Help Manager’s default balloon definition function and tip function with your own functions, the “Application-Defined Routines” section describes how. The “Resources” section describes the resources you can create to provide help balloons for your menus, alert and dialog boxes, and static windows; you can also create resources to override default help balloons provided by system software for various interface elements such as non-document Finder icons.

Data Structures

You can use two data structures to specify a help message to the `HMShowBalloon` function.

You use the help message record to describe the format and location of a help message. You specify the help message record as a parameter to the `HMShowBalloon` function.

If the message you want to pass to the `HMShowBalloon` function is stored in a string list (‘STR#’) resource, you use a Help Manager string list record to specify the resource ID of a string list as well as an index to one of the strings in that list. You specify a Help Manager string list record in a field of a help message record.

The Help Message Record

A help message record describes a help message. The Help Manager displays a help balloon with that message when the help message record is passed in the `aHelpMsg` parameter to the `HMShowBalloon` function. The `HMMMessageRecord` data type defines the help message record.

```
TYPE HMMMessageRecord =
RECORD
    hmmHelpType:      Integer;           {type of next field}
    CASE Integer OF
        khmmString:  (hmmString: Str255); {Pascal string}
        khmmPict:    (hmmPict: Integer);  {'PICT' resource ID}
```

Help Manager

```

khmmStringRes: (hmmStringRes: HMStringResType);
                {'STR#' resource ID }
                { and index}
khmmTEHandle:  (hmmTEHandle: TEHandle);
                {TextEdit handle}
khmmPictHandle:(hmmPictHandle: PicHandle);
                {picture handle}
khmmTERes:     (hmmTERes: Integer);    {'TEXT'/'styl' }
                { resource ID}
khmmSTRRes:    (hmmSTRRes: Integer)    {'STR ' resource ID}
END;

```

Field descriptions

hmmHelpType Specifies the data type of the next field of the help message record. You specify one of these constants for the **hmmHelpType** field.

```

CONST
khmmString      = 1; {Pascal string}
khmmPict        = 2; {'PICT' resource ID}
khmmStringRes   = 3; {'STR#' resource ID/index}
khmmTEHandle    = 4; {TextEdit handle}
khmmPictHandle  = 5; {picture handle}
khmmTERes       = 6; {'TEXT'/'styl' resource ID}
khmmSTRRes      = 7; {'STR ' resource ID}

```

Only one field follows the **hmmHelpType** field, but it can be one of seven different data types. The field that follows the **hmmHelpType** field specifies the help message itself.

hmmString Contains a Pascal string for a help message when you supply the **khmmString** constant in the **hmmHelpType** field. (This is generally not recommended; instead, you should store the help message in a resource, which makes localization easier.)

hmmPict Contains the resource ID of a 'PICT' resource for a help message when you supply the **khmmPict** constant in the **hmmHelpType** field.

hmmStringRes Contains a Help Manager string list record (described in “The Help Manager String List Record” on page 3-97) when you supply the **khmmStringRes** constant in the **hmmHelpType** field.

hmmTEHandle Specifies a TextEdit handle to a help message when you supply the **khmmTEHandle** constant in the **hmmHelpType** field.

hmmPictHandle Specifies a handle to a 'PICT' graphic containing a help message when you supply the **khmmPictHandle** constant in the **hmmHelpType** field.

Help Manager

<code>hmmTERes</code>	Specifies the resource ID of both a 'TEXT' and an 'styl' resource for a help message when you supply the <code>khmmTEHandle</code> constant in the <code>hmmHelpType</code> field.
<code>hmmSTRRes</code>	Specifies the resource ID of an 'STR' resource for a help message when you supply the <code>khmmSTRRes</code> constant in the <code>hmmHelpType</code> field.

Because the Help Manager uses the resource itself or the actual handle that you pass to `HMShowBalloon`, your 'PICT' resource should be purgeable, or, when using a handle to a 'PICT' resource, you should release the handle or dispose of it when you are finished with it.

Examples of how to use a help message record are provided in “Providing Help Balloons for Dynamic Windows” on page 3-74.

The Help Manager String List Record

To display a help message stored in an 'STR#' resource with the `HMShowBalloon` function, use the `khmmStringRes` constant in the `hmmHelpType` field of the help message record (which you pass as a parameter to `HMShowBalloon`), and supply the `hmmStringRes` field of the help message record with a Help Manager string list record. (The help message record is described in the previous section.) The `HMStringResType` data type defines a Help Manager string list record.

```
TYPE HMStringResType =
RECORD
    hmmResID:   Integer; {'STR#' resource ID}
    hmmIndex:   Integer; {index of string}
END;
```

Field descriptions

<code>hmmResID</code>	Specifies the resource ID of the 'STR#' resource.
<code>hmmIndex</code>	Specifies the index of a string within the 'STR#' resource to use for a help message.

Help Manager Routines

This section describes the routines you use to display help balloons for the windows of your application. It also describes how to determine whether help is enabled; how to get the name and size of the text font in help balloons; how to set or override the help resources used with a menu, dialog box, or window; and how to get information about the window the help balloon is displayed in.

If you want to provide help balloons for the menus, alert boxes, dialog boxes, and static windows of your application, or if you want to override default help balloons provided by system software for various interface elements (such as non-document Finder icons), you only need to create the resources containing the descriptive information. “Using the Help Manager” beginning on page 3-18 gives details on how to create these resources.

Help Manager

If help is not enabled, most Help Manager routines do nothing and return the `hmHelpDisabled` result code.

IMPORTANT

All of the Help Manager routines may move or purge memory blocks in the application heap or for some other reason should not be called from within an interrupt. Your application should not call Help Manager routines at interrupt time. *s*

Determining Balloon Help Status

The user turns on Balloon Help assistance by choosing Show Balloons from the Help menu. To determine whether help is currently enabled, you can use the `HMGetBalloons` function. If you display your own help balloons using the `HMShowBalloon` function, you should use the `HMGetBalloons` function to determine whether help is enabled before displaying a help balloon. If help is not enabled, you cannot display any help balloons. You can use the `HMIIsBalloon` function to determine whether a help balloon is currently displayed on the screen.

HMGetBalloons

To determine whether Balloon Help assistance is enabled, use the `HMGetBalloons` function.

```
FUNCTION HMGetBalloons: Boolean;
```

DESCRIPTION

The `HMGetBalloons` function returns `TRUE` if help is currently enabled and `FALSE` if help is not currently enabled. Because the `HMGetBalloons` function does not load the Help Manager into memory, it provides a fast way to determine whether Balloon Help assistance is enabled.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0003</code>

SEE ALSO

To determine whether Balloon Help assistance is available, use the `Gestalt` function as described in “Using the Help Manager” on page 3-18.

HMIIsBalloon

To determine whether the Help Manager is currently displaying a help balloon, use the `HMIIsBalloon` function.

```
FUNCTION HMIIsBalloon: Boolean;
```

DESCRIPTION

The `HMIIsBalloon` function returns `TRUE` if a help balloon is currently displayed on the screen and `FALSE` if a help balloon is not currently displayed. This function is useful for determining whether a balloon is showing before you redraw the screen. For example, you might want to determine whether a balloon is displayed so that you can remove it before opening or closing a window.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0007</code>

Displaying and Removing Help Balloons

When the user turns on Balloon Help assistance, the Help Manager automatically tracks the cursor and displays and removes help balloons as the cursor moves over hot rectangles specified in `'hrc'` resources or over display rectangles associated with menu items specified in `'hmn'` resources and items specified in `'hdlg'` resources. If you want to provide help balloons for areas not defined in these resources, then your application is responsible for tracking the cursor and displaying and removing balloons for these application-defined areas.

To display a help balloon in your application-defined area, use the `HMShowBalloon` function. If your application uses its own menu definition procedure, use the `HMShowMenuBalloon` function to display a balloon described by the standard balloon definition function. To remove a balloon that you display using `HMShowMenuBalloon`, you must use the `HMRemoveBalloon` function. To remove a balloon that you display using `HMShowBalloon`, you can either use the `HMRemoveBalloon` function to remove the help balloon, or you can let the Help Manager remove it for you.

HMShowBalloon

To display a help balloon of the content area of any window of your application, you can use the `HMShowBalloon` function. If the user has enabled Balloon Help assistance, the `HMShowBalloon` function displays a help balloon containing the message specified by the `aHelpMsg` parameter.

```
FUNCTION HMShowBalloon (aHelpMsg: HMMessageRecord;
                        tip: Point; alternateRect: RectPtr;
                        tipProc: Ptr; theProc, variant: Integer;
                        method: Integer): OSErr;
```

<code>aHelpMsg</code>	The message displayed in the help balloon.
<code>tip</code>	The location, in global coordinates, of the help balloon's tip.
<code>alternateRect</code>	A rectangle, in global coordinates, that the Help Manager uses if necessary to calculate a new tip location. If you specify a rectangle in this parameter, the Help Manager automatically calls the <code>HMRemoveBalloon</code> function to remove the help balloon when the user moves the cursor outside the area bounded by the rectangle. If you instead pass <code>NIL</code> in this parameter, your application must use the <code>HMRemoveBalloon</code> function to remove the help balloon when appropriate.
<code>tipProc</code>	The tip function called by the Help Manager before displaying the balloon. Specify <code>NIL</code> to use the Help Manager's default tip function, or supply your own tip function and point to it in this parameter.
<code>theProc</code>	The balloon definition function. To use the standard balloon definition function, specify 0 in this parameter. To use your own balloon definition function, specify the resource ID of the 'WDEF' resource containing your balloon definition function.
<code>variant</code>	The variation code for the balloon definition function. Specify 0 in the <code>variant</code> parameter to use the default help balloon position, specify a code from 1 to 7 to use one of the other positions provided by the standard balloon definition function, or specify another code to use one of the positions provided by your own balloon definition function.
<code>method</code>	A value that indicates whether the Help Manager should save the bits behind the balloon and whether to generate an update event. You can pass one of the following constants in this parameter: <code>kHMRegularWindow</code> , <code>kHMSaveBitsNoWindow</code> , or <code>kHMSaveBitsWindow</code> .

DESCRIPTION

If help is enabled, the `HMShowBalloon` function displays a help balloon with the help message you specify in the `aHelpMsg` parameter. You use global coordinates to specify the tip and the rectangle pointed to by the `alternateRect` parameter. The Help Manager calculates the location and size of the help balloon. If it fits onscreen, the Help Manager displays the help balloon using the specified tip location.

If you use the `HMShowBalloon` function to display help balloons, you must identify hot rectangles, create your own data structures to store their locations, track the cursor yourself, and call `HMShowBalloon` when the cursor moves to your hot rectangles. The Help Manager does not know the locations of your hot rectangles, so it cannot use them for moving the tip if the help balloon is placed offscreen. Instead, the Help Manager uses the alternate rectangle that you point to with the `alternateRect` parameter. Often, you specify the same coordinates for the alternate rectangle that you specify for your hot rectangle. However, you may choose to make your alternate rectangle smaller or larger than your hot rectangle. If you make your alternate rectangle smaller than your hot rectangle, you have greater assurance that the Help Manager will be able to fit the help balloon onscreen; if you specify an alternate rectangle that is larger than your hot rectangle, you have greater assurance that the balloon will not obscure the object it explains.

If you specify a rectangle in the `alternateRect` parameter, the Help Manager automatically calls `HMRemoveBalloon` to remove the balloon when the cursor leaves the area bounded by the rectangle.

If the balloon's first position is partly offscreen or if it intersects the menu bar, the Help Manager tries a combination of different balloon variation codes and different tip positions along the sides of the alternate rectangle to make the balloon fit. Figure 3-5 on page 3-11 shows what happens when the balloon's first two positions are located offscreen. If, after exhausting all possible positions, the Help Manager cannot fit the entire balloon onscreen, the Help Manager displays a balloon at the position that best fits onscreen and clips the help message to fit at this position. If the coordinates specified by both the original tip and the `alternateRect` parameter are offscreen, the Help Manager does not display the balloon at all.

If you specify `NIL` for the `alternateRect` parameter, your application is responsible for tracking the cursor and determining when to remove the balloon. The Help Manager also does not attempt to calculate a new tip location if the balloon is offscreen.

Once the Help Manager determines the location and size of the help balloon, the Help Manager calls the function pointed to by the `tipProc` parameter before displaying the balloon. Specify `NIL` in the `tipProc` parameter to use the Help Manager's default tip function.

You can supply your own tip function and point to it in the `tipProc` parameter. The Help Manager calls the tip function after calculating the location of the balloon and before displaying it. In the parameters of your tip function, the Help Manager returns the tip, the region boundary of the entire balloon, the region boundary for the content area within the balloon frame, and the variation code to be used for the balloon. This allows you to examine and possibly adjust the balloon before it is displayed.

Help Manager

The Help Manager reads the balloon definition function specified by the parameter `theProc` into memory if it isn't already in memory. If the balloon definition function can't be read into memory, the help balloon is not displayed and the `HMShowBalloon` function returns the `resNotFound` result code.

The `method` parameter specifies whether the Help Manager should save the bits behind the balloon and whether to generate an update event. You can supply one of these constants for the parameter.

```
CONST kHMRegularWindow      = 0;  {don't save bits; just update}
      kHMSaveBitsNoWindow   = 1;  {save bits; don't do update}
      kHMSaveBitsWindow     = 2;  {save bits; do update event}
```

If you specify `kHMRegularWindow`, the Help Manager draws and removes the help balloon as if it were a window. That is, when drawing the balloon, the Help Manager does not save bits behind the balloon, and, when removing the balloon, the Help Manager generates an update event. This is the standard behavior of help balloons; it is the behavior you should normally use.

If you specify `kHMSaveBitsNoWindow` in the `method` parameter, the Help Manager does not create a window for displaying the balloon. Instead, the Help Manager creates a help balloon that is more like a menu than a window. The Help Manager saves the bits behind the balloon when it creates the balloon. When it removes the balloon, the Help Manager restores the bits without generating an update event. You should use this method only in a modal environment where the bits behind the balloon cannot change from the time the balloon is drawn to the time it is removed. For example, you might specify the `kHMSaveBitsNoWindow` constant when providing help balloons for pop-up menus that overlay complex graphics, which might take a long time to redraw with an update event.

If you specify `kHMSaveBitsWindow`, the Help Manager treats the help balloon as a hybrid having properties of both a menu and a window. That is, the Help Manager saves the bits behind the balloon when it creates the balloon, and, when it removes the balloon, it both restores the bits and generates an update event. You'll rarely need this option. It is necessary only in a modal environment that might immediately change to a nonmodal environment—that is, where the bits behind the balloon are static when the balloon is drawn, but can possibly change before the balloon is removed.

`HMShowBalloon` returns the `noErr` result code if the help balloon was successfully displayed.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0B01</code>

Help Manager

RESULT CODES

noErr	0	No error; the help balloon was displayed
paramErr	-50	Error in parameter list
memFullErr	-108	Not enough room in heap zone
resNotFound	-192	Unable to read resource
hmHelpDisabled	-850	Help balloons are not enabled
hmBalloonAborted	-853	Because of constant cursor movement, the help balloon wasn't displayed
hmOperationUnsupported	-861	Invalid value passed in the method parameter

SEE ALSO

You specify the help message in the `aHelpMsg` parameter. “Providing Help Balloons for Dynamic Windows” beginning on page 3-74 shows how to specify this information.

You can supply your own tip function (as explained in the description of the `MyTip` function, which begins on page 3-130) and point to it in the `tipProc` parameter.

Figure 3-4 on page 3-10 illustrates the variation codes you can specify in the `variant` parameter and their corresponding help balloon positions for the standard balloon definition function.

If your application uses its own menu definition procedure, you can use the `HMShowMenuBalloon` function to display help balloons for the menus that your menu definition procedure manages. The `HMShowMenuBalloon` function is next.

HMShowMenuBalloon

The Help Manager displays help balloons for applications that provide 'hmenu' resources and use the standard menu definition procedure. If your application uses your own menu definition procedure, you can still use the Help Manager to display help balloons for the menus that your menu definition procedure manages. Use the `HMShowMenuBalloon` function to display balloons described by the standard balloon definition function. If you want to use your own balloon definition function from within your menu definition procedure, call the `HMShowBalloon` function (described in the previous section) and specify the `kHMSaveBitsNoWindow` constant for the `method` parameter. You can also use the `HMShowMenuBalloon` function as an alternative to creating an 'hmenu' resource for your menu.

```
FUNCTION HMShowMenuBalloon (itemNum: Integer; itemMenuID: Integer;
                             itemFlags: LongInt;
                             itemReserved: LongInt;
                             tip: Point; alternateRect: RectPtr;
                             tipProc: Ptr; theProc: Integer;
                             variant: Integer): OSErr;
```

Help Manager

<code>itemNum</code>	The number of the menu item over which the cursor is currently located. Use a positive number in the <code>itemNum</code> parameter to specify a menu item, use <code>-1</code> if the cursor is located over a divider line, or use <code>0</code> if the cursor is located over the menu title.
<code>itemMenuID</code>	The ID of the menu in which the cursor is currently located.
<code>itemFlags</code>	A long integer from the menu flags, telling whether a menu item is enabled or dimmed and whether the menu itself is enabled or dimmed. The Help Manager uses this value to determine which balloon to display from the 'hmenu' resource.
<code>itemReserved</code>	Reserved for future use by Apple. Specify <code>0</code> in this parameter.
<code>tip</code>	The tip for the help balloon. The standard menu definition procedure places the tip 8 pixels from either the right or left edge of the menu item. For menu titles, the standard menu definition procedure centers the tip at the bottom of the menu bar; you should not specify a tip with coordinates in the menu bar for any menu titles. The Help Manager uses the tip you specify in this parameter unless it places the help balloon offscreen or in the menu bar. If the tip is offscreen, the Help Manager uses the rectangle specified in the <code>alternateRect</code> parameter to calculate a new tip location.
<code>alternateRect</code>	The rectangle that the Help Manager uses to calculate a new tip location. (The standard menu definition procedure specifies the alternate rectangle as the rectangle that encloses the menu title or menu item.) If the balloon's first position is offscreen or in the menu bar, the Help Manager tries a different balloon variation code or calculates a new tip by transposing it to an opposite side of the alternate rectangle. If you specify <code>NIL</code> for the <code>alternateRect</code> parameter, the Help Manager does not attempt to calculate a new tip position when the help balloon is offscreen.
<code>tipProc</code>	The tip function that the Help Manager calls before displaying the balloon. Specify <code>NIL</code> to use the Help Manager's default tip function, or supply your own tip function and point to it in this parameter.
<code>theProc</code>	Reserved for use by Apple. Specify <code>0</code> in this parameter.
<code>variant</code>	The variation code for the standard balloon definition function. Specify <code>0</code> to use the default balloon position or a code between 1 and 7 to use one of the other standard positions shown in Figure 3-4 on page 3-10.

DESCRIPTION

The `HMShowMenuBalloon` function saves the bits behind the help balloon before displaying the help balloon. When you remove the balloon, the Help Manager restores the bits that were previously behind it.

Help Manager

After your menu definition procedure determines that the cursor is located in a menu item, you can use the `HMShowMenuBalloon` function to display any help balloons associated with that item. You must then use the `HMRemoveBalloon` function to remove the balloon when the cursor moves away from the menu item.

If you use the `HMShowMenuBalloon` function to display help balloons, you must identify hot rectangles, create your own data structures to store their locations, track the cursor yourself, and call `HMShowMenuBalloon` when the cursor moves to your hot rectangles. The Help Manager does not know the locations of your hot rectangles, so it cannot use them for moving the tip if the balloon is placed offscreen. Instead, the Help Manager uses the alternate rectangle that you point to with the `alternateRect` parameter.

Unlike the way the `alternateRect` parameter works in the `HMShowBalloon` function, specifying an alternate rectangle to `HMShowMenuBalloon` does not cause the Help Manager to track the cursor and remove the balloon for you. You must still track the cursor and use the `HMRemoveBalloon` function to remove the balloon when the cursor moves out of the area specified by the hot rectangle.

Specify `NIL` in the `tipProc` parameter to use the tip function values calculated by the Help Manager. If you supply your own tip function and specify it in the `tipProc` parameter, the Help Manager returns the tip, the region boundary of the entire balloon, the region boundary for the content area within the balloon frame, and the variation code to be used for the help balloon before displaying it. This allows you to examine and possibly adjust the balloon before it is displayed.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0E05</code>

RESULT CODES

<code>noErr</code>	0	No error; the help balloon was displayed
<code>memFullErr</code>	-108	Not enough room in heap zone
<code>hmHelpDisabled</code>	-850	Help balloons are not enabled
<code>hmBalloonAborted</code>	-853	Because of constant cursor movement, the help balloon wasn't displayed
<code>hmSameAsLastBalloon</code>	-854	Menu and item are same as last menu and item

SEE ALSO

You can supply your own tip function (as explained in the description of the `MyTip` function, which begins on page 3-130) and point to it in the `tipProc` parameter.

The `HMRemoveBalloon` function is described next.

HMRemoveBalloon

To remove a help balloon that your application displays using the function `HMShowMenuBalloon`, use the `HMRemoveBalloon` function. If your application does not specify an alternate rectangle to the `HMShowBalloon` function, use `HMRemoveBalloon` to remove the help balloon you display with `HMShowBalloon`.

```
FUNCTION HMRemoveBalloon: OSErr;
```

DESCRIPTION

The `HMRemoveBalloon` function removes any balloon that is currently visible—unless the user is using Close View and is pressing the Shift key. (This action keeps the help balloon onscreen even while the user moves away from the hot rectangle under Close View.)

If you use the `HMShowBalloon` function to display help balloons, you can either let the Help Manager track the cursor and remove the balloon when the cursor moves out of the hot rectangle, or your application can track the cursor and determine when to remove the balloon. To let the Help Manager track the cursor and remove the balloon when using the `HMShowBalloon` function, specify a rectangle in the `alternateRect` parameter. If you want your application to track the cursor and remove the balloon when using the `HMShowBalloon` function, specify `NIL` in the `alternateRect` parameter. You must then use the `HMRemoveBalloon` function to remove the balloon when the user moves the cursor outside the rectangle.

If you use the `HMShowMenuBalloon` function to display help balloons, you must always track the cursor and use the `HMRemoveBalloon` function to remove the balloon when the cursor moves out of the hot rectangle.

S WARNING

The `HMRemoveBalloon` function removes any help balloon that is currently visible, regardless of the application that displayed it. You should call `HMRemoveBalloon` only when the cursor is in the content area of your application window but not in a hot rectangle, and you should never call it when your application is in the background. s

If the user is using Close View and is pressing the Shift key, the help balloon stays onscreen even while the user moves away from the hot rectangle. The `HMRemoveBalloon` function returns a result code of `hmCloseViewActive` in this case.

If you use your own menu definition procedure, you should call `HMRemoveBalloon` when your procedure receives messages about saving or restoring bits. (These messages are described in the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.)

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0002</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error or the help balloon was removed
<code>hmHelpDisabled</code>	<code>-850</code>	Help balloons are not enabled
<code>hmNoBalloonUp</code>	<code>-862</code>	No balloon showing
<code>hmCloseViewActive</code>	<code>-863</code>	Balloon can't be removed because Close View is in use

SEE ALSO

The description of the `HMShowBalloon` function begins on page 3-100; the description of the `HMShowMenuBalloon` function begins on page 3-103.

Enabling and Disabling Balloon Help Assistance

You can enable or disable help using the `HMSetBalloons` function. If you enable or disable help, you do so for all applications. Because the setting of Balloon Help assistance should be under the user's control, in most cases you should not modify the user's setting. However, if you feel your application absolutely must enable or disable Balloon Help assistance, you can use the `HMSetBalloons` function. If you modify this setting, return it to its previous state as soon as possible.

HMSetBalloons

To enable or disable Balloon Help assistance for the user, use the `HMSetBalloons` function.

```
FUNCTION HMSetBalloons (flag: Boolean): OSErr;
```

`flag` Specifies whether help should be enabled or disabled for all applications and the system software.

DESCRIPTION

If the value of the `flag` parameter is `TRUE`, `HMSetBalloons` enables Balloon Help assistance. If the value of the `flag` parameter is `FALSE`, `HMSetBalloons` disables Balloon Help assistance. If a help balloon is showing, you must first remove it using the `HMRemoveBalloon` function before you use `HMSetBalloons` to disable Balloon Help assistance.

SPECIAL CONSIDERATIONS

When Balloon Help assistance is disabled, the Help Manager does not display help balloons for any applications. When help is disabled, the `HMShowBalloon` and `HMShowMenuBalloon` functions do not display help balloons; they return nonzero result codes.

Because the setting of Balloon Help assistance should be under the user's control, you generally should not use the `HMSetBalloons` function.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0104</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>paramErr</code>	<code>-50</code>	Error in parameter list
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone
<code>resNotFound</code>	<code>-192</code>	Unable to read resource

SEE ALSO

The description of the `HMShowBalloon` function begins on page 3-100; the description of the `HMShowMenuBalloon` function begins on page 3-103.

Adding Items to the Help Menu

The Help Manager automatically appends the Help menu when your application inserts an Apple menu into its menu bar. The Menu Manager automatically appends the Help menu to the right of all your menus and to the left of the Application menu (and to the left of the Keyboard menu if a non-Roman script system is installed).

The Help menu is specific to each application. The Help menu items defined by the Help Manager should be common to all applications, but you can append your own menu items for help-related information by using the `HMGetHelpMenuHandle` function.

HMGetHelpMenuHandle

To append items to the Help menu, use the `HMGetHelpMenuHandle` function.

```
FUNCTION HMGetHelpMenuHandle (VAR mh: MenuHandle): OSErr;
```

`mh` A copy of a handle to the Help menu.

Help Manager

DESCRIPTION

The `HMGetHelpMenuHandle` function returns in its `mh` parameter a handle to your application's help menu. With this handle, you can append items to the Help menu by using the `AppendMenu` procedure or other related Menu Manager routines. The Help Manager automatically adds the divider line that separates your items from the rest of the Help menu.

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 word and the menu item in the low word. Both functions return the `HelpMgrID` constant in the high word when the user chooses an appended item from the Help menu. The number of the appended menu item is returned in the low word of the function result. In the future, Apple Computer, Inc., may choose to add other items to the Help menu. To determine the number of items in the Help menu, call the Menu Manager function `CountMItems`.

SPECIAL CONSIDERATIONS

Do not use the Menu Manager function `GetMenuHandle` 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.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0200</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Error in parameter list
<code>memFullErr</code>	-108	Not enough room in heap zone
<code>resNotFound</code>	-192	Unable to read resource
<code>hmHelpManagerNotInited</code>	-855	Help menu not set up

SEE ALSO

“Adding Menu Items to the Help Menu” beginning on page 3-90 provides details and illustrative sample code for using `HMGetHelpMenuHandle`. The `'hmnu'` resource is described in detail in “Providing Help Balloons for Menus” beginning on page 3-27. See the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information about `AppendMenu`, `MenuSelect`, `MenuKey`, and other Menu Manager routines.

Getting and Setting the Font Name and Size

Using the `HMGetFont` and `HMGetFontSize` functions, you can get information about the font name and size currently used for text strings displayed in help balloons. Using the `HMSetFont` and `HMSetFontSize` functions, you can change the font name and size.

HMGetFont

To get information about the font that is currently used to display text in help balloons, use the `HMGetFont` function.

```
FUNCTION HMGetFont (VAR font: Integer): OSErr;
```

`font` The global font number used to display text in help balloons.

DESCRIPTION

The `HMGetFont` function returns in its `font` parameter the global font number used to display text in help balloons. `HMGetFont` returns this information only for Pascal strings stored in the help resources themselves and for strings from 'STR#' and 'STR ' resources; it does not return information about text in 'PICT' or styled text resources, or in handles to either of these resources.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$020A</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

The chapter "TextEdit" in *Inside Macintosh: Text* describes global font numbers.

HMGetFontSize

To get information about the font size that is currently used to display text in help balloons, use the `HMGetFontSize` function.

```
FUNCTION HMGetFontSize (VAR fontSize: Integer): OSErr;
```

`fontSize` The global font size used to display text in help balloons.

DESCRIPTION

The `HMGetFontSize` function returns in its `fontSize` parameter the global font size used to display text in help balloons. This information applies only to Pascal strings stored in the help resources themselves and to strings from 'STR#' and 'STR ' resources; it does not apply to text in 'PICT' or styled text resources, or in handles to either of these resources.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$020B</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

See the chapter "TextEdit" in *Inside Macintosh: Text* for detailed information about font sizes.

HMSetFont

You can use the `HMSetFont` function to specify the font used to display text in help balloons.

```
FUNCTION HMSetFont (font: Integer): OSErr;
```

`font` A global font number.

DESCRIPTION

The `HMSetFont` function sets the font for help balloons in all applications that display help balloons.

This function applies only to Pascal strings stored in the help resources themselves and to strings from 'STR#' and 'STR ' resources; it does not apply to text in 'PICT' or styled text resources, or in handles to either of these resources.

SPECIAL CONSIDERATIONS

Use this function with extreme restraint, because the default font provides a consistent look across applications. If your application uses this function to change the font name or size, the change affects all applications that display help balloons.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0108</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

See the chapter "TextEdit" in *Inside Macintosh: Text* for detailed information about fonts and font numbers.

HMSetFontSize

You can use the `HMSetFontSize` function to specify the font size used to display text in help balloons.

```
FUNCTION HMSetFontSize (fontSize: Integer): OSErr;
```

`fontSize` The global font size the Help Manager uses to display text in help balloons.

DESCRIPTION

The `HMSetFontSize` function sets the font size for help balloons in all applications and software that display help balloons. This function applies only to Pascal strings stored in the help resources themselves and to strings from 'STR#' and 'STR ' resources; it does not apply to text in 'PICT' or styled text resources, or in handles to either of these resources.

SPECIAL CONSIDERATIONS

Use this function with extreme restraint, because the default font size provides a consistent look across applications. If your application uses this function to change the font size, the change affects all applications that display help balloons.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0109</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

See the chapter "TextEdit" in *Inside Macintosh: Text* for detailed information about fonts and font sizes.

Setting and Getting Information for Help Resources

Using the `HMSetMenuResID` or `HMScanTemplateItems` function, you can set help resources for menus, dialog boxes, or windows of your application that do not currently have help resources associated with them. You can also supplement the 'hmnu' and 'hdlg' resources currently associated with the menus and dialog boxes of your application by using the `HMSetMenuResID` or `HMSetDialogResID` function. You can use the `HMGetMenuResID` function to determine the 'hmnu' resource ID associated with a menu.

When you use the `HMSetDialogResID` function, you can supplement any 'hdlg' resources that are specified in item list ('DITL') resources. The resource you specify in the `HMSetDialogResID` function adds to any help that already exists in the form of an 'hdlg' resource for the next dialog box or alert box to be displayed. You can use an 'hdlg' resource (described in “Providing Help Balloons for Items in Dialog Boxes and Alert Boxes” on page 3-51) to provide help balloons for items common to several dialog boxes and alert boxes, and you can use the `HMSetDialogResID` function to provide help balloons for items that you add to individual dialog boxes and alert boxes.

You can use the `HMGetDialogResID` function to get the resource ID of the 'hdlg' resource that will be used by the next dialog box as a result of a previous call to the `HMSetDialogResID` function. If the 'hdlg' resource currently in use has not been overridden by a call to `HMSetDialogResID`, the `HMGetDialogResID` function returns a result code of `resNotFound`.

You can use the `HMGetDialogResID` and `HMSetDialogResID` functions when displaying nested dialog boxes (although, in general, you should close one dialog box before displaying another). For example, you can save the 'hdlg' resource of the current dialog box, set a new 'hdlg' resource, display the new dialog box, and then restore the setting of the previous 'hdlg' resource when you close the second dialog box.

HMSetMenuResID

You can use the `HMSetMenuResID` function to set the 'hmnu' resource for a menu that did not previously have one or to supplement the existing 'hmnu' resource for a menu.

```
FUNCTION HMSetMenuResID (menuID, resID: Integer): OSErr;
```

menuID	The menu to associate with the 'hmnu' resource.
resID	The resource ID of the 'hmnu' resource to use for the menu specified by the menuID parameter.

DESCRIPTION

The `resID` parameter specifies the resource ID of the 'hmnu' resource to use for the menu specified by the `menuID` parameter. The menu identified by the `menuID` parameter should correspond to an existing menu in your menu list. The Help Manager maintains a list of the menus whose 'hmnu' resources you set using the `HMSetMenuResID` function.

Before your application terminates, specify `-1` in the `resID` parameter to disassociate a particular menu and an 'hmnu' resource that you previously associated using the `HMSetMenuResID` function.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$020D</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

“Providing Help Balloons for Menus You Disable for Dialog Boxes” beginning on page 3-47 describes how to use `HMSetMenuResID` to associate an alternate 'hmnu' resource with a menu that your application dims when it displays a dialog box.

HMGetMenuResID

After you use the `HMSetMenuResID` function to associate a menu with an 'hmnu' resource, you can use the `HMGetMenuResID` function to get the resource ID of the 'hmnu' resource.

```
FUNCTION HMGetMenuResID (menuID: Integer;
                        VAR resID: Integer): OSErr;
```

<code>menuID</code>	The menu for which you want the associated 'hmnu' resource. The value specified in the <code>menuID</code> parameter must have been previously associated using the <code>HMSetMenuResID</code> function.
<code>resID</code>	The resource ID of the 'hmnu' resource associated with the specified menu.

DESCRIPTION

`HMGetMenuResID` returns in its `resID` parameter the resource ID of the 'hmnu' resource associated with the menu specified by the `menuID` parameter. If the menu does not have an 'hmnu' resource that was previously set using `HMSetMenuResID`, the `HMGetMenuResID` function returns -1 in the `resID` parameter and a nonzero result code.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0314</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Unable to read resource

SEE ALSO

The `HMSetMenuResID` function is described on page 3-114.

HMScanTemplateItems

You can use the `HMScanTemplateItems` function to search for a resource of type 'hdlg' or 'hrct'.

```
FUNCTION HMScanTemplateItems (whichID, whichResFile: Integer;
                             whichType: ResType): OSErr;
```

`whichID` The resource ID of the 'hdlg' or 'hrct' resource to search for.

`whichResFile`

 The file reference number of the resource file to search.

`whichType` The type of help resource to search for—either 'hdlg' or 'hrct'.

DESCRIPTION

The `HMScanTemplateItems` function searches a resource file for resources of type 'hdlg' or 'hrct'. Specify the resource ID of the 'hdlg' or 'hrct' resource to search for in the `whichID` parameter. Specify the resource type in the `whichType` parameter. When `HMScanTemplateItems` returns the value for `noErr`, the Help Manager applies the help messages in the specified 'hdlg' or 'hrct' resource to the active window.

Help Manager

The resource file specified in the `whichResFile` parameter must already be open. Specify `-1` in the `whichResFile` parameter to search the current resource file.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0410</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>fnOpnErr</code>	<code>-38</code>	File not open
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone
<code>resNotFound</code>	<code>-192</code>	Unable to read resource

SEE ALSO

If you want the capability that `HMScanTemplateItems` provides without modifying your code, you can add a `HelpItem` item to your item list ('DITL') resources or add an 'hwin' resource—as described in “Using a Help Item Versus Using an 'hwin' Resource” on page 3-63 and in “Associating Help Resources With Static Windows” on page 3-68.

HMSetDialogResID

You can use the `HMSetDialogResID` function to set the 'hdlg' resource that specifies help balloons for the next dialog box or alert box.

```
FUNCTION HMSetDialogResID (resID: Integer): OSErr;
```

`resID` The resource ID of the 'hdlg' resource to use when your application displays the next dialog box or alert box.

DESCRIPTION

The `HMSetDialogResID` function uses the 'hdlg' resource specified in the `resID` parameter to supplement whatever 'hdlg' resource might already be associated with the next dialog box or alert box that you display. `HMSetDialogResID` supplements the help messages specified by a `HelpItem` item in the next dialog or alert box's item list ('DITL') resource. Specify `-1` in the `resID` parameter to reset or clear a previous call to the `HMSetDialogResID` function.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$010C</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

You typically use `HMSetDialogResID` in conjunction with the `HMGetDialogResID` function, which is described in the following section.

HMGetDialogResID

You can use the `HMGetDialogResID` function to get the resource ID of the 'hdlg' resource that will be used by the next dialog box as a result of a previous call to the `HMSetDialogResID` function.

```
FUNCTION HMGetDialogResID (VAR resID: Integer): OSErr;
```

`resID` **The resource ID of the last 'hdlg' resource set with the `HMSetDialogResID` function.**

DESCRIPTION

The `HMGetDialogResID` function returns in its `resID` parameter the resource ID of the last 'hdlg' resource set with the `HMSetDialogResID` function.

You can use the `HMGetDialogResID` and `HMSetDialogResID` functions when your application displays nested dialog boxes (although you should generally close one dialog box before displaying another). For example, you can save the 'hdlg' resource of the current dialog box, set a new 'hdlg' resource, display the new dialog box, and then restore the setting of the previous 'hdlg' resource when you close the second dialog box.

If the 'hdlg' resource currently in use was not set by a call to the `HMSetDialogResID` function, the `HMGetDialogResID` function returns a result code of `resNotFound`.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0213</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone
<code>resNotFound</code>	<code>-192</code>	Unable to read resource

SEE ALSO

You typically use `HMGetDialogResID` in conjunction with the `HMSetDialogResID` function, which is described on page 3-117.

Determining the Size of a Help Balloon

If your application does extensive drawing, the Help Manager provides three functions that may be helpful for determining the dimensions of your help balloons before displaying them. Then you can ensure that your help balloons don't obscure an area that requires an inordinate amount of time to update.

To get the size of a help balloon before the Help Manager displays it, use the `HMBalloonRect` or `HMBalloonPict` function. To get the size of the currently displayed help balloon, use the `HMGetBalloonWindow` function.

HMBalloonRect

To get information about the size of a help balloon before the Help Manager displays it, you can use the `HMBalloonRect` function.

```
FUNCTION HMBalloonRect (aHelpMsg: HMMMessageRecord;
                       VAR coolRect: Rect): OSErr;
```

`aHelpMsg` The help message for the help balloon.

`coolRect` The coordinates of the rectangle that encloses the help message. The upper-left corner of the rectangle has the coordinates (0,0).

DESCRIPTION

The `HMBalloonRect` function calculates the coordinates that the Help Manager uses for a particular balloon, permitting you to specify the help message for a help balloon and then obtaining the size (but not the position) of the rectangle used for the balloon. Note that the `HMBalloonRect` function does not display the help balloon.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$040E</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>paramErr</code>	<code>-50</code>	Error in parameter list
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

The `aHelpMsg` parameter is of data type `HMMessageRecord`, which is described in “Providing Help Balloons for Dynamic Windows” beginning on page 3-74.

HMBalloonPict

To get a handle to a picture before displaying it in a help balloon, use the `HMBalloonPict` function.

```
FUNCTION HMBalloonPict (aHelpMsg: HMMessageRecord;
                       VAR coolPict: PicHandle): OSErr;
```

<code>aHelpMsg</code>	The help message for the help balloon; in this case, a picture.
<code>coolPict</code>	A handle to the picture that the Help Manager will use if you later choose to display the help balloon.

DESCRIPTION

The `HMBalloonPict` function does not display the help balloon; it returns a handle to the picture that the Help Manager will use if you later choose to display a help balloon with the specified help message.

The `pictFrame` field of the picture handle in the `coolPict` parameter contains the same rectangle as the rectangle obtained from the `HMBalloonRect` function. The rectangle specifies the display rectangle that surrounds the picture.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$040F</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>paramErr</code>	<code>-50</code>	Error in parameter list
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone

SEE ALSO

The `aHelpMsg` parameter is of data type `HMMessageRecord`. “Providing Help Balloons for Dynamic Windows” beginning on page 3-74 describes the fields of this record.

HMGetBalloonWindow

The Help Manager displays help balloons in special windows; to get a pointer to the window record of the currently displayed help balloon, use the `HMGetBalloonWindow` function.

```
FUNCTION HMGetBalloonWindow (VAR window: WindowPtr): OSErr;
```

`window` A pointer to the window record for the currently displayed help balloon.

DESCRIPTION

In its `window` parameter, `HMGetBalloonWindow` returns a pointer to the window record for the currently displayed help balloon. The window record contains a graphics port record, which in turn defines the port’s rectangle.

If no help balloon is currently displayed, the `HMGetBalloonWindow` function returns `NIL` in the `window` parameter. The `HMGetBalloonWindow` function also returns `NIL` for balloons created with the `HMShowMenuBalloon` function because no windows are created; likewise, `NIL` is returned for balloons created with the `HMShowBalloon` function when the `kHMSaveBitsNoWindow` constant is specified as the method parameter.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0215</code>

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough room in heap zone

SEE ALSO

The description of the `HMShowMenuBalloon` function begins on page 3-103; the description of the `HMShowBalloon` function begins on page 3-100.

Getting the Message of a Help Balloon

Using the `HMExtractHelpMsg` and `HMGetIndHelpMsg` functions, you can extract information from existing help resources.

You can use `HMExtractHelpMsg` to extract the help messages specified in existing help resources. You might find this useful if you have duplicate commands and you want to store help messages in only one resource. For example, if you have a dialog box that replicates portions of a pull-down menu, you could specify help messages in the 'hmenu' resource for the pull-down menu, and use `HMExtractHelpMsg` to extract those help messages to use with the related items in the dialog box's 'hdlg' resource.

HMExtractHelpMsg

You can use the `HMExtractHelpMsg` function to extract the help balloon messages from existing help resources.

```
FUNCTION HMExtractHelpMsg (whichType: ResType;
                           whichResID, whichMsg,
                           whichState: Integer;
                           VAR aHelpMsg: HMMessageRecord): OSErr;
```

whichType The type of help resource. You can use one of these constants: `kHMMenuResType`, `kHMDialogResType`, `kHMRectListResType`, `kHMOverrideResType`, or `kHMFinderApplResType`.

whichResID The resource ID of the help resource whose help message you wish to extract.

whichMsg The index of the component you wish to extract. The header and missing-items components don't count as components to index, because this function always skips those two components. For help resources that include both header and missing-items components, specify 1 to get the help messages contained in a help resource's menu-title component.

Help Manager

whichState

For menu items and items in alert or dialog boxes, specifies the state of the item whose message you wish to extract. Use one of the following constants: `kHMEnabledItem`, `kHMDisabledItem`, `kHMCheckedItem`, or `kHMOtherItem`.

`aHelpMsg` A help message record.

DESCRIPTION

The `HMExtractHelpMsg` function returns in its `aHelpMsg` parameter the help message for an item in a specified state.

The `whichType` parameter identifies the type of resource from which you are extracting the help message. You can use one of these constants for the `whichType` parameter.

```
CONST kHMMenuResType      = 'hmnu'; {menu help resource type}
      kHMDialogResType    = 'hdlg'; {dialog help resource type}
      kHMWindListResType  = 'hwin'; {window help resource type}
      kHMRectListResType  = 'hrct'; {rectangle help resource type}
      kHMOverrideResType  = 'hovr'; {help override resource }
                               { type}
      kHMFinderApplResType = 'hldr'; {application icon help }
                               { resource type}
```

The `whichState` parameter specifies the state of the item whose message you want to extract. You can use one of these constants for the `whichState` parameter.

```
CONST kHMEnabledItem     = 0; {enabled state for menu items; }
                               { contrlHilite value of 0 for }
                               { controls}
      kHMDisabledItem     = 1; {disabled state for menu items; }
                               { contrlHilite value of 255 for }
                               { controls}
      kHMCheckedItem      = 2; {enabled-and-checked state for }
                               { menu items; contrlHilite value }
                               { of 1 for controls that are "on"}
      kHMOtherItem        = 3; {enabled-and-marked state for menu }
                               { items; contrlHilite value }
                               { between 2 and 253 for controls}
```

For the `kHMRectListResType`, `kHMOverrideResType`, and `kHMFinderApplResType` resource types—which don't have states—supply the `kHMEnabledItem` constant for the `whichState` parameter.

Help Manager

The application-defined procedure shown in Listing 3-21 extracts the help balloon message from the 'hmenu' resource with a resource ID of 128. A value of 1 is supplied as the `whichMsg` parameter to retrieve information about the resource's first component (after the header and missing-items components, that is), which is the menu title. The menu title has four possible states; to retrieve the help message for the menu title in its dimmed state, the constant `kHMDisabledItem` is used for the `whichState` parameter. The help message record returned in `aHelpMsg` is then passed to `HMShowBalloon`, which displays the message in a balloon whose tip is located at the point specified in the `tip` parameter.

Listing 3-21 Using the `HMExtractHelpMsg` function

```
FUNCTION MyShowBalloonForDimMenuItem: OSErr;
VAR
    aHelpMsg:      HMMessageRecord;
    tip:           Point;
    alternateRect: Rect;
    err:           OSErr;
BEGIN
    err := HMExtractHelpMsg(kHMMenuResType, 128, 1,
                           kHMDisabledItem, aHelpMsg);

    IF err = noErr THEN
        {be sure to assign a tip and rectangle coordinates here}
        err := HMShowBalloon(aHelpMsg, tip, alternateRect,
                             NIL, 0, 0, kHMRegularWindow);
    MyShowBalloonForDimMenuItem:= err;
END;
```

To retrieve all of the help messages for a given resource, set `whichMsg` to 1 and make repeated calls to `HMExtractHelpMsg`, incrementing `whichMsg` by 1 on each subsequent call until it returns the `hmSkippedBalloon` result code.

SPECIAL CONSIDERATIONS

If `HMCompareItem` appears as a component of an 'hmenu' resource that you're examining, neither this function nor `HMGetIndHelpMsg` performs a comparison against the current name of any menu item. Instead, these functions return the messages listed in your `HMCompareItem` components in the order in which they appear in the 'hmenu' resource.

When supplying an index for the `whichMsg` parameter, don't count the header component or the missing-items component as components to index. This function always skips both components; therefore, for help resources that include both header and missing-items components, specify 1 to get the help messages contained in a help resource's menu-title component.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$0711</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>paramErr</code>	<code>-50</code>	Error in parameter list
<code>memFullErr</code>	<code>-108</code>	Not enough room in heap zone
<code>resNotFound</code>	<code>-192</code>	Unable to read resource
<code>hmSkippedBalloon</code>	<code>-857</code>	No help message to fill in
<code>hmWrongVersion</code>	<code>-858</code>	Wrong version of Help Manager resource
<code>hmUnknownHelpType</code>	<code>-859</code>	Help message record contained a bad type

SEE ALSO

The `aHelpMsg` parameter is of data type `HMMessageRecord`. “Providing Help Balloons for Dynamic Windows” beginning on page 3-74 describes the fields of the help message record.

HMGetIndHelpMsg

To extract the help messages in existing help resources as well as additional information regarding the help resource, such as its variation code, tip location, and so on, use the `HMGetIndHelpMsg` function.

```
FUNCTION HMGetIndHelpMsg (whichType: ResType;
                          whichResID, whichMsg,
                          whichState: Integer;
                          VAR options: LongInt; VAR tip: Point;
                          VAR altRect: Rect; VAR theProc: Integer;
                          VAR variant: Integer;
                          VAR aHelpMsg: HMMessageRecord;
                          VAR count: Integer): OSErr;
```

whichType The type of help resource. You can use one of these constants: `kHMMenuResType`, `kHMDialogResType`, `kHMRectListResType`, `kHMOverrideResType`, or `kHMFinderApplResType`.

whichResID The resource ID of the help resource whose help message you wish to extract.

Help Manager

<code>whichMsg</code>	The index of the component you wish to extract. The header and missing-items components don't count as components to index, because this function always skips those two components. For help resources that include both header and missing-items components, specify 1 to get the help messages contained in a help resource's menu-title component.
<code>whichState</code>	For menu items and items in alert and dialog boxes, specifies the state of the item whose message you wish to extract. Use one of the following constants: <code>kHMEnabledItem</code> , <code>kHMDisabledItem</code> , <code>kHMCheckedItem</code> , or <code>kHMOtherItem</code> .
<code>options</code>	The value of the <code>options</code> element of the help resource.
<code>tip</code>	The coordinates of the help balloon's tip location.
<code>altRect</code>	The coordinates of the help balloon's alternate rectangle.
<code>theProc</code>	The resource ID of the help balloon's 'WDEF' resource.
<code>variant</code>	The balloon definition function's variation code.
<code>aHelpMsg</code>	The help message.
<code>count</code>	The number of components defined in the resource (not counting the header and missing-items components).

DESCRIPTION

Like the `HMExtractHelpMsg` function, the `HMGetIndHelpMsg` function returns in its `aHelpMsg` parameter the help message for an item in a specified state. The `HMGetIndHelpMsg` function uses additional parameters to return even more information about the help balloon than does `HMExtractHelpMsg`.

To retrieve all of the help balloon messages and related information for a given resource, set `whichMsg` to 1 and make repeated calls to `HMGetIndHelpMsg`, incrementing `whichMsg` by 1 on each subsequent call until it returns the `hmSkippedBalloon` result code.

The `whichType` parameter identifies the type of resource from which you are extracting the help message. You can use one of these constants for the `whichType` parameter.

```

CONST kHMMenuResType      = 'hmnu';{menu help resource type}
      kHMDialogResType    = 'hdlg';{dialog help resource type}
      kHMWindListResType  = 'hwin';{window help resource type}
      kHMRectListResType  = 'hrct';{rectangle help resource type}
      kHMOverrideResType  = 'hovr';{help override resource }
                              { type}
      kHMFinderApplResType = 'hfdr';{application icon help }
                              { resource type}

```

Help Manager

The `whichState` parameter specifies the state of the item whose message you want to extract. You can use one of these constants for the `whichState` parameter.

```
CONST kHMEnabledItem      = 0; {enabled state for menu items; }
                             { ctrlHilite value of 0 for }
                             { controls}
      kHMDisabledItem     = 1; {disabled state for menu items; }
                             { ctrlHilite value of 255 for }
                             { controls}
      kHMCheckedItem      = 2; {enabled-and-checked state for }
                             { menu items; ctrlHilite value }
                             { of 1 for controls that are "on" }
      kHMOtherItem        = 3; {enabled-and-marked state for menu }
                             { items; ctrlHilite value }
                             { between 2 and 253 for controls}
```

For the `kHMRectListResType`, `kHMOverrideResType`, and `kHMFinderApplResType` resource types—which don't have states—supply the `kHMEnabledItem` constant for the `whichState` parameter.

SPECIAL CONSIDERATIONS

If `HMCompareItem` appears as a component of an 'hmenu' resource that you're examining, neither this function nor `HMExtractHelpMsg` performs a comparison against the current name of any menu item. Instead, these functions return the messages listed in your `HMCompareItem` components in the order in which they appear in the 'hmenu' resource.

When supplying an index for the `whichMsg` parameter, don't count the header component or the missing-items component as components to index. This function always skips both components; therefore, for help resources that include both header and missing-items components, specify 1 to get the help messages contained in a help resource's menu-title component.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_Pack14</code>	<code>\$1306</code>

Help Manager

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
hmSkippedBalloon	-857	No help message to fill in
hmWrongVersion	-858	Wrong version of Help Manager resource
hmUnknownHelpType	-859	Help message record contained a bad type

SEE ALSO

The `aHelpMsg` parameter is of data type `HMMessageRecord`. “Providing Help Balloons for Dynamic Windows” beginning on page 3-74 describes the fields of the help message record.

Application-Defined Routines

A balloon definition function is responsible for calculating the content region and structure region of the help balloon window and drawing the frame of the help balloon. The Help Manager takes care of positioning, sizing, and drawing your help balloons, and the standard balloon definition function provides a consistent and attractive shape to balloons across all applications. Though it takes extra work on your part, and your balloons will not share the consistent appearance of help balloons used by the Finder and by other applications, you can create your own balloon definition function, described in this section as `MyBalloonDef`.

When you use the `HMShowBalloon` and `HMShowMenuBalloon` functions to display help balloons, you pass a pointer to a tip function in the `tipProc` parameter. Normally, you supply `NIL` in this parameter to use the Help Manager’s default tip function. However, you can also supply your own tip function, described in this section as `MyTip`. The Help Manager calls your tip function after calculating the size and the location of a help balloon and before displaying it. This allows you to examine and, if necessary, adjust the balloon before it is displayed. For example, if you determine that the help balloon would obscure an object that requires extensive redrawing, you might use a different variation code to move the balloon.

MyBalloonDef

Here's a sample declaration for a balloon definition function called `MyBalloonDef`.

```
FUNCTION MyBalloonDef (variant: Integer; theBalloon: WindowPtr;
                      message: Integer;
                      param: LongInt): LongInt;
```

<code>variant</code>	The variation code used to specify the shape and position of the help balloon. You should use the same relative position for the tip of the help balloon that the standard variation codes 0 through 7 specify. This ensures that the tip of the help balloon points to the object that the help balloon describes.
<code>theBalloon</code>	A pointer to the window of the help balloon.
<code>message</code>	Identifies the action your balloon definition function should perform. Your balloon definition function can be sent the same messages as a window definition function, but the only ones your balloon definition function needs to process are the <code>wCalcRgns</code> and <code>wDraw</code> messages. When your balloon definition function receives the <code>wCalcRgns</code> message, your function should calculate the content region and structure region of the help balloon. When your balloon definition function receives the <code>wDraw</code> message, your function should draw the frame of the help balloon. If you want to process other messages in your balloon definition function (for example, performing any additional initialization), you can also process the other standard 'WDEF' messages.
<code>param</code>	As with a window definition function, the value of this parameter depends on the value of the <code>message</code> parameter. Because this parameter is not used by the <code>wCalcRgns</code> and <code>wDraw</code> messages, your balloon definition function should disregard the value of this parameter.

DESCRIPTION

Your balloon definition function must define the appearance of the help balloon, which is a special type of window. You can implement your own balloon definition function by writing a window definition function that performs the tasks described in this section. (The standard balloon definition function is of type 'WDEF' with resource ID 126.)

Your balloon definition function is also responsible for calculating the content region and structure region of the help balloon window and drawing the frame of the help balloon. The content region is the area inside the balloon frame; it contains the help message. The structure region is the boundary region of the entire balloon, including the content area and the pointer that extends from one of the help balloon's corners.

If you want the Help Manager to use your balloon definition function, you specify its resource ID and the desired variation code either in the `HMShowBalloon` function or in

Help Manager

the appropriate elements of the 'hmnu', 'hdlg', or 'hrct' resource. The Help Manager derives your balloon's window definition ID from its resource ID.

SEE ALSO

In the `variant` parameter, you should use the same relative position for the tip of the help balloon that the standard variation codes 0 through 7 specify, as illustrated in Figure 3-4 on page 3-10.

The `wCalcRgns` and `wDraw` messages are described in the chapter "Window Manager" of *Inside Macintosh: Macintosh Toolbox Essentials*.

MyTip

Here's a sample declaration of a tip function called `MyTip`.

```
FUNCTION MyTip (tip: Point; structure: RgnHandle; VAR r: Rect;
               VAR variant: Integer): OSErr;
```

<code>tip</code>	The location of the help balloon tip.
<code>structure</code>	A handle to the help balloon's region structure. The Help Manager returns this value. The structure region is the boundary region of the entire balloon, including the content area and the pointer that extends from one of the help balloon's corners.
<code>r</code>	The coordinates of the help balloon's content region. The content region is the area inside the balloon frame; it contains the help message. If this rectangle is not appropriate for the current screen display, you can specify different coordinates in this parameter.
<code>variant</code>	Variation code to be used for the help balloon. If this variation code is not appropriate for the current screen display, you can specify different coordinates in this parameter.

DESCRIPTION

Before displaying a help balloon created with the `HMShowBalloon` or `HMShowMenuBalloon` function, the Help Manager calls this function if you point to it in the `tipProc` parameter of either `HMShowBalloon` or `HMShowMenuBalloon`. The Help Manager returns the location of the help balloon tip, a handle to the help balloon's region structure, the coordinates of its content region, and the variation code to be used for the help balloon. If the help balloon that `HMShowBalloon` or `HMShowMenuBalloon` initially calculates is not appropriate for your current screen display, you can make minor adjustments to it by specifying a different rectangle in the `r` parameter (in which case the Help Manager automatically adjusts the `structure` parameter so that the entire balloon is larger or smaller as necessary) or by specifying a different variation code in the `variant` parameter.

Help Manager

If you need to make a major adjustment to the help balloon, return the `hmBalloonAborted` result code and call `HMShowBalloon` or `HMShowMenuBalloon` with appropriate new parameter values. To use the values returned in your tip function's parameters, return the `noErr` result code.

Listing 3-22 shows an example of using a tip function to refrain from displaying a balloon if it obscures an area of the screen that requires extensive drawing.

Listing 3-22 Using a tip function

```

VAR
    temprect:          Rect;
    DontObscureRect:  Rect;
    tip:              Point;
    structure:        RgnHandle;
    aHelpMsg:         HMMessageRecord;

BEGIN
    {be sure to determine DontObscureRect and fill in aHelpMsg}
    IF HMShowBalloon(aHelpMsg, tip, NIL, @MyTip, 0, 0,
                    kHMRegularwindow) = noErr
    THEN
        {test whether balloon obscures complex graphic }
        { in DontObscureRect}
        IF SectRect(structure^^.rgnBBox, DontObscureRect,
                    temprect) THEN
            {don't show this balloon but call HMShowBalloon later}
            MyTip := hmBalloonAborted
        ELSE {use the balloon as calculated by the Help Manager}
            MyTip := noErr;
    END;

```

SEE ALSO

Figure 3-4 on page 3-10 illustrates the structure regions and positions of the eight standard help balloons.

The `HMShowBalloon` function is described on page 3-100, and the `HMShowMenuBalloon` function is described on page 3-103.

Resources

This section describes the resources that the Help Manager uses to size, position, and draw help balloons for menus, alert and dialog boxes, static windows, non-document Finder icons, and several default help balloons provided by system software.

Help resources generally specify help messages, a balloon definition function, a variation code, and, when necessary, the balloon tip and either a hot rectangle or an alternate rectangle. The Help Manager uses this information as appropriate when drawing help balloons. These help resources are

- n the menu help ('hmnu') resource, which provides help balloons for menus and menu items
- n the dialog-item help ('hdlg') resource, which provides help balloons for items in dialog boxes and alert boxes
- n the rectangle help ('hrct') resource, which associates a help balloon with a hot rectangle in a static window
- n the window help ('hwin') resource, which associates an 'hrct' or 'hdlg' resource with a hot rectangle in a window or with an item in a dialog box or alert box
- n the Finder icon help ('hfdr') resource, which provides a custom help balloon for your application icon
- n the default help override ('hovr') resource, which overrides the help messages of default help balloons provided in system software

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 Help Manager” beginning on page 3-18 for detailed information.

The Menu Help Resource

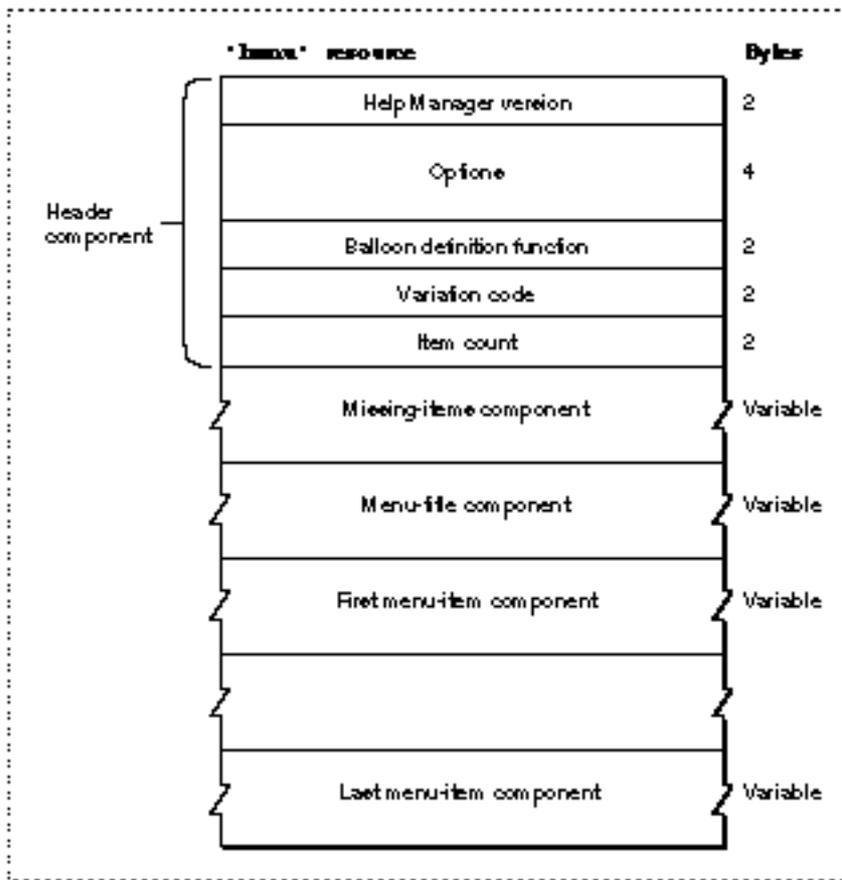
To provide help balloons for a menu—pull-down, pop-up, or hierarchical—that uses the standard menu definition procedure, you can create a menu help resource. A menu help resource is a resource of type 'hmnu'; in it, you specify help balloons for the menu title and for each item in the menu. You create a separate 'hmnu' resource for each menu. All 'hmnu' resources must have resource IDs greater than 128.

The format of a Rez input file for an 'hmnu' resource differs from its compiled output form. This section describes the structure of a Rez-compiled 'hmnu' resource. If you are concerned only with creating 'hmnu' resources, see “Providing Help Balloons for Menus” beginning on page 3-27. That section gives a detailed description, using several code samples, of how to use Rez input files to create 'hmnu' resources.

Help Manager

An 'hmnv' resource consists of a header component, a missing-items component, a menu-title component, and a variable number of menu-item components. Figure 3-23 shows the general structure of a compiled 'hmnv' resource.

Figure 3-23 Structure of a compiled menu help ('hmnv') resource



If you examine a compiled version of an 'hmnv' resource, you find that the header component consists of the following elements:

- n Help Manager version. The version of the Help Manager to use; specified in a Rez input file with the `HelpMgrVersion` constant.
- n Options. The sum of the values of available options, described in “Specifying Options in Help Resources” beginning on page 3-25.
- n Balloon definition function. The resource ID of the window definition function used for drawing the help balloon. The standard balloon definition function is of type 'WDEF' with resource ID 126; this can be specified by the number 0 in the Rez input file.

Help Manager

- n Variation code. A number signifying the preferred position of the help balloon relative to the hot rectangle. The balloon definition function draws the frame of the help balloon based on the variation code specified here. The eight variation codes and how they affect the standard balloon definition function are illustrated in Figure 3-4 on page 3-10.
- n Item count. The number of remaining components—including the missing-items, menu-title, and menu-item components—defined in the rest of this resource.

The Help Manager identifies each component by its order in the resource. The missing-items component always follows the header component of an 'hmenu' resource. The menu-title component always follows the missing-items component. Then a variable number of menu-item components are stored in this resource. The Help Manager determines the end of the 'hmenu' resource by using the item count information in the header component.

The structures of the missing-items component, the menu-title component, and the menu-item components depend on identifiers specified inside the components. The identifiers used in a Rez input file are described in “Specifying the Format for Help Messages” on page 3-23.

The missing-items component, the menu-title component, and the menu-item components can each specify four different help messages:

- n First help message.
 - n In the missing-items component, this is the help message for missing enabled items.
 - n In the menu-title component, this is the help message for the enabled menu title.
 - n In all subsequent menu-item components, this is the help message for enabled menu items.
- n Second help message.
 - n In the missing-items component, this is the help message for missing items that are dimmed by the application.
 - n In the menu-title component, this is the help message for the menu title when the application dims it.
 - n In all subsequent menu-item components, this is the help message for menu items when the application dims them.
- n Third help message.
 - n In the missing-items component, this is the help message for missing enabled-and-checked items.
 - n In the menu-title component, this is the help message for the menu title when system software dims it at the appearance of an alert box or a modal dialog box.
 - n In all subsequent menu-item components, this is the help message for enabled-and-checked menu items.

Help Manager

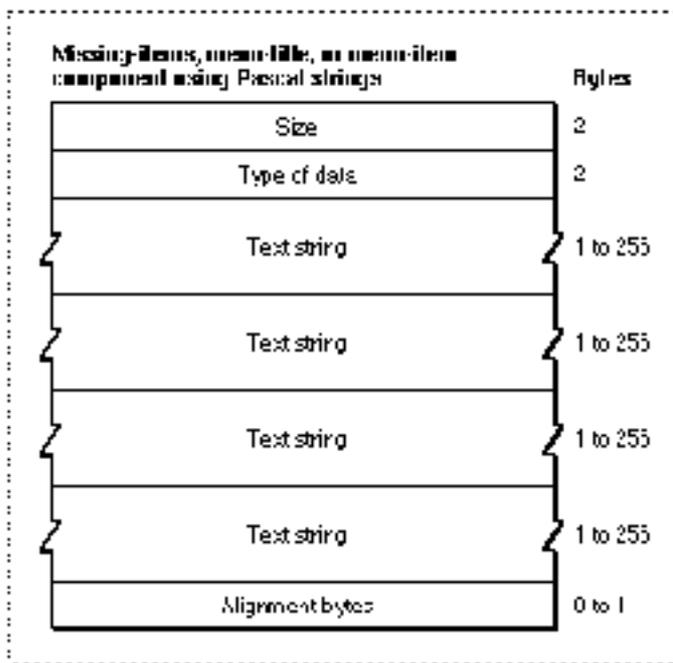
- n Fourth help message.
 - n In the missing-items component, this is the help message for missing enabled-and-marked items.
 - n In the menu-title component, this is the help message for all menu items when system software dims them at the appearance of an alert box or a modal dialog box.
 - n In all subsequent menu-item components, this is the help message for enabled-and-marked menu items.

An empty string or a resource ID of 0 for any messages in the menu-title or menu-item components causes the Help Manager to use the appropriate help message contained in the missing-items component.

Since they all adhere to the formats specified by the previously described identifiers, the missing-items component, the menu-title component, and the menu-item components can have similar structures. The Help Manager determines the end of a component by examining its length, which is stored in the first 2 bytes of the component.

Figure 3-24 shows the structure of a component that stores its help messages as Pascal strings within the 'hmnv' resource itself.

Figure 3-24 Structure of an 'hmnv' component compiled with the `HMStringItem` identifier



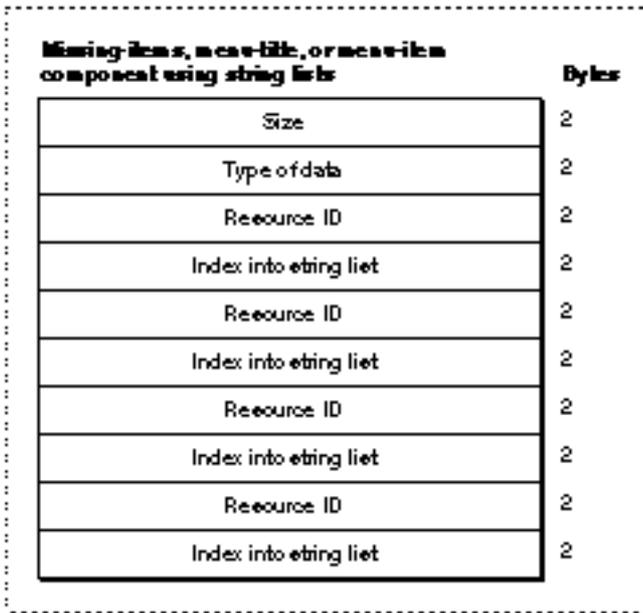
Help Manager

If you examine a compiled version of an 'hmnv' resource, you find that a component identified in a Rez input file by the `HMStringItem` identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 1 is specified here when the help messages are stored as Pascal strings within this component.
- n Text string. The first help message (as previously described).
- n Text string. The second help message (as previously described).
- n Text string. The third help message (as previously described).
- n Text string. The fourth help message (as previously described).
- n Alignment bytes. Zero or one bytes used to make the previous text strings end on a word boundary.

Figure 3-25 shows the structure of an 'hmnv' component that specifies its help messages as text strings stored in string list ('STR#') resources.

Figure 3-25 Structure of an 'hmnv' component compiled with the `HMStringResItem` identifier



Help Manager

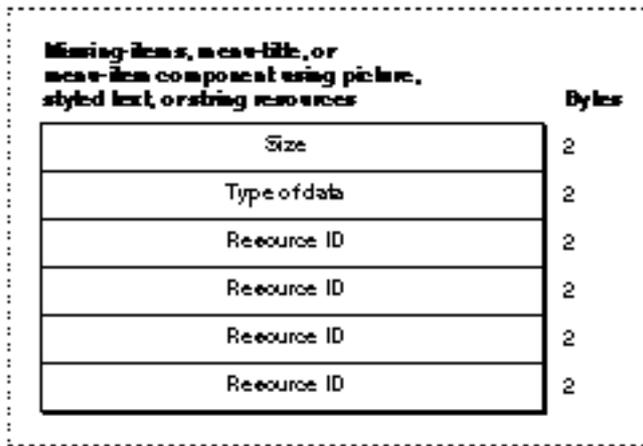
If you examine a compiled version of an 'hmnu' resource, you find that a component identified in a Rez input file by the HMStringResItem identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 3 is specified here when the help messages for this component are stored in string list ('STR#') resources.
- n Resource ID. The resource ID of an 'STR#' resource.
- n Index into the string list resource. A number used as an index to a particular text string within the 'STR#' resource. This text string is used for the first help message (as previously described).

Three more pairs of resource IDs/index numbers follow. The text strings that these pairs refer to are used for the second, third, and fourth help messages, respectively.

Figure 3-26 shows the structure of an 'hmnu' component that specifies its help messages in picture ('PICT') resources, styled text ('TEXT' and 'styl') resources, or string ('STR ') resources.

Figure 3-26 Structure of an 'hmnu' component compiled with the HMPicItem, HMTEResItem, or HMSTRResItem identifier



Help Manager

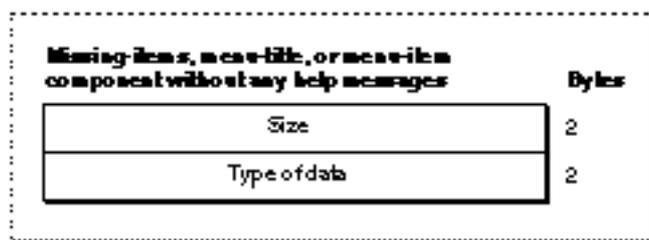
If you examine a compiled version of an 'hmnU' resource, you find that a component identified in a Rez input file by either the `HMPictItem`, `HMTEResItem`, or `HMSTRResItem` identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data.
 - n The value 2 is specified here when the help messages for this component are stored in 'PICT' resources.
 - n The value 6 is specified here when the help messages for this component are stored as styled text—that is, in both 'TEXT' and 'styl' resources.
 - n The value 7 is specified here when the help messages for this component are stored in 'STR' resources.
- n Resource ID.
 - n The resource ID of a 'PICT' resource when the value 2 is specified as the type of data. The Help Manager uses the picture contained in this resource for the first help message (as previously described).
 - n The resource ID common to both a 'TEXT' and an 'styl' resource when the value 6 is specified as the type of data. The Help Manager uses the styled text specified by these resources for the first help message.
 - n The resource ID of an 'STR' resource when the value 7 is specified as the type of data. The Help Manager uses the text contained in this resource for the first help message.

Three more resource IDs follow; the Help Manager uses these resources (either 'PICT', 'TEXT' and 'styl', or 'STR') for the second, third, and fourth help messages, respectively (as previously described).

Figure 3-27 shows the structure of an 'hmnU' component that specifies no help messages.

Figure 3-27 Structure of an 'hmnU' component compiled with the `HMSkipItem` identifier



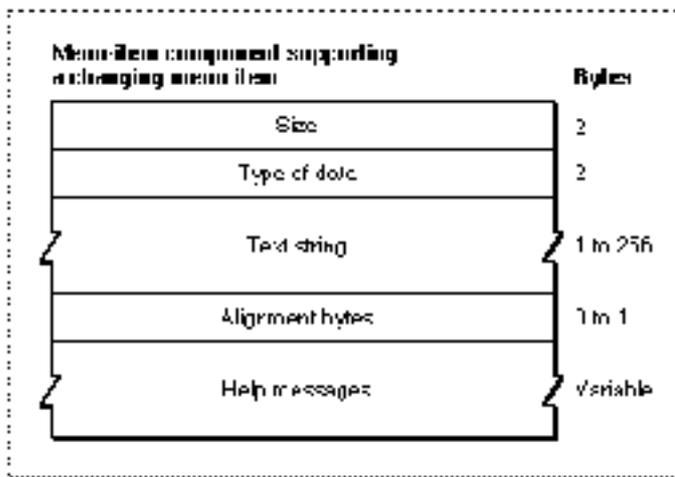
If you examine a compiled version of an 'hmnU' resource, you find that a component identified by the `HMSkipItem` identifier consists of the following elements:

- n Size. The value 4, for the number of bytes contained in this component.
- n Type of data. The value 256.

Help Manager

For menu-item components, two additional identifiers are available: `HMCompareItem` and `HMNamedResourceItem`. When the `HMCompareItem` identifier is specified, the Help Manager compares a string specified in the component against the current menu item. If the string matches the current menu item, the Help Manager uses the help messages specified in the rest of the component, shown in Figure 3-28. This type of component is useful for a menu item that can change names.

Figure 3-28 Structure of a menu-item component compiled with the `HMCompareItem` identifier



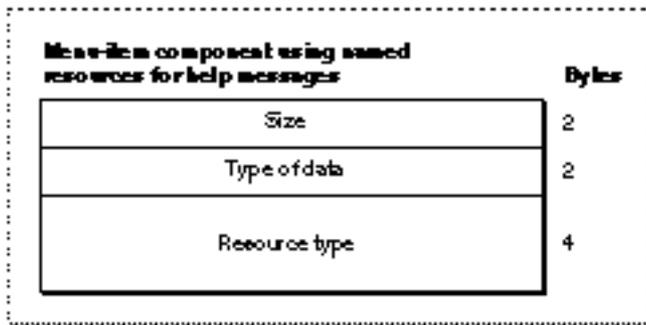
If you examine a compiled version of an 'hmenu' resource, you find that a component identified in a Rez input file by the `HMCompareItem` identifier consists of these elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 512 appears here when the Help Manager is to use the help messages specified in this component only when the current menu item matches a specified text string.
- n Text string. The string against which to compare the current menu item. If the current menu item matches this string, then the Help Manager uses the help messages specified in this component.
- n Alignment bytes. Zero or one bytes used to make the previous text string end on a word boundary.
- n Help messages. The four help messages for the menu item. The structure may follow that of any of the previously described menu-item components; that is, this element consists of a value representing the format of the help messages specified in the rest of the component, the size of the rest of the component, and specifications for four actual help messages for the menu item.

When the identifier `HMNamedResourceItem` is specified, the Help Manager retrieves help messages from a resource that matches the name and state of the current menu item.

Figure 3-29 shows the format of a menu-item component that uses named resources for help messages.

Figure 3-29 Structure of a menu-item component compiled with the `HMNamedResourceItem` identifier



If you examine a compiled version of an 'hmenu' resource, you find that a component identified in a Rez input file by the `HMNamedResourceItem` identifier consists of these elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The number 1024 is specified here when the Help Manager is to use named resources for help messages.
- n Resource type. The resource type ('STR', 'STR#', 'PICT', or, for text, 'TEXT') of the resource that contains the help messages for the current menu item. The Help Manager then uses the `GetNamedResource` function to find the resource with the same name as the current menu item. (If 'TEXT' is specified, the Help Manager also uses the style information contained in an 'styl' resource with the same name.) If the menu item is dimmed, the Help Manager appends an exclamation point (!) to the menu item string and searches for a resource by that name. If the menu item is enabled and marked with a checkmark or other mark, the Help Manager appends the mark to the menu item string and looks for a resource with that name.

The Dialog-Item Help Resource

You can provide help balloons for individual items in a dialog box or an alert box by supplying a dialog-item help resource, which is a resource of type 'hdlg'. You specify different help balloons for various states of an item—by highlight value if the item is a control, and by enabled or disabled states for items that are not controls.

To associate an 'hdlg' resource with a particular alert box or dialog box, either you must include an item of type `HelpItem` in the box's item list ('DITL') resource, or you must create an 'hwin' resource. Listing 3-8 on page 3-59 shows how to use an item of type `HelpItem`—and Listing 3-10 on page 3-72 shows you how to use an 'hwin'

Help Manager

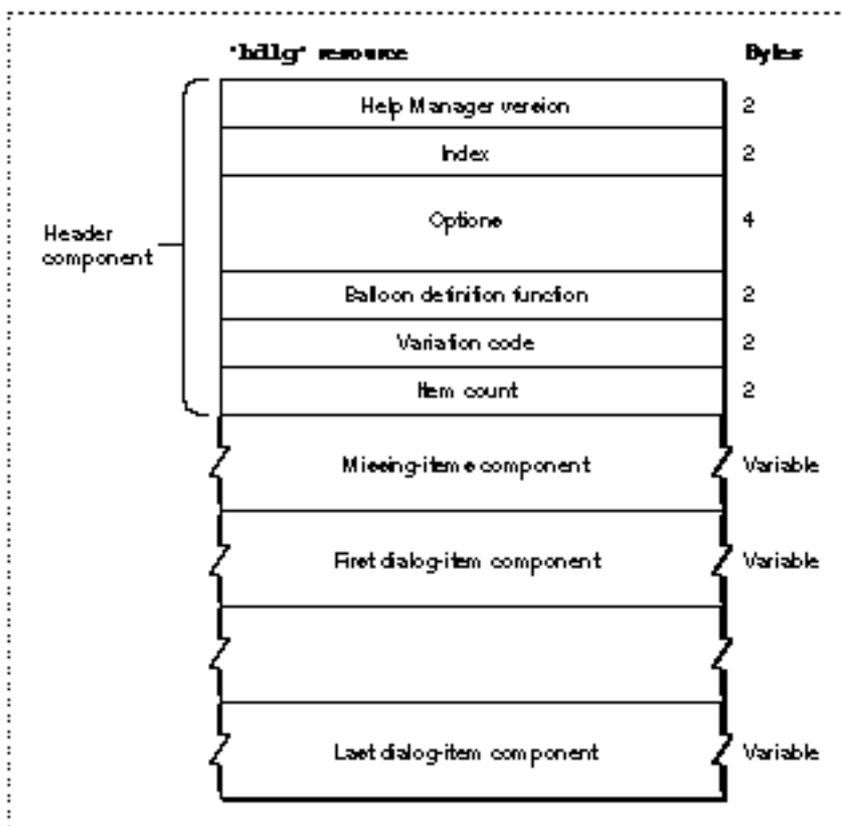
resource—for associating an 'hdlg' resource with a particular alert box or dialog box. For detailed information about using an item of type `HelpItem`, see “Using a Help Item Versus Using an 'hwin' Resource” on page 3-63. For detailed information on using an 'hwin' resource, see “Associating Help Resources With Static Windows” on page 3-68.

All 'hdlg' resources must have resource IDs greater than 128.

The format of a Rez input file for an 'hdlg' resource differs from its compiled output form. This section describes the structure of a Rez-compiled 'hdlg' resource. If you are concerned only with creating 'hdlg' resources, see “Providing Help Balloons for Items in Dialog Boxes and Alert Boxes” on page 3-51 for a detailed description, using several code samples, of how to use Rez input files to create 'hdlg' resources.

An 'hdlg' resource consists of a header component, a missing-items component, and a variable number of dialog-item components. Figure 3-30 shows the general structure of a compiled 'hdlg' resource.

Figure 3-30 Structure of a compiled dialog-item help ('hdlg') resource



Help Manager

If you examine a compiled version of an 'hdlg' resource, you find that the header component consists of the following elements:

- n Help Manager version. The version of the Help Manager to use. This is usually specified in a Rez input file with the `HelpMgrVersion` constant.
- n Index. An index (starting with 0) into an item list ('DITL') resource. The Help Manager adds the value of this index to the number of the first item in the item list resource and then associates the result with an item number within the item list resource; therefore, index 0 corresponds to item 1 in the item list resource (because 0 plus 1 equals 1). The Help Manager then uses the first dialog-item component in the 'hdlg' resource to provide help for the item to which this index corresponds. Subsequent dialog-item components specify help messages for subsequent items in the item list resource. For example, when 4 is specified as the index, the first dialog-item component specifies help messages for the fifth item in an item list resource. (As explained earlier, either an item of type `helpItem` in the item list resource or an 'hwin' resource is used to associate the messages in the dialog-item components of this 'hdlg' resource with the items of a particular dialog box or alert box.)
- n Options. The sum of the values of available options, described in “Specifying Options in Help Resources” beginning on page 3-25.
- n Balloon definition function. The resource ID of the window definition function used for drawing the help balloon. The standard balloon definition function is of type 'WDEF' with resource ID 126; this can be specified by the number 0 in the Rez input file.
- n Variation code. A number signifying the preferred position of the help balloon relative to the hot rectangle. The balloon definition function draws the frame of the help balloon based on the variation code specified here. The eight variation codes and how they affect the standard balloon definition function are illustrated in Figure 3-4 on page 3-10.
- n Item count. The number of remaining components—that is, the missing-items component plus all dialog-item components—defined in the rest of this resource.

The missing-items component always follows the header component of an 'hdlg' resource. Then a variable number of dialog-item components are stored in this resource. The Help Manager determines the end of the 'hdlg' resource by using the item count information in the header component. The Help Manager determines the type of each component by its order in the resource.

The structures of the missing-items component and the dialog-item components depend on identifiers specified inside the components. The identifiers used in a Rez input file are described in “Specifying the Format for Help Messages” on page 3-23.

Help Manager

The missing-items component and the dialog-item components can each specify four different help messages:

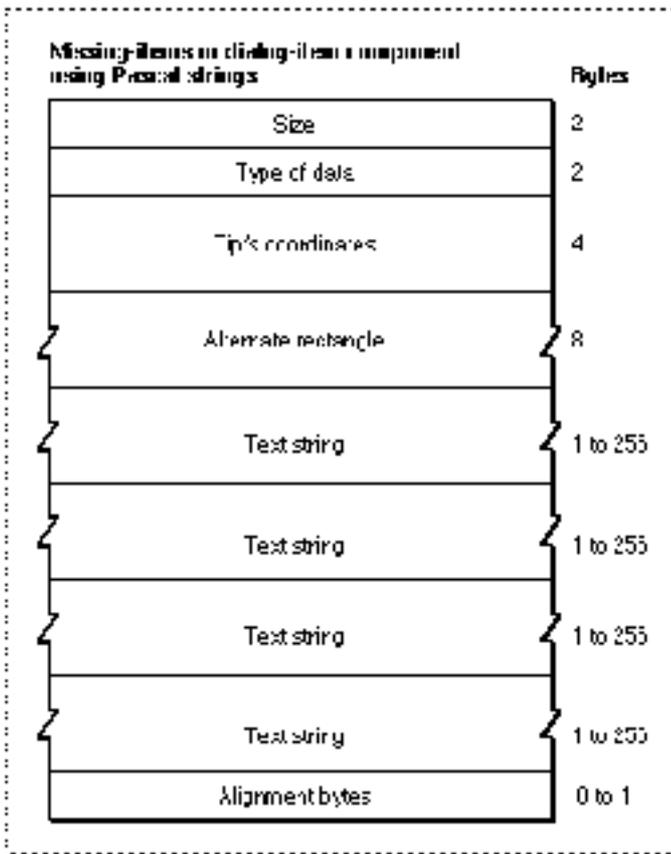
- n First help message.
 - n In the missing-items component, this is the help message both for missing, active, unselected controls (that is, those with highlight values of 0) and for missing enabled items that are not controls.
 - n In dialog-item components, this is the help message for an active, unselected control (that is, one with a highlight value of 0) or for an enabled item that is not a control.
- n Second help message.
 - n In the missing-items component, this is the help message both for missing dimmed controls (that is, those with highlight values of 255) and for missing disabled items that are not controls.
 - n In dialog-item components, this is the help message for a dimmed control (that is, one with a highlight value of 255) or for a disabled item that is not a control.
- n Third help message.
 - n In the missing-items component, this is the help message for missing active controls that are checked (that is, those with highlight values of 1).
 - n In dialog-item components, this is the help message for an active control that is checked (that is, one with a highlight value of 1).
- n Fourth help message.
 - n In the missing-items component, this is the help message for missing, selected controls with highlight values between 2 and 253.
 - n In dialog-item components, this is the help message for a selected control with any highlight value between 2 and 253.

An empty string or a resource ID of 0 for a message in any dialog-item component causes the Help Manager to use the appropriate help message contained in the missing-items component.

Since they both adhere to the formats specified by the previously described identifiers, the missing-items component and the dialog-item components can have similar structures. The Help Manager determines the end of a component by examining its length, which is stored in the first 2 bytes of the component.

Figure 3-31 shows the structure of a component that stores its help messages as Pascal strings within the 'hdlg' resource itself.

Figure 3-31 Structure of an 'hdlg' component compiled with the `HMStringItem` identifier



If you examine a compiled version of an 'hdlg' resource, you find that a component identified in a Rez input file by the `HMStringItem` identifier consists of the following elements:

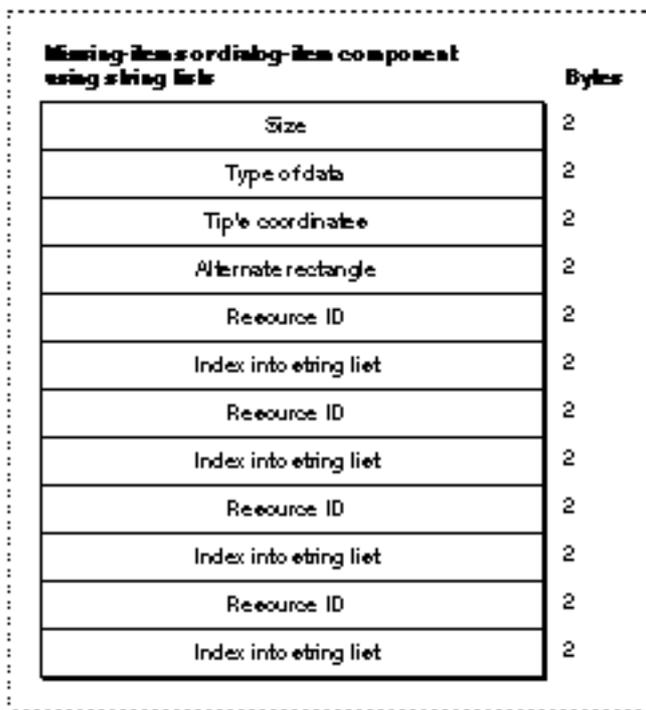
- n Size. The number of bytes contained in this component.
- n Type of data. The value 1 is specified here when the help messages are stored as Pascal strings within this component.
- n Tip's coordinates. The coordinates of the help balloon's tip. The tip's coordinates are local to the item's display rectangle.
- n Alternate rectangle. The coordinates for a rectangle used by the Help Manager for transposing the tip if a help balloon does not fit onscreen. These coordinates are local to the item's display rectangle.

Help Manager

- n Text string. The first help message (as previously described).
- n Text string. The second help message (as previously described).
- n Text string. The third help message (as previously described).
- n Text string. The fourth help message (as previously described).
- n Alignment bytes. Zero or one bytes used to make the previous text strings end on a word boundary.

Figure 3-32 shows the structure of an 'hdlg' component that specifies its help messages as text strings stored in string list ('STR#') resources.

Figure 3-32 Structure of an 'hdlg' component compiled with the `HMStringResItem` identifier



If you examine a compiled version of an 'hdlg' resource, you find that a component identified in a Rez input file by the `HMStringResItem` identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 3 is specified here when the help messages for this component are stored in string list ('STR#') resources.
- n Tip's coordinates. The coordinates of the help balloon's tip. The tip's coordinates are local to the item's display rectangle.

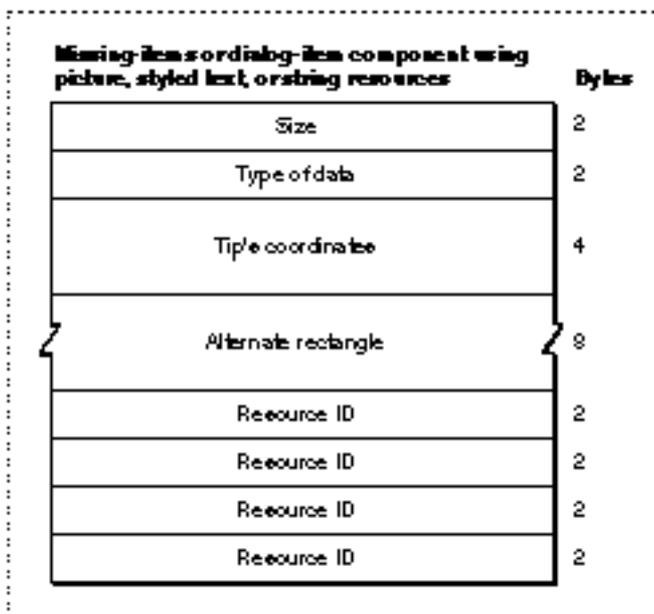
Help Manager

- n Alternate rectangle. The coordinates for a rectangle used by the Help Manager for transposing the tip if a help balloon does not fit onscreen. These coordinates are local to the item's display rectangle.
- n Resource ID. The resource ID of an 'STR#' resource.
- n Index into the string list resource. A number used as an index to a particular text string within the 'STR#' resource. This text string is used for the first help message (as previously described).

Three more pairs of resource IDs and their index numbers follow. The text strings referenced by these pairs are used for the second, third, and fourth help messages, respectively.

Figure 3-33 shows the structure of an 'hdlg' component that specifies its help messages in picture ('PICT') resources, styled text ('TEXT' and 'styl') resources, or string ('STR ') resources.

Figure 3-33 Structure of an 'hdlg' component compiled with the HMPictItem, HMTResItem, or HMSTRResItem identifier



Help Manager

If you examine a compiled version of an 'hdlg' resource, you find that a component identified in a Rez input file by either the `HMPictItem`, `HMTEResItem`, or `HMSTRResItem` identifier consists of the following elements:

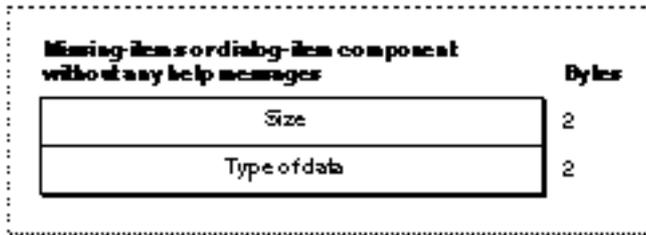
- n **Size.** The number of bytes contained in this component.
- n **Type of data.**
 - n The value 2 is specified here when the help messages for this component are stored in 'PICT' resources.
 - n The value 6 is specified here when the help messages for this component are stored as styled text—that is, in both 'TEXT' and 'styl' resources.
 - n The value 7 is specified here when the help messages for this component are stored in 'STR ' resources.
- n **Tip's coordinates.** The coordinates of the help balloon's tip. The tip's coordinates are local to the item's display rectangle.
- n **Alternate rectangle.** The coordinates for a rectangle used by the Help Manager for transposing the tip if a help balloon does not fit onscreen. These coordinates are local to the item's display rectangle.
- n **Resource ID.**
 - n The resource ID of a 'PICT' resource when the value 2 is specified as the type of data. The Help Manager uses the picture contained in this resource for the first help message (as previously described).
 - n The resource ID common to both a 'TEXT' and an 'styl' resource when the value 6 is specified as the type of data. The Help Manager uses the styled text specified by these resources for the first help message.
 - n The resource ID of an 'STR ' resource when the value 7 is specified as the type of data. The Help Manager uses the text contained in this resource for the first help message.

Three more resource IDs follow; the Help Manager uses these resources (either 'PICT', 'TEXT' and 'styl', or 'STR ') for the second, third, and fourth help messages, respectively (as previously described).

Help Manager

Figure 3-34 shows the structure of an 'hdlg' component that specifies no help messages.

Figure 3-34 Structure of an 'hdlg' component compiled with the HMSkipItem identifier



If you examine a compiled version of an 'hdlg' resource, you find that a component identified by the HMSkipItem identifier consists of the following elements:

- n Size. The value 4, for the number of bytes contained in this component.
- n Type of data. The value 256.

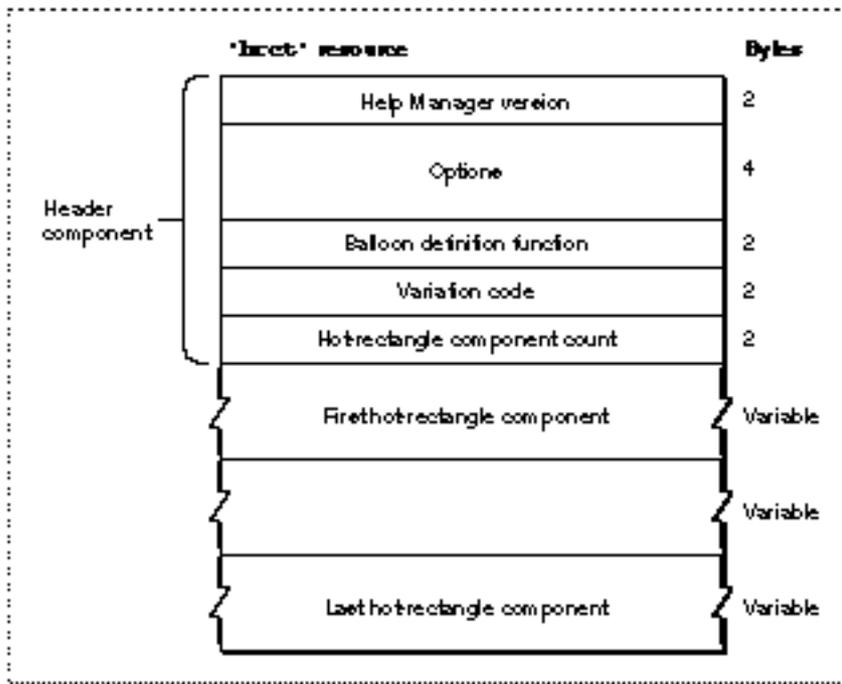
The Rectangle Help Resource

You can use a rectangle help resource to define hot rectangles for displaying help balloons within a static window, and to specify the help messages for those balloons. A rectangle help resource is a resource of type 'hrcr'. All 'hrcr' resources must have resource IDs greater than 128.

To associate the hot rectangles and help messages defined in an 'hrcr' resource with a particular window, you must also create a window help ('hwin') resource, which is described in “Associating Help Resources With Static Windows” on page 3-68.

The format of a Rez input file for an 'hrcr' resource differs from its compiled output form. This section describes the structure of a Rez-compiled 'hrcr' resource. If you are concerned only with creating 'hrcr' resources, see “Specifying Help for Rectangles in Windows” on page 3-67 for a detailed description of how to use Rez input files to create 'hrcr' resources.

An 'hrcr' resource consists of a header component and a variable number of hot-rectangle components. Figure 3-35 shows the general structure of a compiled 'hrcr' resource.

Figure 3-35 Structure of a compiled rectangle help ('hrcr') resource

If you examine a compiled version of an 'hrcr' resource, you find that the header component consists of the following elements:

- n Help Manager version. The version of the Help Manager to use. This is usually specified in a Rez input file with the `HelpMgrVersion` constant.
- n Options. The sum of the values of available options, described in “Specifying Options in Help Resources” beginning on page 3-25.
- n Balloon definition function. The resource ID of the window definition function used for drawing the help balloon. The standard balloon definition function is of type 'WDEF' with resource ID 126; this can be specified by 0 in the Rez input file.
- n Variation code. A number signifying the preferred position of the help balloon relative to the hot rectangle. The balloon definition function draws the frame of the help balloon based on the variation code specified here. The eight variation codes and how they affect the standard balloon definition function are illustrated in Figure 3-4 on page 3-10.
- n Hot-rectangle component count. The number of hot-rectangle components defined in the rest of this resource.

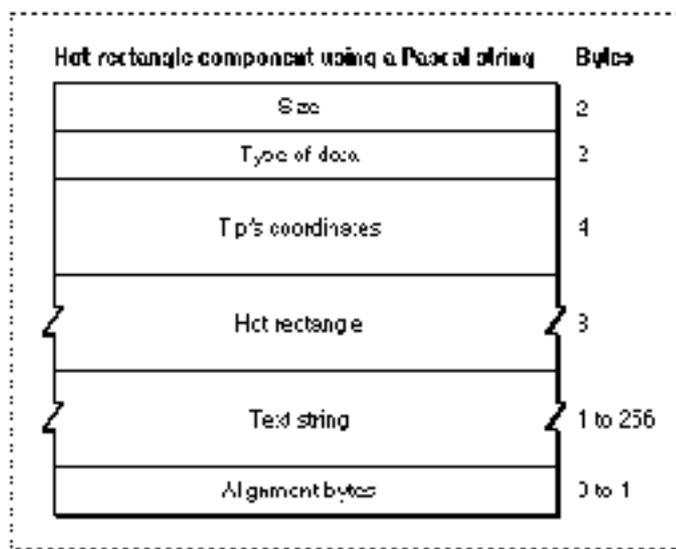
Help Manager

The Help Manager determines the end of the 'hrcr' resource by using the component count information in the header component.

The structures of the hot-rectangle components depend on identifiers specified inside the components. The identifiers used in a Rez input file are described in "Specifying the Format for Help Messages" on page 3-23.

Figure 3-36 shows the structure of a component that stores its help message as a Pascal string within the 'hrcr' resource itself.

Figure 3-36 Structure of an 'hrcr' component compiled with the `HMStringItem` identifier



If you examine a compiled version of an 'hrcr' resource, you find that a component identified in a Rez input file by the `HMStringItem` identifier consists of the following elements:

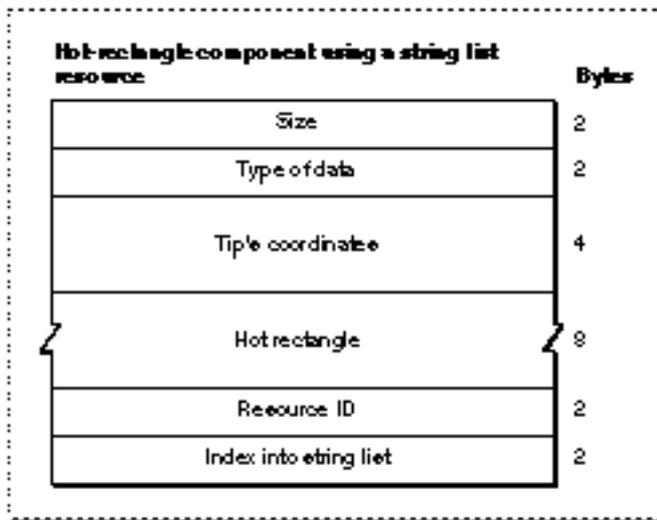
- n Size. The number of bytes contained in this component.
- n Type of data. The value 1 is specified here when the help message is stored as a Pascal string within this component.
- n Tip's coordinates. The coordinates of the help balloon's tip. The tip's coordinates are local to the window.
- n Hot rectangle. The coordinates (local to the window) of a rectangle. The Help Manager displays a help message when the user moves the cursor over this rectangle.

Help Manager

- n Text string. The help message that the Help Manager displays when the user moves the cursor over the hot rectangle.
- n Alignment bytes. Zero or one bytes used to make the previous text strings end on a word boundary.

Figure 3-37 shows the structure of a hot-rectangle component that specifies its help message as a text string stored in a string list ('STR#') resource.

Figure 3-37 Structure of an 'hrcr' component compiled with the `HMStringResItem` identifier

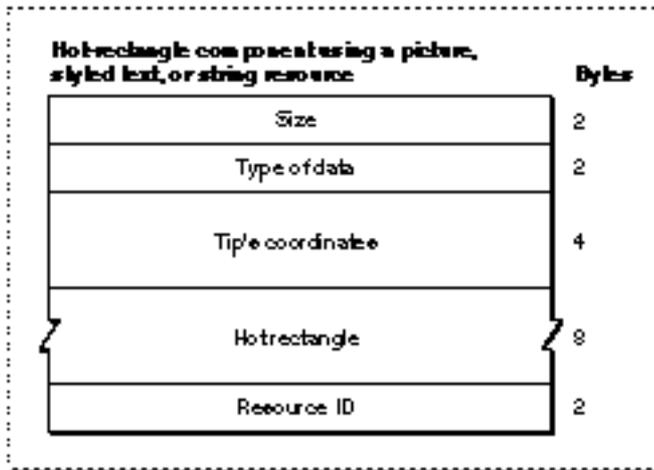


If you examine a compiled version of an 'hrcr' resource, you find that a component identified in a Rez input file by the `HMStringResItem` identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 3 is specified here when the help message for this component is stored in an 'STR#' resource.
- n Tip's coordinates. The coordinates of the help balloon's tip. The tip's coordinates are local to the window.
- n Hot rectangle. The coordinates (local to the window) of a rectangle. The Help Manager displays a help message when the user moves the cursor over this rectangle.
- n Resource ID. The resource ID of an 'STR#' resource.
- n Index into the string list resource. A number used as an index to a particular text string within the 'STR#' resource. When the user moves the cursor over the hot rectangle, the Help Manager displays this text string for the help message.

Figure 3-38 shows the structure of a hot-rectangle component that specifies its help message in a picture ('PICT') resource, in styled text ('TEXT' and 'styl') resources, or in a string ('STR') resource.

Figure 3-38 Structure of an 'hrc' component compiled with the HMPictItem, HMTEResItem, or HMSTRResItem identifier



If you examine a compiled version of an 'hrc' resource, you find that a component identified in a Rez input file by either the HMPictItem, HMTEResItem, or HMSTRResItem identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data.
 - n The value 2 is specified here when the help message for this component is stored in a 'PICT' resource.
 - n The value 6 is specified here when the help message for this component is stored as styled text—that is, in both 'TEXT' and 'styl' resources.
 - n The value 7 is specified here when the help message for this component is stored in an 'STR' resource.
- n Tip's coordinates. The coordinates of the help balloon's tip. The tip's coordinates are local to the window.
- n Hot rectangle. The coordinates (local to the window) of a rectangle. The Help Manager displays a help message when the user moves the cursor over this rectangle.

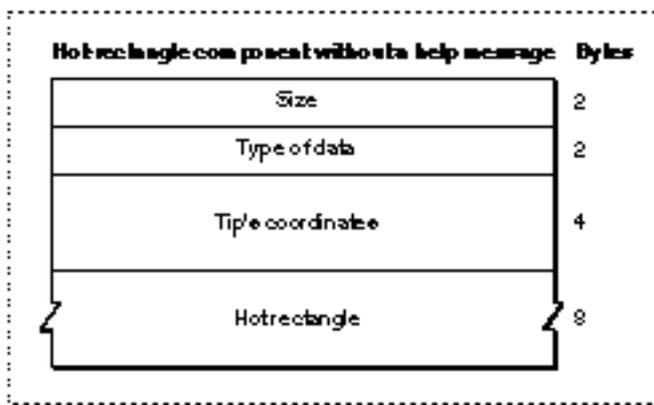
Help Manager

n Resource ID.

- n The resource ID of a 'PICT' resource when the value 2 is specified as the type of data. When the user moves the cursor over the hot rectangle, the Help Manager displays the picture stored in this resource for the help message.
- n The resource ID common to both a 'TEXT' and an 'styl' resource when the value 6 is specified as the type of data. When the user moves the cursor over the hot rectangle, the Help Manager displays the styled text specified in these resources for the help message.
- n The resource ID of an 'STR' resource when the value 7 is specified as the type of data. When the user moves the cursor over the hot rectangle, the Help Manager uses the text string stored in this resource for the help message.

Figure 3-39 shows the structure of a hot-rectangle component that doesn't specify a help message.

Figure 3-39 Structure of an 'hrcr' component compiled with the HMSkipItem identifier



If you examine a compiled version of an 'hrcr' resource, you find that a component identified by the HMSkipItem identifier consists of the following elements:

- n Size. The value 4, for the number of bytes contained in this component.
- n Type of data. The value 256.
- n Tip's coordinates. In this instance, the Help Manager does not use this information because it does not display a help balloon.
- n Hot rectangle. The coordinates (local to the window) of a rectangle that is to be skipped. When the user moves the cursor over this rectangle, the Help Manager does not display any help messages.

The Window Help Resource

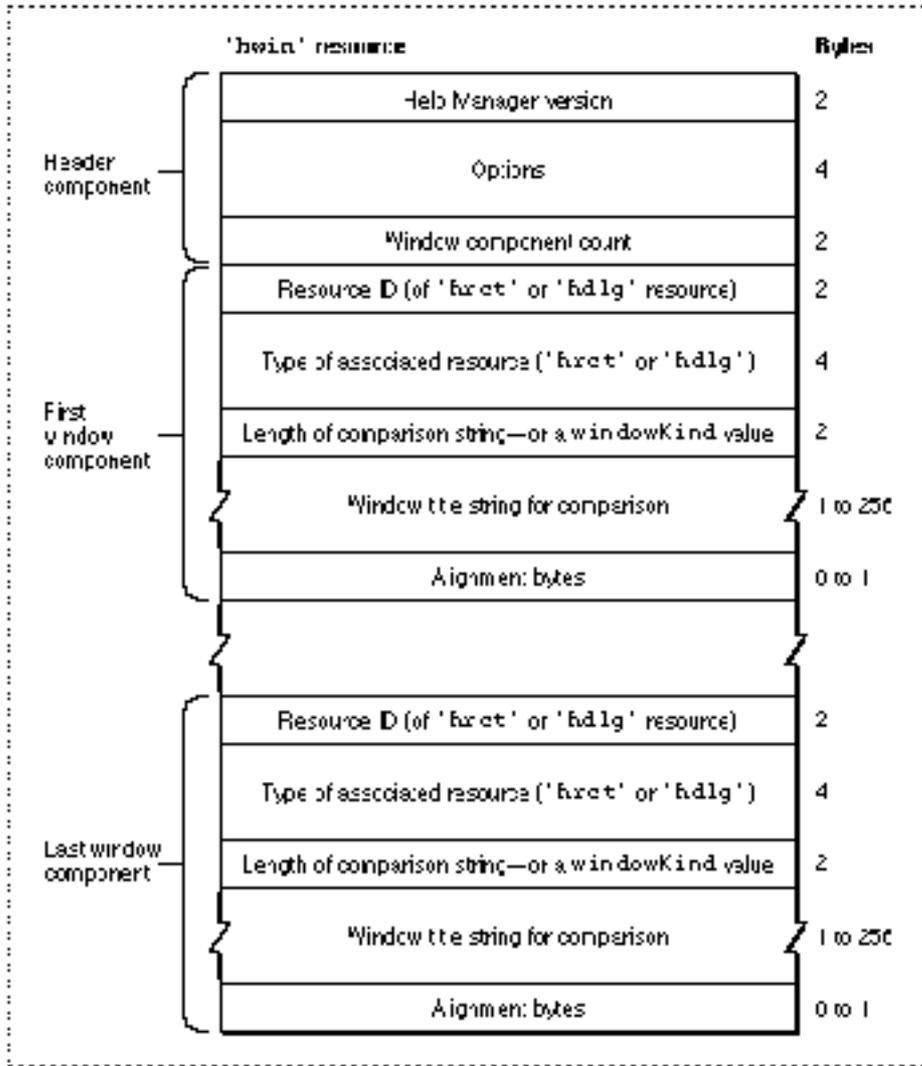
To associate the help balloons defined in an 'hrct' resource with a particular window, you must create a window help resource. Unless you include an item of type `HelpItem` in an item list resource, you also must create a window help resource to associate an 'hdlg' resource with a particular alert box or dialog box. The window help resource is a resource of type 'hwin'. All 'hwin' resources must have resource IDs greater than 128.

The 'hwin' resource merely associates 'hrct' and 'hdlg' resources with windows. To specify hot rectangles, help balloon characteristics, and help messages for areas in a static window, you must use 'hrct' or 'hdlg' resources, which are described in “Specifying Help for Rectangles in Windows” on page 3-67 and “Providing Help Balloons for Items in Dialog Boxes and Alert Boxes” on page 3-51, respectively.

The format of a Rez input file for an 'hwin' resource differs from its compiled output form. This section describes the structure of a Rez-compiled 'hwin' resource. If you are concerned only with creating 'hwin' resources, see “Associating Help Resources With Static Windows” on page 3-68 for a detailed description of how to use Rez input files to create 'hwin' resources.

An 'hwin' resource consists of a header component and a variable number of window components. Figure 3-40 shows the general structure of a compiled 'hwin' resource.

Figure 3-40 Structure of a compiled window help ('hwin') resource



Help Manager

If you examine a compiled version of an 'hwin' resource, you find that the header component consists of the following elements:

- n Help Manager version. The version of the Help Manager to use. This is usually specified in a Rez input file with the `HelpMgrVersion` constant.
- n Options. The sum of the values of available options, described in “Specifying Options in Help Resources” beginning on page 3-25.
- n Window component count. The number of window components defined in the rest of this resource. The Help Manager determines the end of the 'hwin' resource by using this component count information.

If you examine a compiled version of an 'hwin' resource, you find that a window component consists of the following elements:

- n Resource ID. The ID of the associated resource (either 'hrcr' or 'hdlg') that specifies the help messages for the window.
- n Type of associated resource. A resource type; either 'hrcr' or 'hdlg'.
- n Length of comparison string—or a `windowKind` value. If the integer in this element is positive, this is the number of characters used for matching this component to a window's title. If the integer in this element is negative, this is a value used for matching this component to a window by the `windowKind` value in the window's window record.
- n Window title string. If the previous element is a positive integer, this element consists of characters that the Help Manager uses to match this component to a window by the window's title. If the previous element is a negative integer, this is an empty string.
- n Alignment bytes. Zero or one bytes used to make the window title string end on a word boundary.

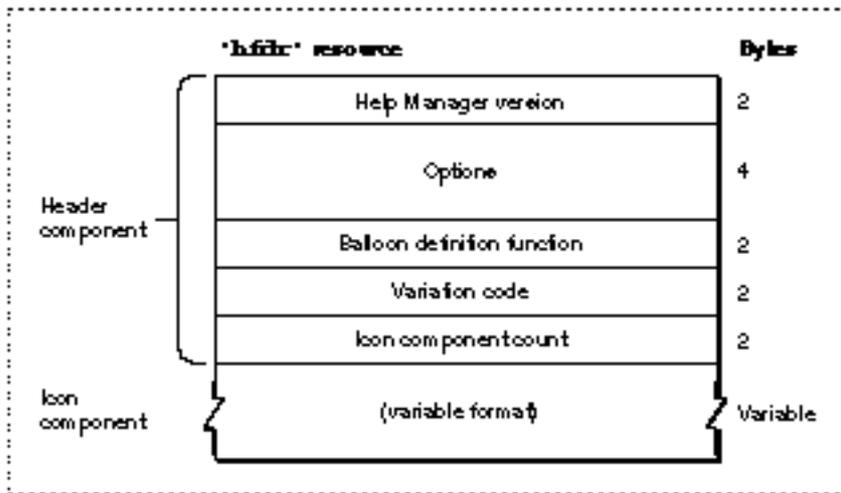
The Finder Icon Help Resource

The Help Manager displays default help messages for all Finder icon types. By creating a Finder icon help override resource, you can provide your own help message for the Help Manager to display when the user moves the cursor over your non-document icons. A Finder icon help resource is a resource of type 'hfdr'. An 'hfdr' resource must have a resource ID of -5696.

The format of a Rez input file for an 'hfdr' resource differs from its compiled output form. This section describes the structure of a Rez-compiled 'hfdr' resource. If you are concerned only with creating 'hfdr' resources, see “Overriding Help Balloons for Non-Document Icons” on page 3-84 for a detailed description of how to use Rez input files to create an 'hfdr' resource.

An 'hfdr' resource consists of a header component and one icon component. Figure 3-41 shows the general structure of a compiled 'hfdr' resource.

Figure 3-41 Structure of a compiled Finder icon help ('hfdr') resource



If you examine a compiled version of an 'hfdr' resource, you find that the header component consists of the following elements:

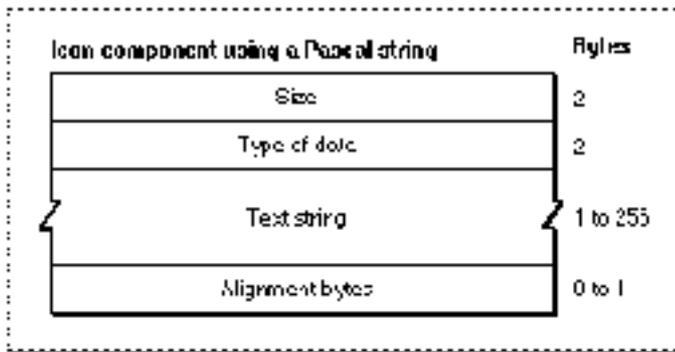
- n Help Manager version. The version of the Help Manager to use. This is usually specified in a Rez input file with the `HelpMgrVersion` constant.
- n Options. The sum of the values of available options, described in “Specifying Options in Help Resources” beginning on page 3-25.
- n Balloon definition function. The resource ID of the window definition function used for drawing the help balloon. The standard balloon definition function is of type 'WDEF' with resource ID 126; this can be specified by the number 0 in the Rez input file.
- n Variation code. A number signifying the preferred position of the help balloon relative to the hot rectangle. The balloon definition function draws the frame of the help balloon based on the variation code specified here. The eight variation codes and how they affect the standard balloon definition function are illustrated in Figure 3-4 on page 3-10.
- n Icon component count. The value 1, because only one icon component can be defined in this resource.

The structure of the icon component depends on the identifier specified for that component. The identifiers used in a Rez input file are described in “Specifying the Format for Help Messages” on page 3-23.

Help Manager

Figure 3-42 shows the structure of an icon component that stores its help message as a Pascal string within the 'hfd:r' resource itself.

Figure 3-42 Structure of an 'hfd:r' component compiled with the HMStringItem identifier

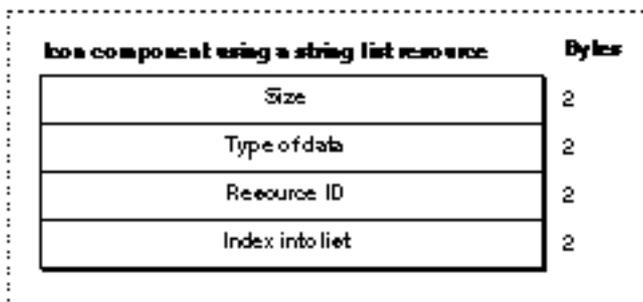


If you examine a compiled version of an 'hfd:r' resource, you find that a component identified in a Rez input file by the HMStringItem identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 1 is specified here when the help messages are stored as a Pascal string within this component.
- n Text string. The help message that the Help Manager displays when the user moves the cursor over the icon.
- n Alignment bytes. Zero or one bytes used to make the previous text strings end on a word boundary.

Figure 3-43 shows the structure of an icon component that specifies its help message as a text string stored in a string list ('STR#') resource.

Figure 3-43 Structure of an 'hfd:r' component compiled with the HMStringResItem identifier



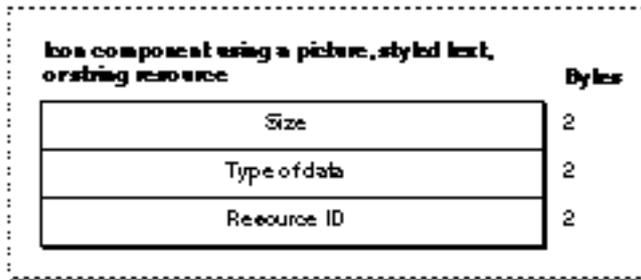
Help Manager

If you examine a compiled version of an 'hldr' resource, you find that a component identified in a Rez input file by the `HMStringResItem` identifier consists of the following elements:

- n **Size.** The number of bytes contained in this component.
- n **Type of data.** The value 3 is specified here when the help messages for this component are stored in string list ('STR#') resources.
- n **Resource ID.** The resource ID of an 'STR#' resource.
- n **Index into the string list resource.** A number used as an index to a particular text string within the 'STR#' resource. The Help Manager displays this text string for the help message.

Figure 3-44 shows the structure of an icon component that specifies its help message in a picture ('PICT') resource, in styled text ('TEXT' and 'styl') resources, or in a string ('STR ') resource.

Figure 3-44 Structure of an 'hldr' component compiled with the `HMPictItem`, `HMTEResItem`, or `HMSTRResItem` identifier



If you examine a compiled version of an 'hldr' resource, you find that a component identified in a Rez input file by either the `HMPictItem`, `HMTEResItem`, or `HMSTRResItem` identifier consists of the following elements:

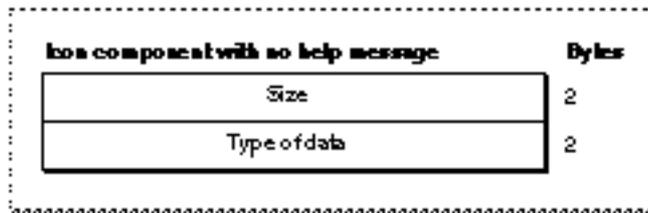
- n **Size.** The number of bytes contained in this component.
- n **Type of data.**
 - n The value 2 is specified here when the help message for this component is stored in a 'PICT' resource.
 - n The value 6 is specified here when the help message for this component is stored as styled text—that is, in both 'TEXT' and 'styl' resources.
 - n The value 7 is specified here when the help message for this component is stored in an 'STR ' resource.

Help Manager

- n Resource ID.
 - n The resource ID of a 'PICT' resource when the value 2 is specified as the type of data. The Help Manager displays the picture stored in this resource for the help message.
 - n The resource ID common to both a 'TEXT' and an 'styl' resource when the value 6 is specified as the type of data. The Help Manager displays the styled text specified in these resources for the help message.
 - n The resource ID of an 'STR' resource when the value 7 is specified as the type of data. The Help Manager uses the text string stored in this resource for the help message.

Figure 3-45 shows the structure of an icon component that doesn't specify a help message.

Figure 3-45 Structure of an 'hldr' component compiled with the HMSkipItem identifier



If you examine a compiled version of an 'hldr' resource, you find that a component identified by the HMSkipItem identifier consists of the following elements:

- n Size. The value 4, for the number of bytes contained in this component.
- n Type of data. The value 256.

The Default Help Override Resource

The Help Manager also provides default help balloons for the title bar and the close and zoom boxes of an active window, for the windows of inactive applications, for inactive windows of an active application, and for the area outside a modal dialog box.

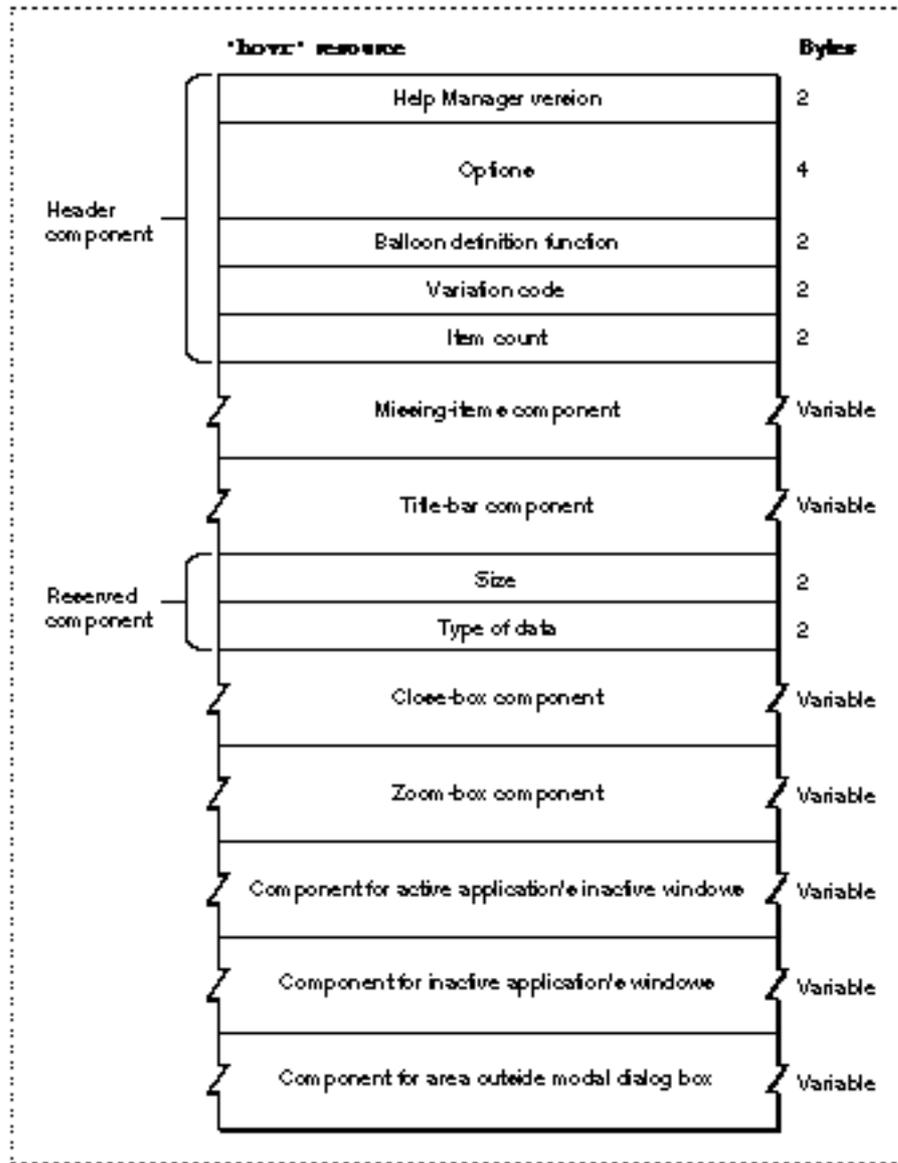
Apple has researched and tested these help messages to ensure that they are as effective as possible for users. Normally, you don't need to override them. However, by creating a default help override resource you can override one or more of these defaults if absolutely necessary. A default help override resource is a resource of type 'hovr'. The 'hovr' resource must have a resource ID greater than 128.

The format of a Rez input file for an 'hovr' resource differs from its compiled output form. This section describes the structure of a Rez-compiled 'hovr' resource. If you are concerned only with creating 'hovr' resources, see "Overriding Other Default Help Balloons" on page 3-87 for a detailed description of how to use Rez input files to create 'hovr' resources.

Help Manager

An 'hovr' resource consists of a header component, a missing-items component, and seven additional components for various interface elements. Figure 3-46 shows the general structure of a compiled 'hovr' resource.

Figure 3-46 Structure of a compiled default help override ('hovr') resource



Help Manager

If you examine a compiled version of an 'hovr' resource, you find that the header component consists of the following elements:

- n Help Manager version. The version of the Help Manager to use. This is usually specified in a Rez input file with the `HelpMgrVersion` constant.
- n Options. The sum of the values of available options, described in “Specifying Options in Help Resources” beginning on page 3-25.
- n Balloon definition function. The resource ID of the window definition function used for drawing the help balloon. The standard balloon definition function is of type 'WDEF' with resource ID 126; this can be specified by 0 in the Rez input file.
- n Variation code. A number signifying the preferred position of the help balloon relative to the hot rectangle. The balloon definition function draws the frame of the help balloon based on the variation code specified here. The eight variation codes and how they affect the standard balloon definition function are illustrated in Figure 3-4 on page 3-10.
- n Item count. The value 8 for the number of components defined in the rest of this resource.

The Help Manager uses the order of the components in this resource to determine their purposes.

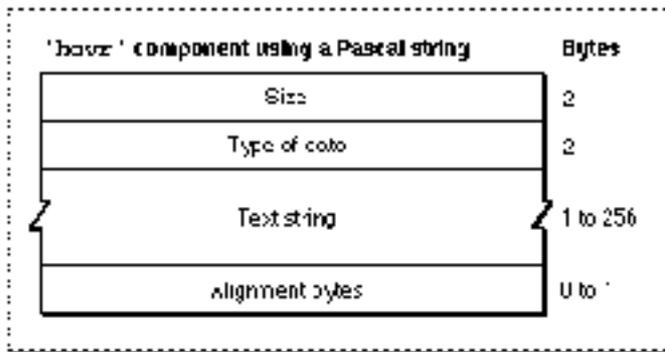
The structures of the remaining components depend on identifiers specified inside the components. The identifiers used in a Rez input file are described in “Specifying the Format for Help Messages” on page 3-23.

Each component can specify one help message, as listed here.

- n Missing-items component. The Help Manager expects seven more components to follow, in the order listed here. If fewer than seven components are specified in the Rez input file, the Help Manager adds components to the end of the list until there are seven. Each component that the Help Manager adds uses the message specified in the missing-items component. The Help Manager also uses the missing-items component's help message if the input file specifies an empty string or a resource ID of 0 for any other component's help message.
- n Title-bar component. The help message for title bar of the active window.
- n Reserved component. This element is reserved and should have no help message. The `HMSkipItem` identifier should always be specified in the Rez input file for this component.
- n Close-box component. The help message for the close box of the active window.
- n Zoom-box component. The help message for the zoom box of the active window.
- n Component for active application's inactive windows. The help message for the inactive windows of the active application.
- n Component for inactive applications' windows. The help message for the windows of inactive applications.
- n Component for area outside modal box. The help message for the desktop area outside a modal dialog box or an alert box.

Figure 3-47 shows the structure of an 'hovr' component that stores its help message as a Pascal string within the 'hovr' resource itself.

Figure 3-47 Structure of an 'hovr' component compiled with the HMStringItem identifier

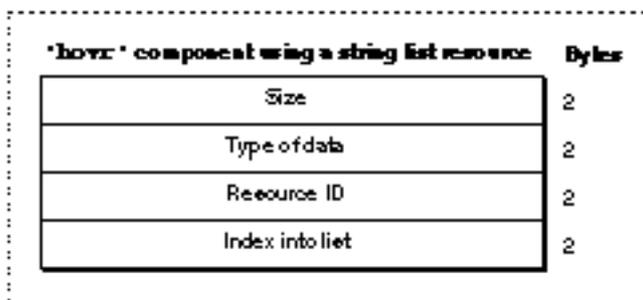


If you examine a compiled version of an 'hovr' resource, you find that a component identified in a Rez input file by the HMStringItem identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 1 is specified here when the help message is stored as a Pascal string within this component.
- n Text string. The help message appropriate for the component (as previously described).
- n Alignment bytes. Zero or one bytes used to make the text string end on a word boundary.

Figure 3-48 shows the structure of an 'hovr' component that specifies its help message as a text string stored in a string list ('STR#') resource.

Figure 3-48 Structure of an 'hovr' component compiled with the HMStringResItem identifier



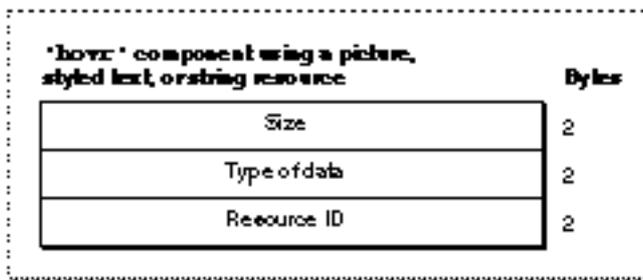
Help Manager

If you examine a compiled version of an 'hovr' resource, you find that a component identified in a Rez input file by the `HMStringResItem` identifier consists of the following elements:

- n Size. The number of bytes contained in this component.
- n Type of data. The value 3 is specified here when the help message for this component is stored in a string list ('STR#') resource.
- n Resource ID. The resource ID of an 'STR#' resource.
- n Index into the string list resource. A number used as an index to a particular text string within the 'STR#' resource. The Help Manager uses this text string for the help message of the appropriate component (as previously described).

Figure 3-49 shows the structure of an 'hovr' component that specifies its help message in a picture ('PICT') resource, in styled text ('TEXT' and 'styl') resources, or in a string ('STR ') resource.

Figure 3-49 Structure of an 'hovr' component compiled with the `HMPictItem`, `HMTEResItem`, or `HMSTRResItem` identifier



If you examine a compiled version of an 'hovr' resource, you find that a component identified in a Rez input file by either the `HMPictItem`, `HMTEResItem`, or `HMSTRResItem` identifier consists of the following elements:

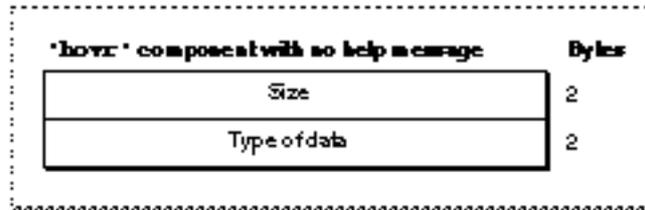
- n Size. The number of bytes contained in this component.
- n Type of data.
 - n The value 2 is specified here when the help message for this component is stored in a 'PICT' resource.
 - n The value 6 is specified here when the help message for this component is stored as styled text—that is, in both 'TEXT' and 'styl' resources.
 - n The value 7 is specified here when the help message for this component is stored in an 'STR ' resource.

Help Manager

- n Resource ID.
 - n The resource ID of a 'PICT' resource when the value 2 is specified as the type of data. The Help Manager displays the picture stored in this resource for the help message.
 - n The resource ID common to both a 'TEXT' and an 'styl' resource when the value 6 is specified as the type of data. The Help Manager displays the styled text specified in these resources for the help message.
 - n The resource ID of an 'STR' resource when the value 7 is specified as the type of data. The Help Manager uses the text string stored in this resource for the help message.

Figure 3-50 shows the structure of an 'hovr' component that doesn't specify a help message.

Figure 3-50 Structure of an 'hovr' component compiled with the HMSkipItem identifier



If you examine a compiled version of an 'hovr' resource, you find that a component identified in the Rez input file by the HMSkipItem identifier consists of the following elements:

- n Size. The value 4, for the number of bytes contained in this component.
- n Type of data. The value 256.

Summary of the Help Manager

Pascal Summary

Constants

CONST

```

gestaltHelpMgrAttr      = 'help';    {Gestalt selector}
gestaltHelpMgrPresent  = 0;          {if this bit is set, then
                                     { Help Manager is present}

hmBalloonHelpVersion   = $0002;    {Help Manager version}
kBalloonWDEFID         = 126;      {resource ID of standard balloon }
                                     { 'WDEF' function}

kHMHelpID               = -5696;    {ID of various Help Manager }
                                     { resources (in Pack14 range); }
                                     { also used for 'hldr' resource ID}

{Help menu constants}
kHMAboutHelpItem       = 1;          {About Balloon Help menu item}
kHMHelpMenuID          = -16490;    {Help menu resource ID}
kHMShowBalloonsItem    = 3;          {Show/Hide Balloons menu item}

{HelpItem type for 'DITL' resources}
helpItem                = 1;          {help item}

{option bits for help resources}
hmDefaultOptions        = 0;          {use defaults}
hmUseSubID              = 1;          {use subrange resource IDs }
                                     { for owned resources}

hmAbsoluteCoords        = 2;          {ignore coords of window }
                                     { origin and treat upper-left }
                                     { corner of window as 0,0}

hmSaveBitsNoWindow     = 4;          {don't create window; save }
                                     { bits; no update event}

hmSaveBitsWindow       = 8;          {save bits behind window and }
                                     { generate update event}

hmMatchInTitle          = 16;        {match window by string }
                                     { anywhere in title string}

```

CHAPTER 3

Help Manager

```
{constants for hmmHelpType field of HMMessageRecord}
khmmString          = 1;          {Pascal string}
khmmPict            = 2;          {'PICT' resource ID}
khmmStringRes       = 3;          {'STR#' res ID and index}
khmmTEHandle        = 4;          {TextEdit handle}
khmmPictHandle      = 5;          {picture handle}
khmmTERes           = 6;          {'TEXT' and 'styl' resource ID}
khmmSTRRes          = 7;          {'STR ' resource ID}
{resource types for styled text in resources}
kHMTETextResType    = 'TEXT';    {'TEXT' resource type}
kHMTStyleResType    = 'styl';    {'styl' resource type}

{constants for whichState parameter when extracting help }
{ message records from 'hmnu' and 'hdlg' resources}
kHMEEnabledItem     = 0;          {enabled state for menu items; }
                                { contrlHilite value of 0 for }
                                { controls}
kHMDisabledItem     = 1;          {disabled state for menu items; }
                                { contrlHilite value of 255 for }
                                { controls}
kHMCheckedItem      = 2;          {enabled-and-checked state for }
                                { menu items; contrlHilite }
                                { value of 1 for controls that }
                                { are "on"}
kHMOtherItem        = 3;          {enabled-and-marked state for }
                                { menu items; contrlHilite }
                                { value between 2 and 253 for }
                                { controls}

{resource types for whichType parameter used when extracting }
{ help message}
kHMMenuResType      = 'hmnu';    {menu help resource type}
kHMDialogResType    = 'hdlg';    {dialog help resource type}
kHMWindListResType  = 'hwin';    {window help resource type}
kHMRectListResType  = 'hrct';    {rectangle help resource type}
kHMOverrideResType  = 'hovr';    {help override resource type}
kHMFinderApplResType = 'hldr';    {app icon help resource type}

{constants for method parameter in HMShowBalloon}
kHMRegularWindow    = 0;          {don't save bits; just update}
kHMSaveBitsNoWindow = 1;          {save bits; don't do update}
kHMSaveBitsWindow   = 2;          {save bits; do update event}
```

CHAPTER 3

Help Manager

```
{constants for help types in 'hmnu', 'hdlg', 'hrct', 'hovr', and }
{ 'hfdrr' resources--useful only for walking these resources}
kHMStringItem          = 1;           {Pascal string}
kHMPictItem            = 2;           {'PICT' resource ID}
kHMStringResItem       = 3;           {'STR#' resource ID & index}
kHMTEResItem           = 6;           {'TEXT' & 'styl' resource ID}
kHMSTRResItem          = 7;           {'STR ' resource ID}
kHMSkipItem            = 256;         {don't display a balloon}
kHMCompareItem         = 512;         {for 'hmnu', use help message
                                       { if menu item matches string}
kHMNamedResourceItem  = 1024;        {for 'hmnu', use menu item to
                                       { get a named resource}
kHMTrackCntlItem       = 2048;       {reserved}
```

Data Types

```
TYPE HMStringResType = {Help Manager string list record}
RECORD
    hmmResID: Integer; { 'STR#' resource ID}
    hmmIndex: Integer; {index of string}
END;

HMMessageRecPtr = ^HMMessageRecord;
HMMessageRecord = {help message record}
RECORD
    hmmHelpType: Integer; {type of next field}
    CASE Integer OF
        khmmString: (hmmString: Str255); {Pascal string}
        khmmPict: (hmmPict: Integer); {'PICT' resource ID}
        khmmStringRes: (hmmStringRes: HMStringResType);
                                {'STR#' resource
                                { ID and index}
        khmmTEHandle: (hmmTEHandle: TEHandle); {TextEdit handle}
        khmmPictHandle: (hmmPictHandle: PicHandle);
                                {picture handle}
        khmmTERes: (hmmTERes: Integer); {'TEXT'/'styl'
                                { resource ID}
        khmmSTRRes: (hmmSTRRes: Integer) {'STR ' resource ID}
    END;
```

Help Manager Routines

Determining Help Balloon Status

```
FUNCTION HMGetBalloons      : Boolean;
FUNCTION HMIIsBalloon      : Boolean;
```

Displaying and Removing Help Balloons

```
FUNCTION HMShowBalloon      (aHelpMsg: HMMessageRecord; tip: Point;
                             alternateRect: RectPtr; tipProc: Ptr;
                             theProc: Integer; variant: Integer;
                             method: Integer): OSErr;

FUNCTION HMShowMenuBalloon (itemNum: Integer; itemMenuID: Integer;
                             itemFlags: LongInt; itemReserved: LongInt;
                             tip: Point; alternateRect: RectPtr;
                             tipProc: Ptr; theProc: Integer;
                             variant: Integer): OSErr;

FUNCTION HMRemoveBalloon   : OSErr;
```

Enabling and Disabling Balloon Help Assistance

```
FUNCTION HMSetBalloons     (flag: Boolean): OSErr;
```

Adding Items to the Help Menu

```
FUNCTION HMGetHelpMenuHandle
                                     (VAR mh: MenuHandle): OSErr;
```

Getting and Setting the Font Name and Size

```
FUNCTION HMGetFont             (VAR font: Integer): OSErr;
FUNCTION HMGetFontSize         (VAR fontSize: Integer): OSErr;
FUNCTION HMSetFont             (font: Integer): OSErr;
FUNCTION HMSetFontSize         (fontSize: Integer): OSErr;
```

Setting and Getting Information for Help Resources

```
FUNCTION HMSetMenuResID       (menuID: Integer; resID: Integer): OSErr;
FUNCTION HMGetMenuResID       (menuID: Integer; VAR resID: Integer): OSErr;
FUNCTION HMScanTemplateItems
                                     (whichID: Integer; whichResFile: Integer;
                                     whichType: ResType): OSErr;

FUNCTION HMSetDialogResID     (resID: Integer): OSErr;
FUNCTION HMGetDialogResID     (VAR resID: Integer): OSErr;
```

Determining the Size of a Help Balloon

```

FUNCTION HMBalloonRect      (aHelpMsg: HMMessageRecord;
                           VAR coolRect: Rect): OSErr;

FUNCTION HMBalloonPict     (aHelpMsg: HMMessageRecord;
                           VAR coolPict: PicHandle): OSErr;

FUNCTION HMGetBalloonWindow
                           (VAR window: WindowPtr): OSErr;

```

Getting the Message of a Help Balloon

```

FUNCTION HMExtractHelpMsg  (whichType: ResType;
                           whichResID: Integer; whichMsg: Integer;
                           whichState: Integer;
                           VAR aHelpMsg: HMMessageRecord): OSErr;

FUNCTION HMGetIndHelpMsg   (whichType: ResType;
                           whichResID: Integer; whichMsg: Integer;
                           whichState: Integer;
                           VAR options: LongInt; VAR tip: Point;
                           VAR altRect: Rect; VAR theProc: Integer;
                           VAR variant: Integer;
                           VAR aHelpMsg: HMMessageRecord;
                           VAR count: Integer): OSErr;

```

Application-Defined Routines

```

FUNCTION MyBalloonDef      (variant: Integer; theBalloon: WindowPtr;
                           message: Integer; param: LongInt): LongInt;

FUNCTION MyTip             (tip: Point; structure: RgnHandle;
                           VAR r: Rect; VAR variant: Integer): OSErr;

```

C Summary

Constants

```

enum {
    #define gestaltHelpMgrAttr  'help'  /*Gestalt selector*/
    gestaltHelpMgrPresent      = 0      /*if this bit is set, then */
                                   /* Help Manager is present*/
};
enum {
    hmBalloonHelpVersion      = 0x0002, /*Help Manager version*/

```

CHAPTER 3

Help Manager

```
kBalloonWDEFID          = 126,          /*resource ID of standard balloon */
                          /* 'WDEF' function*/
kHMHelpID                = -5696,       /*ID of various Help Manager */
                          /* resources (in Pack14 range); */
                          /* also used for 'hldr' resource ID*/

/*Help menu constants*/
kHMAboutHelpItem        = 1,           /*About Balloon Help menu item*/
kHMHelpMenuID           = -16490,     /*Help menu resource ID*/
kHMShowBalloonsItem     = 3,           /*Show/Hide Balloons menu item*/

/*help item type for 'DITL' resources*/
HelpItem                = 1,           /*help item*/

/*option bits for help resources*/
hmDefaultOptions        = 0,           /*use defaults*/
hmUseSubID              = 1,           /*use subrange resource IDs */
                          /* for owned resources*/
hmAbsoluteCoords        = 2           /*ignore coords of window */
                          /* origin and treat upper-left */
                          /* corner of window as 0,0*/
};
enum {
    hmSaveBitsNoWindow    = 4,         /*don't create window; save */
                          /* bits; no update event*/
    hmSaveBitsWindow      = 8,         /*save bits behind window and */
                          /* generate update event*/
    hmMatchInTitle        = 16,        /*match window by string */
                          /* anywhere in title string*/

/*constants for hmmHelpType field of HMMessageRecord*/
khmmString              = 1,         /*Pascal string*/
khmmPict                = 2,         /*'PICT' resource ID*/
khmmStringRes           = 3,         /*'STR#' res ID and index*/
khmmTEHandle            = 4,         /*TextEdit handle*/
khmmPictHandle          = 5,         /*picture handle*/
khmmTERes               = 6,         /*'TEXT' and 'styl' resource ID*/
khmmSTRRes              = 7,         /*'STR ' resource ID*/
/*resource types for styled text in resources*/
#define kHMTETextResType  'TEXT'     /*'TEXT' resource type*/
#define kHMTStyleResType  'styl'     /*'styl' resource type*/
```

CHAPTER 3

Help Manager

```
/*constants for whichState parameter when extracting help */
/* message records from 'hmnu' and 'hdlg' resources*/
kHMEnabledItem          = 0,          /*enabled state for menu items; */
                               /* contrlHilite value of 0 for */
                               /* controls*/
};
enum {
    kHMDisabledItem      = 1,          /*disabled state for menu items; */
                               /* contrlHilite value of 255 for */
                               /* controls*/
    kHMCheckedItem       = 2,          /*enabled-and-checked state for */
                               /* menu items; contrlHilite */
                               /* value of 1 for controls that */
                               /* are "on"*/
    kHMOtherItem         = 3,          /*enabled-and-marked state for */
                               /* menu items; contrlHilite */
                               /* value between 2 and 253 for */
                               /* controls*/

/*resource types for whichType parameter used when extracting */
/* help message*/
#define kHMMenuResType   'hmnu'   /*menu help resource type*/
#define kHMDialogResType 'hdlg'   /*dialog help resource type*/
#define kHMWindListResType 'hwin' /*window help resource type*/
#define kHMRectListResType 'hrct' /*rectangle help resource type*/
#define kHMOverrideResType 'hovr' /*help override resource type*/
#define kHMFinderApplResType 'hfdr' /*app icon help resource type*/

/*constants for method parameter in HMShowBalloon*/
kHMRegularWindow      = 0,          /*don't save bits; just update*/
kHMSaveBitsNoWindow   = 1,          /*save bits; don't do update*/
kHMSaveBitsWindow     = 2          /*save bits; do update event*/
};
enum {
/*constants for help types in 'hmnu', 'hdlg', 'hrct', 'hovr', and */
/* 'hfdr' resources--useful only for walking these resources*/
    kHMStringItem        = 1,          /*Pascal string*/
    kHMPictItem          = 2,          /*'PICT' resource ID*/
    kHMStringResItem     = 3,          /*'STR#' resource ID & index*/
    kHMTEResItem         = 6,          /*'TEXT' & 'styl' resource ID*/
    kHMSTRResItem        = 7,          /*'STR ' resource ID*/
    kHMSkipItem          = 256,        /*don't display a balloon*/

```

CHAPTER 3

Help Manager

```
kHMCompareItem      = 512,      /*for 'hmnu', use help message */
                    /* if menu item matches string*/
kHMNamedResourceItem = 1024,    /*for 'hmnu', use menu item to */
                    /* get a named resource*/
kHMTrackCntlItem    = 2048     /*reserved*/
};
```

Data Types

```
struct HMStringResType {        /*Help Manager string list record*/
    short    hmmResID;          /*'STR#' resource ID*/
    short    hmmIndex;         /*index of string*/
};
typedef struct HMStringResType HMStringResType;

struct HMMessageRecord {       /*help message record*/
    short    hmmHelpType;      /*type of next field*/
    union {
        char        hmmString[256]; /*Pascal string*/
        short       hmmPict;        /*'PICT' resource ID*/
        Handle      hmmTEHandle;    /*TextEdit handle*/
        HMStringResType    hmmStringRes; /*'STR#' resource ID and index*/
        short       hmmPictRes;     /*unused*/
        Handle      hmmPictHandle;  /*picture handle*/
        short       hmmTERes;       /*'TEXT'/'styl' resource ID*/
        short       hmmSTRRes;     /*'STR ' resource ID*/
    } u;
};
typedef struct HMMessageRecord HMMessageRecord;
typedef HMMessageRecord *HMMessageRecPtr;
```

Help Manager Routines

Determining Help Balloon Status

```
pascal Boolean HMGetBalloons
                                (void);
pascal Boolean HMIIsBalloon     (void);
```

Displaying and Removing Help Balloons

```

pascal OSErr HShowBalloon (const HMessageRecord *aHelpMsg, Point tip,
                          RectPtr alternateRect, Ptr tipProc,
                          short theProc, short variant, short method);

pascal OSErr HShowMenuBalloon
                          (short itemNum, short itemMenuID,
                          long itemFlags, long itemReserved,
                          Point tip, RectPtr alternateRect,
                          Ptr tipProc, short theProc, short variant);

pascal OSErr HRemoveBalloon
                          (void);

```

Enabling and Disabling Balloon Help Assistance

```

pascal OSErr HMSetBalloons (Boolean flag);

```

Adding Items to the Help Menu

```

pascal OSErr HMGetHelpMenuHandle
                          (MenuHandle *mh);

```

Getting and Setting the Font Name and Size

```

pascal OSErr HMGetFont      (short *font);
pascal OSErr HMGetFontSize (short *fontSize);
pascal OSErr HMSetFont      (short font);
pascal OSErr HMSetFontSize (short fontSize);

```

Setting and Getting Information for Help Resources

```

pascal OSErr HMSetMenuResID
                          (short menuID, short resID);

pascal OSErr HMGetMenuResID
                          (short menuID, short *resID);

pascal OSErr HMScanTemplateItems
                          (short whichID, short whichResFile,
                          ResType whichType);

pascal OSErr HMSetDialogResID
                          (short resID);

pascal OSErr HMGetDialogResID
                          (short *resID);

```

Determining the Size of a Help Balloon

```
pascal OSErr HMBalloonRect (const HMMessageRecord *aHelpMsg,
                           Rect *coolRect);
pascal OSErr HMBalloonPict (const HMMessageRecord *aHelpMsg,
                            PicHandle *coolPict);
pascal OSErr HMGetBalloonWindow
                           (WindowPtr *window);
```

Getting the Message of a Help Balloon

```
pascal OSErr HMExtractHelpMsg
                           (ResType whichType, short whichResID,
                           short whichMsg, short whichState,
                           HMMessageRecord *aHelpMsg);
pascal OSErr HMGetIndHelpMsg
                           (ResType whichType, short whichResID,
                           short whichMsg, short whichState,
                           long *options, Point *tip, Rect *altRect,
                           short *theProc, short *variant,
                           HMMessageRecord *aHelpMsg, short *count);
```

Application-Defined Routines

```
pascal long MyBalloonDef (short variant, WindowPtr theBalloon,
                          short message, long param);
pascal OSErr MyTip (Point tip, RgnHandle structure,
                   Rect *r, short *variant);
```

Assembly-Language Summary

Data Structures

Help Message Data Structure

0	hmmHelpType	word	Resource type
2	hmmHelpMessage	variable	Help balloon message

Trap Macros

Trap Macros Requiring Routine Selectors

_Pack14

Selector	Routine
\$0002	HMRemoveBalloon
\$0003	HMGetBalloons
\$0007	HMIsBalloon
\$0104	HMSetBalloons
\$0108	HMSetFont
\$0109	HMSetFontSize
\$010C	HMSetDialogResID
\$0200	HMGetHelpMenuHandle
\$020A	HMGetFont
\$020B	HMGetFontSize
\$020D	HMSetMenuResID
\$0213	HMGetDialogResID
\$0215	HMGetBalloonWindow
\$0314	HMGetMenuResID
\$040E	HMBalloonRect
\$040F	HMBalloonPict
\$0410	HMScanTemplateItems
\$0711	HMExtractHelpMsg
\$0B01	HMShowBalloon
\$0E05	HMShowMenuBalloon
\$1306	HMGetIndHelpMsg

Result Codes

noErr	0	No error
fnOpnErr	-38	File not open
paramErr	-50	Error in parameter list
memFullErr	-108	Not enough room in heap zone
resNotFound	-192	Unable to read resource
hmHelpDisabled	-850	Help balloons are not enabled
hmBalloonAborted	-853	Because of constant cursor movement, the help balloon wasn't displayed
hmSameAsLastBalloon	-854	Menu and item are same as previous menu and item
hmHelpManagerNotInited	-855	Help menu not set up
hmSkippedBalloon	-857	No help message to fill in
hmWrongVersion	-858	Wrong version of Help Manager resource
hmUnknownHelpType	-859	Help message record contained a bad type
hmOperationUnsupported	-861	Invalid value passed in the method parameter
hmNoBalloonUp	-862	No balloon showing
hmCloseViewActive	-863	Balloon can't be removed because Close View is in use

List Manager

Contents

Introduction to Lists	4-4
Appearance of Lists	4-4
Selection of List Items	4-9
Keyboard Navigation of Lists	4-15
Movement of a Selection With Arrow Keys	4-15
Extension of a Selection With Arrow Keys	4-16
Type Selection in a Text-Only List	4-20
Multiple Lists in a Window	4-20
About the List Manager	4-22
Using the List Manager	4-26
Creating a List	4-27
Adding Rows and Columns to a List	4-30
Responding to Events Affecting a List	4-32
Working With List Selections	4-34
Customizing Cell Highlighting	4-38
Manipulating List Cells	4-40
Searching a List for a Particular Item	4-43
Supporting Keyboard Navigation of Lists	4-45
Supporting Type Selection of List Items	4-45
Supporting Arrow-Key Navigation of Lists	4-48
Supporting the Anchor Algorithm for Extending Lists With Arrow Keys	4-52
Outlining the Current List	4-53
Writing Your Own List Definition Procedure	4-58
Responding to the Initialization Message	4-60
Responding to the Draw Message	4-60
Responding to the Highlighting Message	4-62
Responding to the Close Message	4-62
Using the Pictures List Definition Procedure	4-63

List Manager Reference	4-65
Data Structures	4-65
The Cell Record	4-65
The Data Handle	4-66
The List Record	4-66
List Manager Routines	4-70
Creating and Disposing of Lists	4-70
Adding and Deleting Columns and Rows To and From a List	4-73
Determining or Changing the Selection	4-77
Accessing and Manipulating Cell Data	4-79
Responding to Events Affecting Lists	4-84
Modifying a List's Appearance	4-87
Searching a List for a Particular Item	4-90
Changing the Size of Cells and Lists	4-91
Getting Information About Cells	4-93
Application-Defined Routines	4-96
List Definition Procedures	4-96
Match Functions	4-99
Click-Loop Procedures	4-100
Summary of the List Manager	4-102
Pascal Summary	4-102
Constants	4-102
Data Types	4-102
List Manager Routines	4-103
Application-Defined Routines	4-105
C Summary	4-106
Constants	4-106
Data Types	4-106
List Manager Routines	4-107
Application-Defined Routines	4-109
Assembly-Language Summary	4-110
Data Structures	4-110
Trap Macros	4-111

List Manager

This chapter describes how your application can use the List Manager to create scrollable lists that allow the user to select one or more of a group of items. The List Manager lets you create one-column lists or multicolumn lists. By default, it creates lists that contain only unstyled text, but with extra effort, you can use the List Manager to create lists that display items graphically.

Read the information in this chapter if you need to allow users to select one or more items from a group of items. If you only need to allow the user to select one item from a small group of items, a pop-up menu may be more appropriate than a list. If, however, you would like the user to be able to select one of many items or to be able to select multiple items, the List Manager provides a convenient and intuitive interface.

If the contents of a group of items might change, use a list rather than a pop-up menu. Users generally expect the contents of pop-up menus to remain the same, whereas a list provides instant visual feedback when its contents change, thus preventing user confusion. For example, you might use the List Manager to create a list of appointments and allow the user to add or remove appointments to or from the list.

Although the List Manager can handle small, simple lists effectively, it is not suitable for displaying large amounts of data (such as that used by a spreadsheet application). The List Manager cannot maintain lists that occupy more than 32 KB of memory, and performance degrades sharply well before the 32 KB limit. Also, the List Manager expects all cells to be equal in size. Thus, if you are writing a spreadsheet application, you should use the Control Manager and your own internal data structures. However, you should still read the sections of this chapter that concern selection of list items so that your application can have a user interface consistent with the List Manager's.

To use this chapter, you should be familiar with the concepts of the Control Manager, the Event Manager, and the Window Manager, and, if you plan to create a list in a modal or modeless dialog box, with the Dialog Manager. For more information on these topics, see *Inside Macintosh: Macintosh Toolbox Essentials*.

This chapter begins by describing lists and the user interface for them. The chapter then discusses how you can

- n create lists
- n respond to events affecting lists
- n get information about a list
- n get or change the contents of list items
- n search through a list for a particular item
- n support keyboard navigation of lists
- n manage multiple lists within the same window or dialog box
- n write your own list definition procedure to handle nonstandard lists, such as lists of pictures

Introduction to Lists

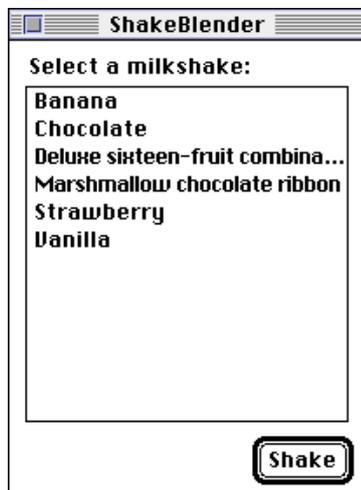
You can use the List Manager to store and update elements of data within a list and to display the list in a rectangle within a window. The List Manager provides routines that allow you to create, manipulate, and display lists. It can also respond appropriately to a mouse click within a list by, for example, scrolling a list when the user clicks in a scroll bar. Thus, using the List Manager is easier than using the Control Manager and QuickDraw to create a scrolling list of items.

Appearance of Lists

A **list** is a series of items displayed within a rectangle. Each item in a list is contained within an invisible rectangular **cell**. All cells in a list created by the List Manager are the same size, but cells may contain different types of data. Your application may allow the user to select one or more items in a list by clicking them. When a user selects an item, the List Manager highlights the cell containing the item.

Figure 4-1 illustrates a window that includes a list of six items.

Figure 4-1 A one-column, text-only list without a scroll bar

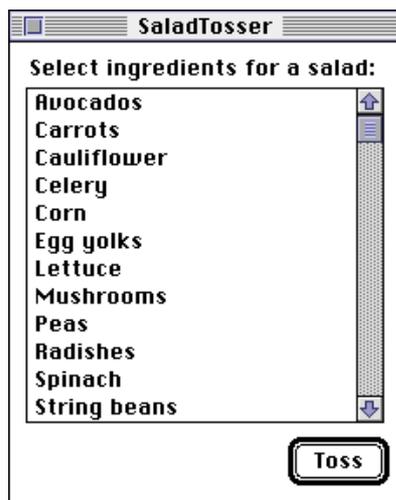


List Manager

The font used for a text-only list is determined by the font of the current graphics port. Usually, you should create lists in the system font. Regardless of the font your application uses, if a string is too long to fit in a list using the current font, the List Manager uses condensed type in an effort to fit it. If the string is still too long, the List Manager truncates the string displayed and appends an ellipsis to it. Both of these techniques are illustrated in Figure 4-1. Both the strings “Deluxe sixteen-fruit combination” and “Marshmallow chocolate ribbon” are condensed; the first of these is also truncated.

Lists may contain a vertical scroll bar, a horizontal scroll bar, or both. By using scroll bars, you can include more items in a list than can fit in the list’s rectangle, and the user can scroll to view multiple items. If there is any chance that a list may contain more cells than can fit within the list’s rectangle, you should include a scroll bar in the list. Figure 4-2 illustrates a list that includes a vertical scroll bar.

Figure 4-2 A one-column, text-only list with a vertical scroll bar



List Manager

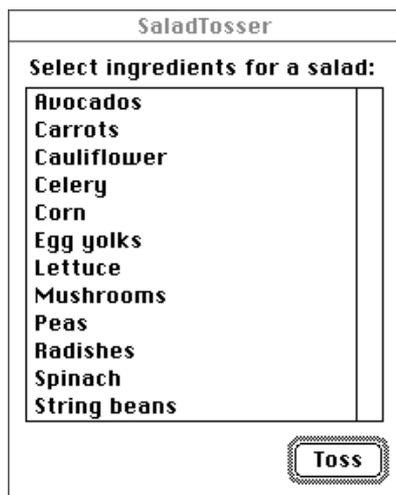
If a list includes a scroll bar but there are a small enough number of items in the list that all the list's items are visible, the List Manager automatically disables the scroll bar. For example, Figure 4-3 shows such a list.

Figure 4-3 A list whose scroll bar has been disabled



When a window containing one or more lists becomes deactivated, your application should call the List Manager to deactivate the lists as well. Figure 4-4 shows a deactivated list.

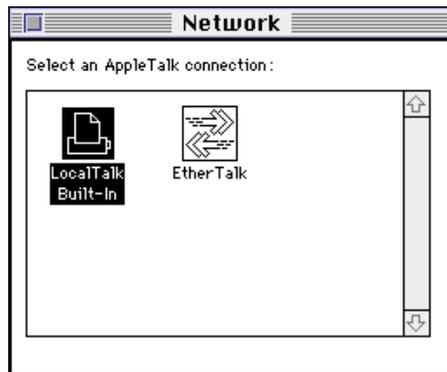
Figure 4-4 A deactivated list



List Manager

Your application can create one-column lists of the type illustrated in Figure 4-2 through Figure 4-4 using the List Manager. Your application can also create lists that contain two or more columns. For example, the Network control panel allows the user to select a network connection from a three-column list. In Figure 4-5, there are only two possible network connections, so there are no items in the third column of the list.

Figure 4-5 A list containing multiple columns and graphical elements

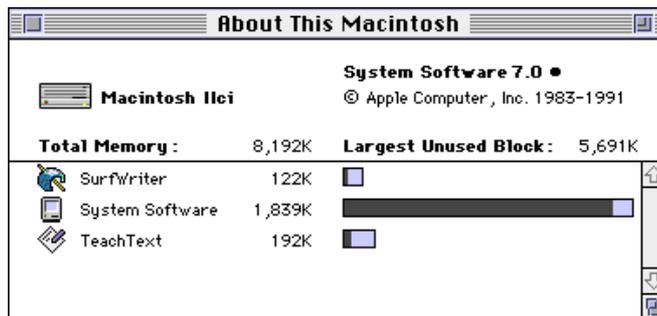


Note that the list in Figure 4-5 contains graphical elements rather than just text. To create a list with graphical elements, you must write a custom list definition procedure, because the default list definition procedure supports only the display of text. A **list definition procedure** is a code resource of type 'LDEF' that defines the characteristics of a list. In addition to using a list definition procedure to support graphical items in lists, you can write one to customize the display of text in a list. For example, to use styled text in a list, you would need to create a list definition procedure.

List Manager

You can also use a list definition procedure to create lists that contain cells which display more than one type of information. For example, the Finder's "About This Macintosh" modeless dialog box contains a list of applications that are currently in use. Each cell in the list includes a small icon of the application, the name of the application, the amount of memory in the application's partition, and a graphical indication of how much of that memory has been used, as illustrated in Figure 4-6.

Figure 4-6 A list of items whose cells display more than one type of information



Note that the list in Figure 4-6 is not a multicolumn list. It is a one-column list, but each cell of the list displays several types of information.

Your application specifies whether the List Manager should leave room for a size box, although your application is responsible for drawing any grow icon; the List Manager does not draw the grow icon automatically. Usually, size boxes are useful only for lists that are on the bottom of the windows that contain them, like the list in Figure 4-6. In this case, resizing the window changes the size of the list. Your application should ensure that the user cannot shrink the size of the window so much that the list is no longer visible.

In addition to requesting a vertical scroll bar, your application may request that the List Manager use a horizontal scroll bar for your list. A second scroll bar is useful mainly if your application allows the user to resize a window containing a list both horizontally and vertically so that only a portion of the list is visible. A second scroll bar is also useful to allow the user to scroll through a table of cells. Usually, however, if you are implementing a spreadsheet-like application, you should not be using the List Manager. Most multicolumn lists created by the List Manager, such as the one illustrated in Figure 4-5, should not include two scroll bars.

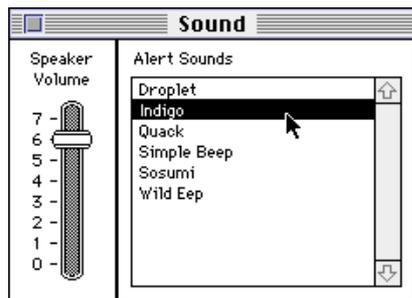
Selection of List Items

Sometimes, an application might create a list simply for the user to view. For example, a desktop-publishing application might create a list of fonts used in a document. The user should be able to scroll the list to examine all of the fonts, but the application can ensure (by ignoring mouse clicks on list cells) that clicking cells of the list has no effect. More often, however, applications create lists so that users can select items from them by clicking the items' cells.

Your application can allow the user to select items in a list by calling the `LClick` function whenever a mouse-down event occurs. The `LClick` function handles all user interaction, including highlighting of items, until the user releases the mouse button. The `LClick` function also examines the state of the modifier keys (specifically the Shift and Command keys) and changes the selection appropriately.

Figure 4-7 illustrates the Sound control panel, which allows users to select a system alert sound from a list of available alert sounds.

Figure 4-7 A list with an item selected

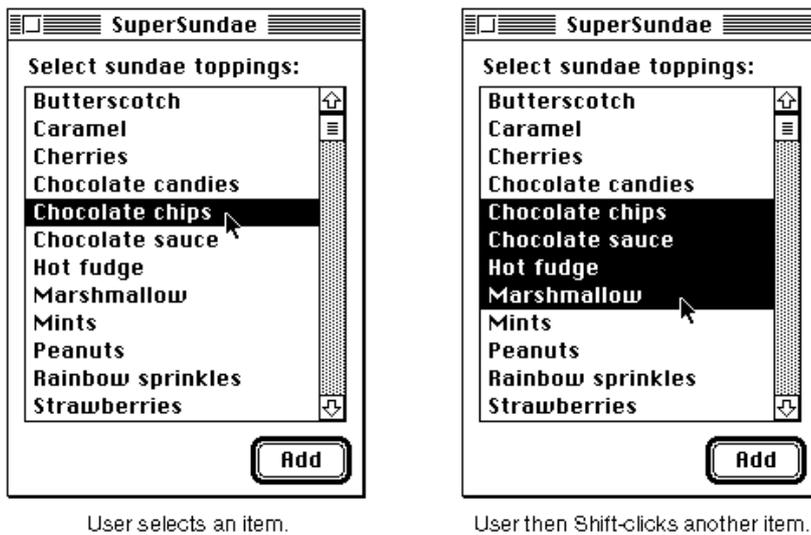


When the user selects a cell (such as the “Indigo” system alert sound) by clicking the item’s cell, the List Manager highlights the cell. In the list shown in Figure 4-7, the user can also select a cell by clicking another cell and dragging the cursor to the desired cell (such as the cell containing “Indigo”) before releasing the mouse button. This type of list allows the user to select only one item, because there can be only one system alert sound. While you can create a list that has this behavior, the List Manager by default allows the user to select a range of cells or even several discontinuous ranges of cells by using the Shift and Command keys.

List Manager

The user can use the Shift key to select a range of cells. By pressing the Shift key when clicking a cell, the user can select all items in a given range. For example, in Figure 4-8 the user extends a selection of just one item to cover several items by pressing the Shift key and clicking another item. The List Manager then highlights all cells ranging from the already selected cell to the newly selected cell, thus making the entire range of cells selected. In a one-column list, like that in Figure 4-8, the List Manager highlights a rectangular range of cells in response to a Shift-click.

Figure 4-8 Selection of a range of items in a list



After pressing the mouse button while also pressing the Shift key (but before releasing the mouse button), the user can extend or shrink the range of cells selected by dragging the cursor. The user can even drag the cursor below the list to select a range that includes items not initially visible. For example, Figure 4-9 illustrates the effect of dragging after the initial selection of the range of cells illustrated in Figure 4-8.

Figure 4-9 Effect of dragging after Shift-clicking

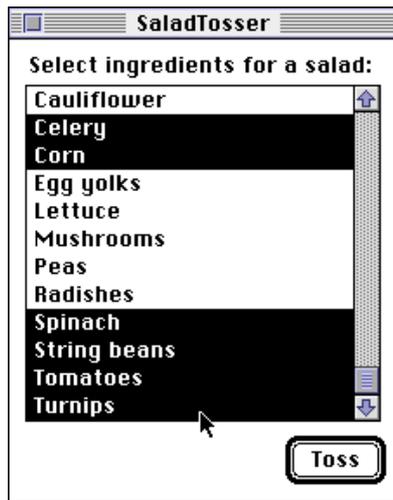
Virtually every application that supports Shift-clicking to extend list selections should also support the selection of discontinuous ranges of list cells. The default behavior of the List Manager is to allow a user to add a cell to the current selection by pressing the Command key when clicking a cell. If a user Command-clicks a cell that is already selected, the List Manager removes the cell from the selection.

To add or remove a range of cells from the current selection, a user can press the mouse button while also pressing the Command key and then drag the cursor over other cells. The List Manager determines whether to add or remove selections in a range of cells by checking the status of the first cell clicked in. If that cell is initially selected, then Command-dragging deselects all cells in the range over which the cursor passes. If that cell is initially not selected, then Command-dragging selects all cells in the range over which the cursor passes. Once the user changes a cell's selection status by Command-dragging over a cell, the selection status of the cell stays the same for the duration of the drag even if the user moves the cursor back over the cell. In this way, the use of the Command key differs from that of the Shift key.

List Manager

Figure 4-10 illustrates use of the Command key. This example shows a list created by an application that allows a user to choose what vegetables to include in a salad to be tossed by a device attached to the computer.

Figure 4-10 Selection of discontinuous items in a list

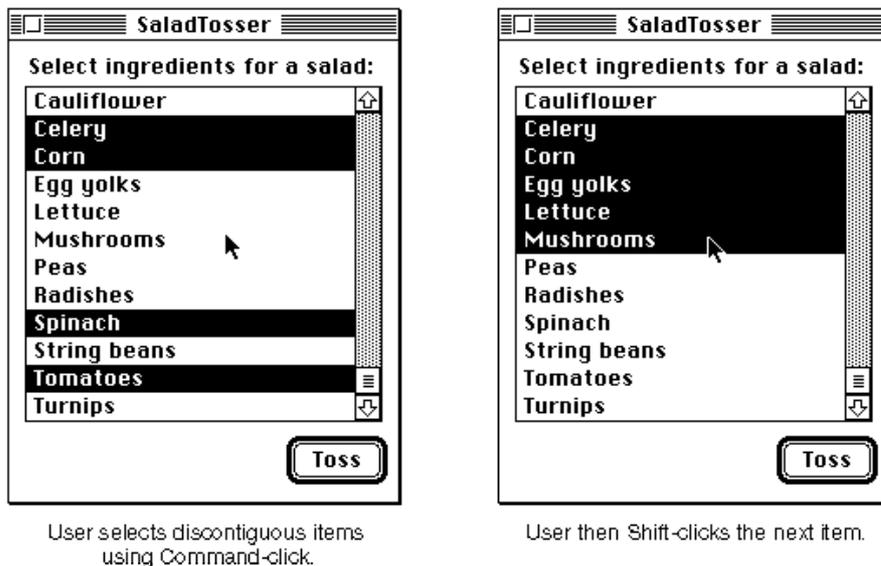


Initially, the user has selected "Celery" and "Corn." By pressing the Command key and the mouse button while the cursor is over the item "Spinach," then dragging the cursor downward to "Turnips" (which automatically scrolls into view), the user can select additional items. Without the feature of Command-clicking to select discontinuous ingredients, a user of this list would be able to select only alphabetical ranges of ingredients for the salad.

List Manager

If a user Shift-clicks a cell after having created discontinuous selection ranges, the discontinuity is lost. The List Manager selects all cells in the range of the first selected cell and the newly selected cell, unless the newly selected cell precedes the first selected cell, in which case the List Manager selects all cells in the range of the newly selected cell and the last selected cell. Figure 4-11 illustrates how the selection changes when a user Shift-clicks a cell that follows one range of selected cells but precedes another. In this example, after selecting “Celery,” “Corn,” “Spinach,” and “Tomatoes,” the user Shift-clicks the item labeled “Mushrooms.”

Figure 4-11 Effect of Shift-clicking in a list that contains discontinuous items



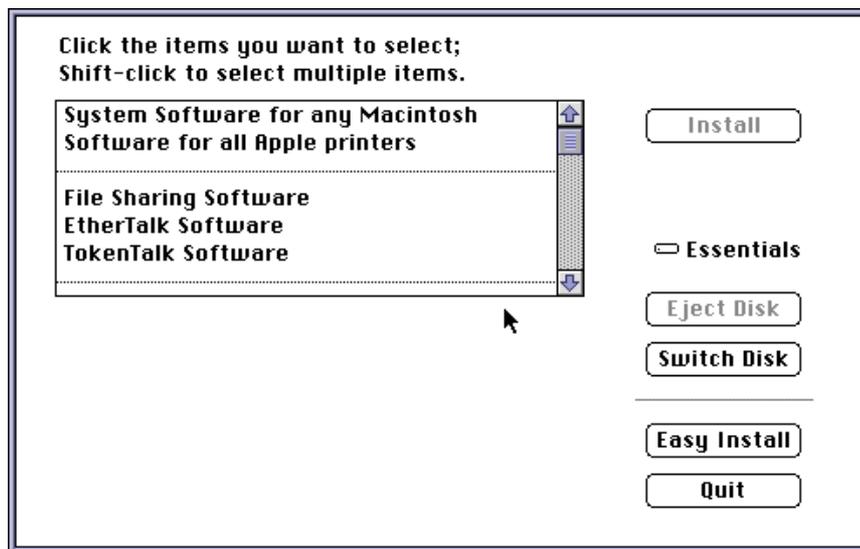
If a user presses both the Command and Shift keys when clicking a cell, then the pressing of the Shift key is ignored and the List Manager behaves as if only the Command key is pressed.

Your application can customize the algorithm the List Manager uses to manage the selection of list items. (You can do this by setting one or more flags in the `selFlags` field of the list record.) For example, your application can permit the user to select only one element of a list at a time, in which case the Shift and Command keys are ignored.

List Manager

Some applications may wish to make the Shift key work in lists just like the Command key. This is especially useful for applications geared toward novice users, who might not think of using the Command key to select several discontinuous items in a list. If your application uses a nonstandard behavior, then it should make this clear to the user. For example, the Installer application includes a list that treats Shift-clicks like Command-clicks, and it indicates to the user that Shift-clicking selects multiple items. This is illustrated in Figure 4-12.

Figure 4-12 Notifying the user of nonstandard list behavior



The List Manager provides a number of other ways that your application can customize the selection of items within a list. In particular, your application can

- n allow only one item to be selected at a time. (By default, the List Manager allows multiple items to be selected.)
- n allow the user to select a range of items by clicking the first item and dragging to the last item without necessarily pressing the Shift or Command key. Ordinarily, dragging in this manner results in only the last item's being selected.
- n disable discontinuous selections, while still allowing the user to select a range of items.
- n cause all previously selected cells to be deselected when the user Shift-clicks.
- n allow the user to deselect a range of cells by Shift-dragging. Ordinarily, Shift-dragging causes cells to become selected even if the first cell clicked is already selected.

List Manager

- n disable the feature that allows the user to shrink a selection by Shift-clicking to select a range of cells and then dragging the cursor to a position within that range. When this feature is disabled, all cells in the cursor's path during a Shift-drag become selected even if the user drags the cursor back over the cell.
- n turn off the highlighting of selected cells that contain no data.

“Customizing Cell Highlighting” beginning on page 4-38 discusses the techniques that your application can use to customize the selection of lists.

Keyboard Navigation of Lists

Although it is easy to use the mouse to select list items, some users prefer to use the keyboard. Keyboard navigation and selection of list items is a particularly useful feature for long lists. Your application should support keyboard navigation of lists in two ways. First, your application should support the use of the arrow keys to move or extend a selection. Second, if your application uses text-only lists (or lists whose items can be identified by text strings), your application should allow the user to select an item simply by typing the text associated with it.

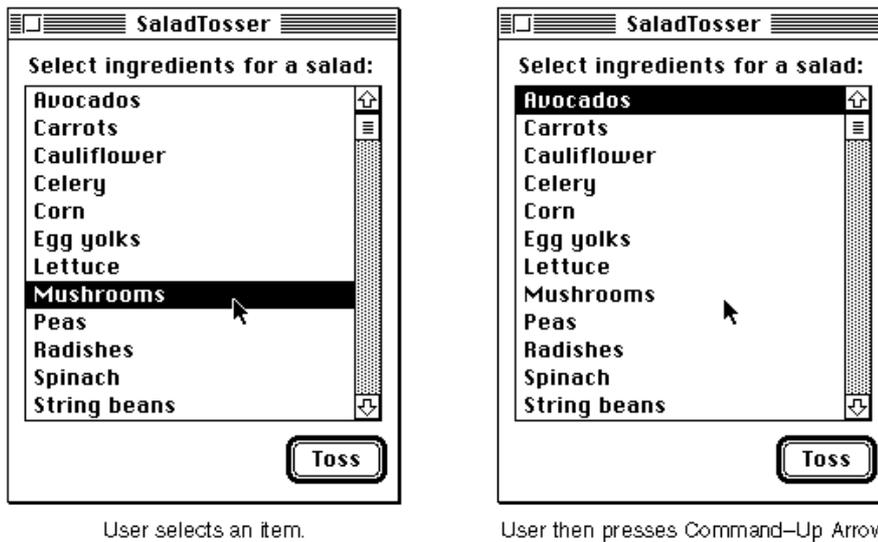
The List Manager does not provide any routines to automatically handle keyboard navigation of lists, but your application can provide code to manage keyboard navigation of lists. “Supporting Keyboard Navigation of Lists” beginning on page 4-45 shows code that handles keyboard navigation.

Movement of a Selection With Arrow Keys

When a user presses an arrow key and is not pressing the Shift or Command key, the user is attempting to move the selection one cell. For example, your application should respond to the pressing of the Up Arrow key by selecting the cell that is one cell above the first selected cell and deselecting any other selected cells. If the first selected cell is already in the first row, then your application should respond simply by deselecting all cells other than that first selected cell. Your application should respond to the pressing of the Left Arrow key by moving the selection one cell to the left. Your application should respond to the pressing of the Down Arrow key or the Right Arrow key by selecting the cell that is one cell below or to the right of the last selected cell and deselecting any other selected cells. If the last selected cell is already in the last row, then your application should respond simply by deselecting all cells other than that last selected cell.

When a user presses an arrow key while pressing the Command key, your application should move the first (or last) selected cell as far as it can move in the appropriate direction. For example, Command–Left Arrow indicates that the first selected cell should be moved as far left as possible (and all other cells should be deselected). Figure 4-13 illustrates how an application responds to the pressing of the Command–Up Arrow keys.

Figure 4-13 Response to pressing the Command–Up Arrow keys



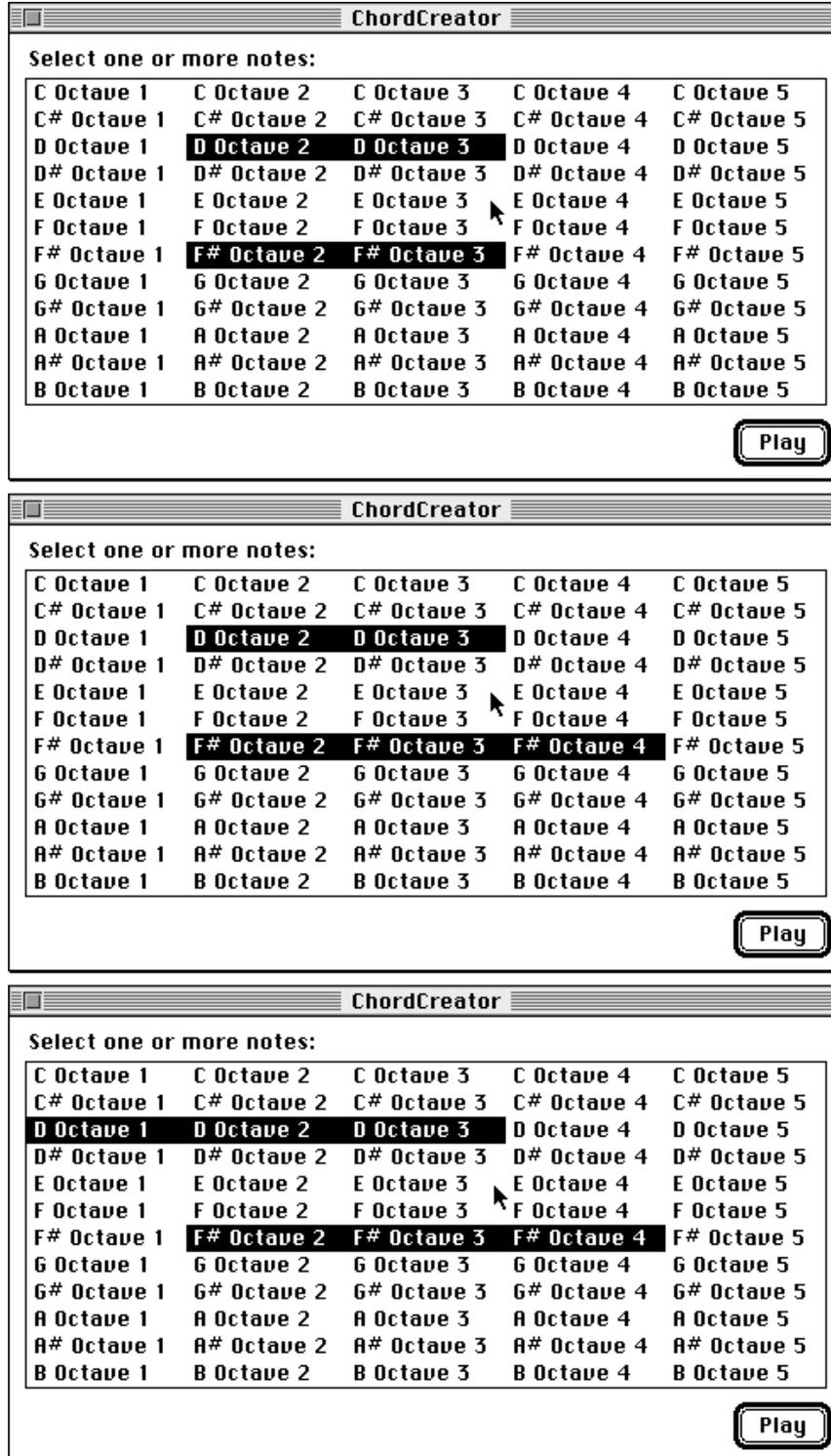
Extension of a Selection With Arrow Keys

A user may press the Shift key when pressing an arrow key to extend the current selection. There are two different algorithms that your application can use to respond to a Shift–arrow key combination.

The first potential response is the *extend algorithm*, in which your application simply finds the first (or last) selected cell, and then selects another cell in the direction of the arrow key. For example, if the user presses Shift–Right Arrow, your application should find the last selected cell and highlight the cell one column to the right of it, unless that cell is already highlighted. If the user presses Command–Shift–Up Arrow, your application should select the cell in the first row that was in the same column as the first selected cell and select all cells in between.

Figure 4-14 shows the effect of the extend algorithm when the user selects items using the Shift key and arrow keys. In this example, after selecting two discontinuous ranges, the user then presses Shift–Right Arrow, extending the last selected cell by one cell to the right. The user then presses Shift–Left Arrow, extending the selection one cell to the left of the first selected cell.

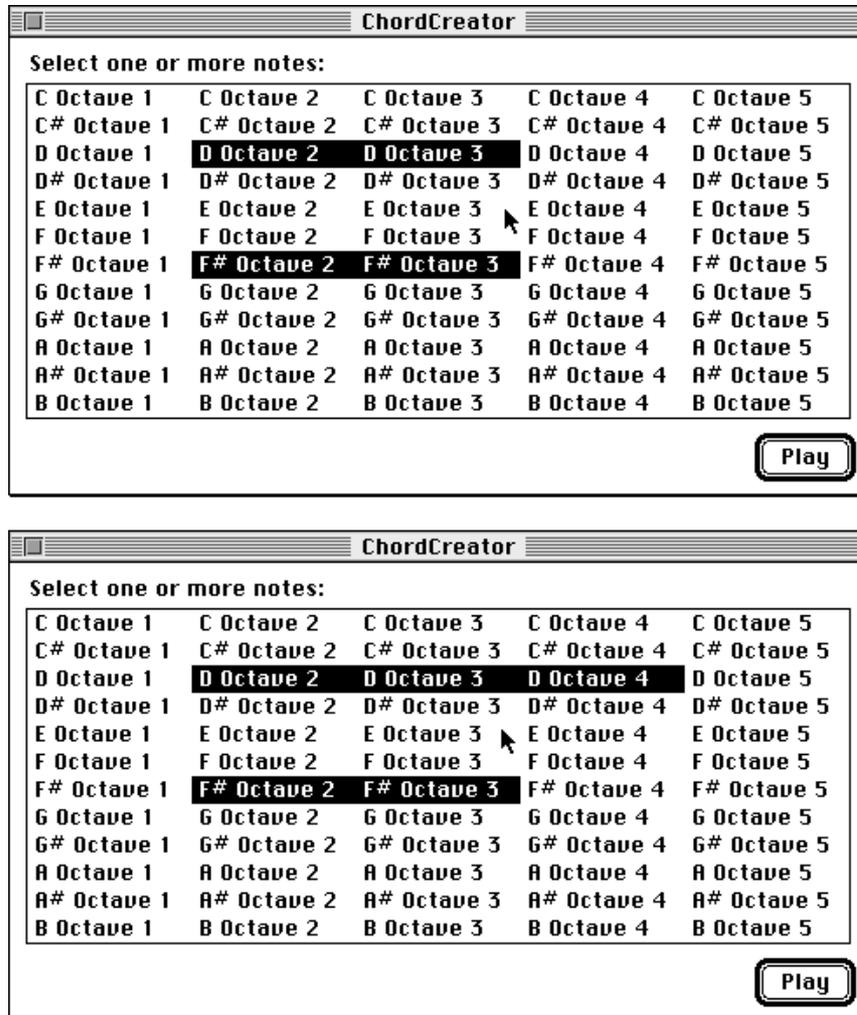
Figure 4-14 Response to user making a discontinuous selection, then pressing Shift–Right Arrow followed by Shift–Left Arrow using the extend algorithm



List Manager

While the extend algorithm is intuitive and works well for simple lists, a more powerful algorithm for managing extensions of selections with the arrow keys is the *anchor algorithm*. This algorithm is far more difficult to implement than the extend algorithm, but allows the user more power than the extend algorithm to extend a list in whatever way is desired, and it works more intuitively both for lists that are likely to contain many discontinuous items and for multicolumn lists.

The anchor algorithm works by moving the user's selection relative to an anchor cell. The application should determine which cell to make the anchor cell by examining the last cell in the rectangular range of cells last selected by the user. If the user has pressed either the Right Arrow or Down Arrow key, the anchor cell should be the first cell in this range; otherwise, it should be the last cell. The application then finds the cell that is on the other end of the rectangular range of cells last selected by the user. It then attempts to move this cell in the direction specified by the arrow key, and it highlights all cells in the rectangle whose corners are the anchor cell and the moving cell. Figure 4-15 illustrates this process.

Figure 4-15 Response to Shift–Right Arrow using the anchor algorithm

The top window of Figure 4-15 shows two rectangular ranges of selected cells. Suppose the application determines that the range of cells last selected by the user is the range containing “D Octave 2” and “D Octave 3.” Because the user pressed Shift–Right Arrow, the application designates the first cell in this range to be the anchor cell. It then extends each row of the rectangular range one cell to the right, as shown in the bottom window of Figure 4-15.

List Manager

The application must remember the anchor cell in case the user clicks another Shift–arrow key combination before making any other changes to the list. If this occurs, the application should keep the same anchor cell. Thus in Figure 4-15, if, after pressing Shift–Right Arrow, the user presses Shift–Left Arrow, then the application keeps the same anchor cell (ordinarily, if the Shift–Left Arrow keys are pressed, the last cell in the range becomes the anchor cell). The rectangular range of cells previously extended one cell to the right thus reverts to its original state. Therefore, if your application supports the anchor algorithm, the user can use Shift–arrow key combinations to extend a rectangular range of cells in any direction around an anchor cell that is determined by the first arrow key pressed.

Type Selection in a Text-Only List

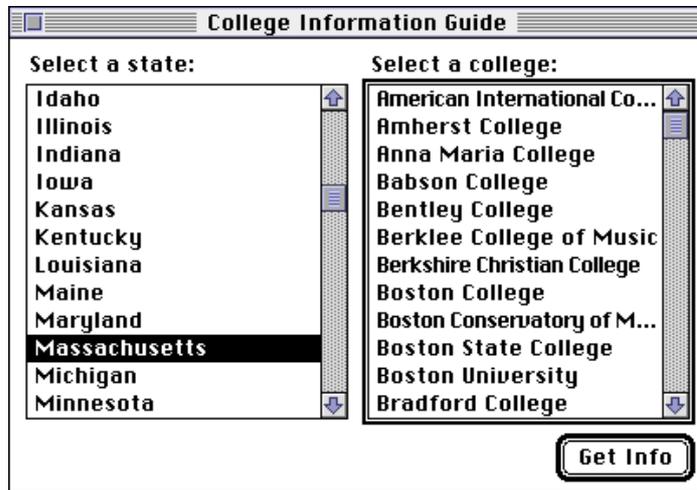
In a text-only list, when the user types the name of an item in a list, your application should respond by scrolling to that item and selecting it. This behavior (allowing a user to type the name of an item in a list to select it) is known as **type selection**. Rather than requiring the user to type the entire name of a list item, however, your application should continually attempt to determine the best match in the list for the user's typing.

In particular, every time the user types a character, your application should add it to a string that keeps track of the characters the user has typed in searching the list. Your application should attempt to find an exact match for this string, or if no exact match exists, your application should select the first item that alphabetically follows the text indicated by the string.

Sometimes the user may start to type the name of one list item and then type the name of another. Your application should support this by automatically resetting the internal string used to keep track of the user's typing after a given amount of time has elapsed without the user's pressing a key. To compute the amount of time after which your application should reset the string, you can use a formula (provided later in this chapter) that depends on the value the user sets for the autokey threshold in the Keyboard control panel. For users who specify a long delay until keys repeat, your application should use a long time span before it resets the internal string it uses to keep track of the user's typing.

Multiple Lists in a Window

In a window with multiple lists that support keyboard navigation, you need to show which list is the target of keyboard input. To help the user in such a window, your application should draw a 2-pixel-wide outline around the current list, that is, the list that would be affected by typing. The box should surround the entire list, including any scroll bars, and there should be 1 pixel of white space between the outline and the list's border. Figure 4-16 illustrates a window containing more than one list.

Figure 4-16 An outlined list in a window with more than one list

In Figure 4-16, the second list is outlined. Thus, the user knows that using the keyboard affects this list only. Your application should allow the user to press the Tab key to move the outline to the next list in a window. In a window with more than two lists, your application should allow the user to press Shift-Tab to move the outline to the previous list in a window.

Ordinarily, your application should not outline a list that is the only list in its window. However, if there is an editable text item in a dialog box containing a list, or if keyboard input could have some other effect, then your application should outline a list when the user can navigate it with the keyboard. The user should be able to use the Tab key to switch between a list and an editable text item; however, there is no need to outline the editable text item, since the insertion point indicates to the user that using the keyboard results in any text being inserted there.

When a window containing multiple lists is deactivated, your application should remove the outline from the current list and not replace it until the window is activated.

About the List Manager

The List Manager uses a list record to keep track of information about a list. In most cases your application can get or set the information in a list record using List Manager routines. When necessary, your application can examine fields of the list record directly.

Each cell in a list can be described by a data structure of type `Cell`:

```
TYPE Cell = Point;
```

The `Cell` data type has the same structure as the `Point` data type; however, the fields (horizontal and vertical coordinates) of a cell record have different meaning. The horizontal coordinate of a cell specifies its column number, and the vertical coordinate of a cell specifies the cell's row number. Note, however, that the first cell in a list is defined to be cell (0,0). So a cell with coordinates (3,4) is in the fourth column and fifth row. Thus you can visually identify a cell's coordinates using the formula (*column-1, row-1*). Figure 4-17 illustrates a list in which each cell item's text is set to the coordinates of the cell.

Figure 4-17 Coordinates of cells

Cell coordinates			
(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)
(0,4)	(1,4)	(2,4)	(3,4)
(0,5)	(1,5)	(2,5)	(3,5)
(0,6)	(1,6)	(2,6)	(3,6)
(0,7)	(1,7)	(2,7)	(3,7)

A list record is defined by the `ListRect` data type.

```
TYPE ListRec =
RECORD
    rView:      Rect;           {list's display rectangle}
    port:      GrafPtr;       {list's graphics port}
    indent:    Point;         {indent distance for drawing}
    cellSize:  Point;         {size in pixels of a cell}
    visible:   Rect;         {boundary of visible cells}
    vScroll:   ControlHandle; {vertical scroll bar}
    hScroll:   ControlHandle; {horizontal scroll bar}
    selFlags:  SignedByte;   {selection flags}
    lActive:   Boolean;       {TRUE if list is active}
```

List Manager

```

    lReserved: SignedByte;      {reserved}
    listFlags: SignedByte;     {automatic scrolling flags}
    clikTime: LongInt;         {TickCount at time of last click}
    clikLoc: Point;            {position of last click}
    mouseLoc: Point;          {current mouse location}
    lClikLoop: Ptr;           {routine called by LClick}
    lastClick: Cell;          {last cell clicked}
    refCon: LongInt;          {for application use}
    listDefProc:               {list definition procedure}
        Handle;
    userHandle: Handle;        {for application use}
    dataBounds: Rect;         {boundary of cells allocated}
    cells: DataHandle;        {cell data}
    maxIndex: Integer;        {used internally}
    cellArray:                 {offsets to data}
        ARRAY[1..1] OF Integer;
END;
    ListPtr = ^ListRec;        {pointer to a list record}
    ListHandle = ^ListPtr;     {handle to a list record}

```

The only fields of a list record that you need to be familiar with are the `rView`, `port`, `cellSize`, `visible`, and `dataBounds` fields.

The `rView` field specifies the rectangle in which the list's visible rectangle is located, in local coordinates of the graphics port specified by the `port` field. Note that the list's visible rectangle does not include the area needed for the list's scroll bars. The width of a vertical scroll bar (which equals the height of a horizontal scroll bar) is 15 pixels.

The `cellSize` field specifies the size in pixels of each cell in the list. Usually, you let the List Manager automatically calculate the dimensions of a cell. It determines the default vertical size of a cell by adding the ascent, descent, and leading of the port's font. (This is 16 pixels for 12-point Chicago, for example.) For best results, you should make the height of your application's list equal to a multiple of this height. The List Manager determines the default horizontal size of a cell by dividing the width of the list's visible rectangle by the number of columns in the list.

The `visible` field specifies which cells in a list are visible within the area specified by the `rView` field. The List Manager sets the `left` and `top` fields of `visible` to the coordinates of the first visible cell; however, the List Manager sets the `right` and `bottom` fields so that each is 1 greater than the horizontal and vertical coordinates of the last visible cell. For example, if a list contains 4 columns and 10 rows but only the first 2 columns and the first 5 rows are visible (that is, the last visible cell has coordinates (1,4)), the List Manager sets the `visible` field to (0,0,2,5).

The List Manager sets the `visible` field using this method so that you can test whether a cell is visible within a list by calling QuickDraw's `PtInRect` function with a given cell and the contents of this field. Also, this allows your application to compute the number of visible rows, for example, by subtracting the `top` field of `visible` from `bottom`.

List Manager

The `dataBounds` field (located near the end of the list record) specifies the total cell dimensions of the list, including cells that are not visible. It works much like the `visible` field; that is, its `right` and `bottom` fields are each 1 greater than the horizontal and vertical coordinates of the last cell in the list. For example, if a list contains 4 columns and 10 rows (that is, the last cell in the list has coordinates (3,9)), the List Manager sets the `dataBounds` field to (0,0,4,10).

Your application seldom needs to access the remaining fields of the list record, although they are described here for your quick reference.

The `indent` field indicates the location, relative to the top-left corner of a cell, at which drawing should begin. For example, the default list definition procedure sets the vertical coordinate of this field to near the bottom of the cell, so that characters drawn with QuickDraw's `DrawText` procedure appear in the cell.

The `vScroll` and `hScroll` fields are handles to the vertical and horizontal scroll bars associated with a list. You can determine which scroll bars a list contains by checking whether these fields are `NIL`.

The `lActive` field is `TRUE` if a list is active or `FALSE` if it is inactive. You should not change the value in this field directly, but should use the `LActivate` procedure to activate or inactivate a list.

The `selFlags` field specifies the algorithm that the List Manager uses to select cells in response to a click in a list. This field is described in more detail in "Customizing Cell Highlighting" on page 4-38.

The `listFlags` field indicates whether automatic vertical and horizontal scrolling is enabled. If automatic scrolling is enabled, then a list scrolls when the user clicks a cell but then drags the cursor out of the rectangle specified by the `rView` field. For example, if a user drags the cursor below this field, the list scrolls downward. By default, the List Manager enables vertical automatic scrolling if your list has a vertical scroll bar; it enables horizontal scrolling if your list has a horizontal scroll bar. Your application can disable or enable automatic scrolling by using the following bit values:

```
CONST lDoVAutoScroll = 2; {allows vertical autoscrolling}
      lDoHAutoScroll = 1; {allows horizontal autoscrolling}
```

The `clikTime` and `clikLoc` fields indicate the time at which the user last clicked the mouse and the local coordinates of the click. The `lastClick` field (located later in the list record) indicates the cell coordinates of the last click. You can access the value in the `lastClick` field via the `LLastClick` function. If your application depends on the accuracy of the values in these fields, and if your application treats keyboard selection of list items identically to mouse selection of list items, then it should update the values of these fields after highlighting a cell in response to a keyboard event. (In particular, this is necessary if your application implements the anchor algorithm for extending cell selections with the arrow keys.)

List Manager

The `mouseLoc` field indicates the current location of the cursor in local coordinates (v, h). Ordinarily, you should use the Event Manager's `GetMouse` procedure to obtain this information, but this field may be more convenient to access from within a click-loop procedure (explained next).

The `lClickLoop` field usually contains `NIL`, but your application may place a pointer to a custom click-loop procedure in this field. A click-loop procedure manages the selection of list items and the scrolling of a list in response to a mouse click in the visible rectangle of a list. It is unlikely that your application will need to define its own click-loop procedures, because the List Manager's `LClick` function provides a default click-loop procedure that uses a robust algorithm to respond to mouse clicks. Your application needs to use a custom click-loop procedure only if it needs to perform some special processing while the user drags the cursor after clicking in a list. For more information on click-loop procedures, see "Click-Loop Procedures" on page 4-100.

The `refCon` and `userHandle` fields are for your application's use. You might, for example, use the `refCon` field to store the value of the A5 register, or to keep track of whether a list should be outlined. Typically, an application uses the `userHandle` field to store a handle to some additional storage associated with a list, but you can use the field in any way that is convenient for your application.

The `listDefProc` field contains a handle to the code used by the list definition procedure.

The `cells` field contains a handle to data that stores the list contents. The handle is declared like this:

```
TYPE  dataArray      = PACKED ARRAY[0..32000] OF Char;
      dataPtr        = ^dataArray;
      dataHandle     = ^dataPtr;
```

Because of the way the `cells` field is defined, no list can contain more than 32,000 bytes of data. The List Manager slows down considerably when a list approaches this size, and the List Manager may fail if you attempt to store more data than this in a list.

The List Manager uses the `cellArray` field to store offsets to data in the relocatable block specified by the `cells` field.

Your application will never need to access the `lReserved` and `maxIndex` fields.

S WARNING

Your application should not change the `cells` field directly or access the information in the `cellArray` field directly. The List Manager provides routines that you can use to manipulate the information in a list. s

Using the List Manager

This section explains how you can take advantage of the List Manager's features and how you can customize lists that your application creates by providing support for features not built into the List Manager. In particular, this section explains how you can

- n use the `LNew` function and the `LDispose` procedure to create a list within a rectangle in a window and then dispose of that list
- n add rows and columns to a list by using the `LAddColumn` and `LAddRow` functions along with the `LSetCell` procedure; delete them by using the `LDelColumn` and `LDelRow` procedures; and temporarily disable drawing of a list while adding multiple columns or rows by using the `LSetDrawingMode` procedure
- n call the `LClick` function to let the List Manager automatically respond to mouse clicks in a list by scrolling the list and changing the selection as appropriate; and call the `LUpdate` and `LActivate` procedures to respond to update and activate events
- n use the `LGetSelect` function and the `LSetSelect` procedure to get information about which cells are selected or to change the selection; and use the `LAutoScroll` and `LScroll` procedures to scroll to a particular cell
- n customize the algorithm that the List Manager uses to highlight cells in response to a mouse click by modifying the `selFlags` field of the list record
- n manipulate list items by using the `LAddToCell`, `LClrCell`, `LGetCellDataLocation`, and `LGetCell` procedures
- n search through a list for a particular item by using the `LSearch` function and writing a custom match function
- n respond to arrow-key and other key-down events to change or extend the selection
- n manage multiple lists within the same window or dialog box by drawing an outline around the list that would be affected by keyboard input using the `refCon` field of the list record to link the lists
- n write your own list definition procedure

Creating a List

To create a list, you can use the `LNew` function. Listing 4-1 shows a typical use of the `LNew` function to create a vertically scrolling list in a rectangular space in a window.

Listing 4-1 Creating a list with a vertical scroll bar

```

FUNCTION MyCreateVerticallyScrollingList
    (myWindow: WindowPtr; myRect: Rect;
     columnsInList: Integer;
     myLDEF: Integer): ListHandle;

CONST
    kDoDraw          = TRUE;    {always draw list after changes}
    kNoGrow          = FALSE;   {don't leave room for size box}
    kIncludeScrollBar = TRUE;   {leave room for scroll bar}
    kScrollBarWidth  = 15;     {width of vertical scroll bar}

VAR
    myDataBounds:    Rect;      {initial dimensions of the list}
    myCellSize:     Point;     {size of each cell in list}

BEGIN
    {specify dimensions of the list}
    {start with a list that contains no rows}
    SetRect(myDataBounds, 0, 0, columnsInList, 0);

    {let the List Manager calculate the size of a cell}
    SetPt(myCellSize, 0, 0);

    {adjust the rectangle to leave room for the scroll bar}
    myRect.right := myRect.right - kScrollBarWidth;

    {create the list}
    MyCreateVerticallyScrollingList :=
        LNew(myRect, myDataBounds, myCellSize, myLDEF, myWindow,
            kDoDraw, kNoGrow, NOT kIncludeScrollBar,
            kIncludeScrollBar);

END;

```

The `LNew` function called in the last line of Listing 4-1 takes a number of parameters that let you specify the characteristics of the list you wish to create.

List Manager

The first parameter to `LNew` sets the rectangle for the list's visible rectangle, specified in local coordinates of the window specified in the fifth parameter to `LNew`. Because this rectangular area does not include room for scroll bars, the `MyCreateVerticallyScrollingList` function adjusts the right of this rectangle to leave enough room.

The second parameter to `LNew` specifies the data bounds of the list. By setting the `topLeft` field of this rectangle to (0,0), you can use the `botRight` field to specify the number of columns and rows you want in the list. The `MyCreateVerticallyScrollingList` function initially creates a list of no rows. While your application is free to preallocate rows when creating a list, it is often easier to only preallocate columns and then add rows after creating the list, as described in the next section.

The third parameter is the size of a cell. By setting this parameter to (0,0), you let the List Manager compute the size automatically. The algorithm the List Manager uses to compute this size is given in the discussion of the `cellSize` field of the list record in "About the List Manager" beginning on page 4-22.

To specify that you wish to use the default list definition procedure, pass 0 as the fourth parameter to `LNew`. To use a custom list definition procedure, pass the resource ID of the list definition procedure. Note that the code for the appropriate list definition procedure is loaded into your application's heap; the code for the default list definition procedure is about 150 bytes in size.

In the sixth parameter to `LNew`, your application can specify whether the List Manager should initially enable the automatic drawing mode. When this mode is enabled, the List Manager always redraws the list after changes. Usually, your application should set this parameter to `TRUE`. This does not preclude your application from temporarily disabling the automatic drawing mode.

The last three parameters to `LNew` specify whether the List Manager should leave room for a size box, whether it should include a horizontal scroll bar, and whether it should include a vertical scroll bar. Note that while the List Manager draws scroll bars automatically, it does not draw the grow icon in the size box. Usually, your application can draw the grow icon by calling the Window Manager's `DrawGrowIcon` procedure.

The `LNew` function creates a list according your specifications and returns a handle to the list's list record. Your application uses the returned handle to refer to the list when using other List Manager routines.

List Manager

Lists are often used in dialog boxes. Because the Control Manager does not define a control for lists, you must define a list in a dialog item list as a user item. Listing 4-2 shows an application-defined procedure that creates a one-column, text-only list in a dialog box.

Listing 4-2 Installing a list in a dialog box

```

FUNCTION MyCreateTextListInDialog (myDialog: DialogPtr;
                                   myItemNumber: Integer)
                                   : ListHandle;

CONST
  kTextLDEF = 0;                {resource ID of default LDEF}
VAR
  myUserItemRect:   Rect;       {enclosure of user item}
  myUserItemType:   Integer;    {for GetDialogItem}
  myUserItemHdl:    Handle;     {for GetDialogItem}
BEGIN
  GetDialogItem(myDialog, myItemNumber, myUserItemType,
                myUserItemHdl, myUserItemRect);
  MyCreateTextListInDialog :=
    MyCreateVerticallyScrollingList(myDialog, myUserItemRect,
                                     1, kTextLDEF);
END;

```

The `MyCreateTextListInDialog` function defined in Listing 4-2 calls the `MyCreateVerticallyScrollingList` function defined in Listing 4-1, after finding the rectangle in which to install the new list by using the Dialog Manager's `GetDialogItem` procedure. For more information on the Dialog Manager, see *Inside Macintosh: Macintosh Toolbox Essentials*.

List Manager

The List Manager does not automatically draw a 1-pixel border around a list. Listing 4-3 shows an application-defined procedure that draws a border around a list.

Listing 4-3 Drawing a border around a list

```
PROCEDURE MyDrawListBorder (myList: ListHandle);
VAR
  myBorder:      Rect;           {box for list}
  myPenState:    PenState;      {current status of pen}
BEGIN
  myBorder := myList^.rView;    {get view rectangle}
  GetPenState(myPenState);      {store pen state}
  PenSize(1, 1);               {set pen to 1 pixel}
  InsetRect(myBorder, -1, -1);  {adjust rectangle for framing}
  FrameRect(myBorder);         {draw border}
  SetPenState(myPenState);     {restore old pen state}
END;
```

The `MyDrawListBorder` procedure defined in Listing 4-3 uses standard `QuickDraw` routines to save the state of the pen, set the pen size to 1 pixel, draw the border, and restore the pen state.

When you are finished using a list, you should dispose of it using the `LDispose` procedure, passing a handle to the list as the only parameter. The `LDispose` procedure disposes of the list record, as well as the data associated with the list; however, it does not dispose of any application-specific data that you might have stored in a relocatable block specified by the `userHandle` field of the list record. Thus, if you use this field to store a handle to a relocatable block, you should dispose of the relocatable block before calling `LDispose`.

Adding Rows and Columns to a List

Your application can choose to preallocate the cells it needs when it creates a list. For example, an application might preallocate the columns it needs, and then add rows to the list one by one. Other applications might create a list and add both rows and columns to it later. Regardless of the technique your application uses to create its cells, it can set the data in a cell by using the `LSetCell` procedure.

You specify the data, the length of the data, the location of the cell whose data you wish to set, and a handle to the list containing the cell, as parameters to the `LSetCell` procedure. Listing 4-4 demonstrates an application-defined procedure that adds rows to a one-column list based on the contents of a string list resource. The `MyAddItemsFromStringList` procedure adds each row to the list using the `LAddRow` function, then sets the data of the cell in the first (and only) column of the newly added row using the `LSetCell` procedure.

Listing 4-4 Adding items from a string list to a one-column, text-only list

```

PROCEDURE MyAddItemsFromStringList (myList: ListHandle;
                                     stringListID: Integer);
VAR
  index:      Integer;           {index within string list}
  rowNum:    Integer;           {row number to add string to}
  myString:  Str255;            {string to add}
  aCell:     Cell;              {cell to store string in}
BEGIN
                                     {compute new row number}
  rowNum := myList^.dataBounds.bottom;
  index := 1;                       {start with first string}
  REPEAT
    GetIndString(myString, stringListID, index);
    IF myString <> '' THEN
      BEGIN {add new row for string}
        {specify #rows to add, row number of first new row}
        rowNum := LAddRow(1, rowNum, myList);
        {prepare to set cell data--specify }
        { the cell's column number, row number}
        SetPt(aCell, 0, rowNum);
        {set cell data to string}
        LSetCell(@myString[1], Length(myString), aCell,
                  myList);
      END;
      rowNum := rowNum + 1;
      index := index + 1;
    UNTIL myString = '';
  END;

```

The `MyAddItemsFromStringList` procedure defined in Listing 4-4 adds strings from a string list resource to the end of a list. It keeps track of the index of the string in the string list with the `index` variable, and it tracks the number of the new row to add in the `rowNum` variable.

The `MyAddItemsFromStringList` procedure adds a new row by calling the `LAddRow` function. The first parameter to `LAddRow` specifies the number of rows to add, and the second parameter specifies the row number of the first new row. `LAddRow` returns the row number of the first row added, which differs from the second parameter only if that parameter specifies a row number that is out of range.

After creating a new row, `MyAddItemsFromStringList` sets the cell in the first column of the added row to the text contained within the string. Note that the procedure does not copy the length byte of the string.

List Manager

To add columns to a list, your application can use the `LAddColumn` function, which works just like `LAddRow`.

To delete a row or column from a list, your application can call the `LDelRow` procedure or the `LDelColumn` procedure. The first parameter of each of these procedures is the number of rows (or columns) to delete, and the second parameter is the row or column number of the first to be deleted. For example, this code deletes the first row of a list:

```
LDelRow(1, 0, myList); {#rows to delete, starting row number}
```

When making many changes to a list, your application should temporarily disable the automatic drawing mode (unless the list is in a window that is not yet visible). To do so, call the `LSetDrawingMode` procedure to turn off the automatic drawing mode, make the changes to the list, turn the automatic drawing mode back on, and redraw the list (by invalidating a rectangle containing the list and its scroll bars and later calling the `LUpdate` procedure when your application receives an update event). You might do these steps as follows:

```
LSetDrawingMode (FALSE, myList);
{...(make changes to the list)...}
LSetDrawingMode (TRUE, myList);
InvalRect(myList^.rView);
IF (myList^.vScroll <> NIL) THEN
    InvalRect(myList^.vScroll^.contrlRect);
IF (myList^.hScroll <> NIL) THEN
    InvalRect(myList^.hScroll^.contrlRect);
```

Responding to Events Affecting a List

Your application must respond to several different types of events involving a list by calling appropriate List Manager routines. If a mouse-down event occurs in a list, your application should call the `LClick` function. If your application receives an update event, and some part of the list is within the update region, then it should call the `LUpdate` procedure. If a window containing a list is activated or deactivated, your application should activate or deactivate the list by calling the `LActivate` procedure. Finally, if a key-down event occurs, your application may need to call its own internal procedures to scroll the list or select items as necessary. This section explains how to handle mouse-down, update, and activate events; for information on handling key-down events, see “Supporting Keyboard Navigation of Lists” on page 4-45.

List Manager

The `LClick` function automatically responds to a mouse-down event by handling user interaction until the user releases the mouse button. The List Manager performs any scrolling as necessary and changes the selection as appropriate. After handling the event, the `LClick` function returns `TRUE` if the click was a double click. Listing 4-5 shows an application-defined procedure that uses the `LClick` function to handle mouse-down events in a list.

Listing 4-5 Responding to a mouse-down event in a list

```
PROCEDURE MyHandleMouseDownInList (theEvent: EventRecord;
                                   theList: ListHandle);

BEGIN
    SetPort(theList^^.port);
    GlobalToLocal(theEvent.where);
    IF LClick(theEvent.where, theEvent.modifiers, theList) THEN
        MyDoubleClick(theList);
    END;
```

In response to a double click, your application should simulate the selection of the default button if there is one. If your dialog box does not contain a default button, then your application can respond to a double click with some other appropriate behavior.

Listing 4-6 illustrates an application-defined procedure that responds to an update event affecting a list.

Listing 4-6 Responding to an update event in a list

```
PROCEDURE MyUpdateList (theList: ListHandle);
BEGIN
    SetPort(theList^^.port);           {set up the drawing environment}
                                       {update list and scroll bars}
    LUpdate(theList^^.port^.visRgn, theList);
    MyDrawListBorder(theList);        {draw border around list}
END;
```

List Manager

Your list update procedure might also do some other drawing appropriate to a particular list. For example, if your application supports multiple lists in a window, then your list-updating procedure should redraw an outline around the current list in response to an update event. For more information on outlining the current list, see “Outlining the Current List” on page 4-53.

Note that the call to the `LUpdate` procedure must be bracketed by calls to the Window Manager’s `BeginUpdate` and `EndUpdate` procedures. See the chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for more information.

In response to an activate event, your application should call `LActivate` for each list in the window. For example, this code deactivates a list:

```
LActivate (FALSE, myList);
```

To activate a list, pass `TRUE` as the first parameter to `LActivate`.

Working With List Selections

The List Manager provides routines that make it easy to determine what the selection is or to change the selection, whether your list allows just one item to be selected at a time or allows many items to be selected. The List Manager also provides a routine that allows you to automatically scroll the first selected cell to the upper-left corner of the list’s visible rectangle. In addition, you can write your own routine to scroll a list just enough so that a particular cell is visible.

Your application can use the `LGetSelect` function to determine whether a given cell is selected or to find the next selected cell. Your application can use the `LSetSelect` procedure to select or deselect a given cell.

Listing 4-7 shows an application-defined procedure that finds the first cell in a selection.

Listing 4-7 Finding the first selected cell in a list

```
FUNCTION MyGetFirstSelectedCell (theList: ListHandle;
                                VAR theCell: Cell): Boolean;
BEGIN
    SetPt(theCell, 0, 0);
    MyGetFirstSelectedCell := LGetSelect(TRUE, theCell, theList);
END;
```

List Manager

The first parameter (`TRUE`) passed to the `LGetSelect` function indicates that `LGetSelect` should search the list (beginning with the cell specified in the second parameter) for the first selected cell. If you pass `TRUE` as the first parameter, `LGetSelect` sets the cell specified in the second parameter to the coordinates of the first selected cell that it finds, or it returns `FALSE` if no cells including or after the cell specified by the second parameter are selected. If you pass `FALSE` as the first parameter to `LGetSelect`, then the function returns `TRUE` only if the cell specified in the second parameter is selected. The `MyGetFirstSelectedCell` function defined in Listing 4-7 thus returns `TRUE` only if at least one cell is selected, in which case the second parameter to the function is set to the coordinates of that cell.

Finding the last selected cell in a list is slightly more complex. Listing 4-8 illustrates how this might be done.

Listing 4-8 Finding the last selected cell in a list

```
PROCEDURE MyGetLastSelectedCell (theList: ListHandle;
                                VAR theCell: Cell);
VAR
    aCell:           Cell;
    moreCellsInList: Boolean;
BEGIN
    IF MyGetFirstSelectedCell(theList, aCell) THEN
        REPEAT
            theCell := aCell;
            moreCellsInList := LNextCell(TRUE, TRUE, aCell, theList);
        UNTIL NOT LGetSelect(TRUE, aCell, theList);
    END;
```

The `MyGetLastSelectedCell` procedure goes from one selected cell to the next until there are no more selected cells. It calls the `LNextCell` function to move from one cell to the next cell in the list. If it did not do this, then the procedure would loop infinitely, since `LGetSelect` would repeatedly return `TRUE` for the first selected cell. The first two parameters to `LNextCell` indicate whether the function should return the next cell in the current row, the next cell in the current column, or, if both are set to `TRUE`, the next cell regardless of location.

List Manager

Your application can use the `LSetSelect` procedure to set or deselect a cell by passing `TRUE` or `FALSE`, respectively, as the first parameter to the routine. Listing 4-9 illustrates a useful procedure that uses `LSetSelect` and `LGetSelect` to select a single cell in a list while deselecting all other cells.

Listing 4-9 Selecting a cell and deselecting other cells

```
PROCEDURE MySelectOneCell (theList: ListHandle; theCell: Cell);
VAR
  nextSelectedCell:   Cell;
  moreCellsInList:   Boolean;
BEGIN
  IF MyGetFirstSelectedCell(theList, nextSelectedCell) THEN
    WHILE LGetSelect(TRUE, nextSelectedCell, theList) DO
      BEGIN
        {move to next selected cell...}
        IF (nextSelectedCell.h <> theCell.h) OR
           (nextSelectedCell.v <> theCell.v) THEN
          {...and remove cell from selection}
          LSetSelect(FALSE, nextSelectedCell, theList)
        ELSE
          moreCellsInList :=
            {move to next cell}
            LNextCell(TRUE, TRUE, nextSelectedCell, theList);
      END;
      LSetSelect(TRUE, theCell, theList);
    END;
  END;
```

The `MySelectOneCell` procedure defined in Listing 4-9 deselects each selected cell, except that if it encounters the cell that is ultimately to be selected, then it does not deselect that cell. This prevents an annoying flickering that would otherwise occur if you were to call `MySelectOneCell` to select a cell already selected.

The List Manager provides the `LAutoScroll` procedure to enable your application to scroll the first selected cell to the upper-left corner of the list's visible rectangle—for example:

```
LAutoScroll(myList);
```

List Manager

Sometimes, you might want your application to scroll a list just enough so that a certain cell (such as a cell the user has just selected using the keyboard) is visible. For example, this is how the Standard File Package responds if the user presses the Down Arrow key when the currently selected item is on the bottom of the list's visible rectangle. You can mimic this effect by calling the `LScroll` procedure, which requires that your application indicate how many columns and rows to scroll. Negative numbers indicate scrolling up or to the left. Positive numbers indicate scrolling down or to the right. Listing 4-10 illustrates the use of the `LScroll` procedure.

Listing 4-10 Scrolling so that a particular cell is visible

```

PROCEDURE MyMakeCellVisible (theList: ListHandle; theCell: Cell);
VAR
    visibleRect:      Rect;      {rectangle enclosing visible cells}
    dCols, dRows:    Integer; {number of rows to scroll}
BEGIN
    visibleRect := theList^.visible;
    IF NOT PtInRect(theCell, visibleRect) THEN
        BEGIN
            {cell is not already visible}
            WITH theCell, visibleRect DO
                BEGIN
                    IF h > right - 1 THEN
                        dCols := h - right + 1      {move to left}
                    ELSE IF h < left THEN
                        dCols := h - left;          {move to right}
                    IF v > bottom - 1 THEN
                        dRows := v - bottom + 1     {move up}
                    ELSE IF v < top THEN
                        dRows := v - top;           {move down}
                END;
                LScroll(dCols, dRows, theList);
            END;
        END;
    END;
END;

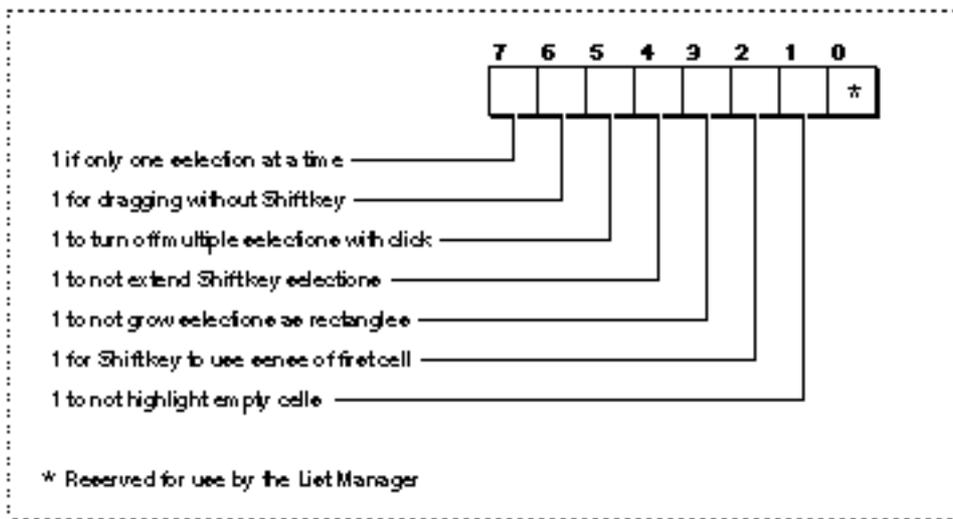
```

The `MyMakeCellVisible` procedure defined in Listing 4-10 simply computes the number of cells between the last visible row and column and the selected cell. Note that the last visible column for a list is equal to `theList^.visible.right - 1`, and the last visible row is `theList^.visible.bottom - 1`.

Customizing Cell Highlighting

You can change the `selFlags` field of the list record to modify the algorithm the List Manager uses to select cells in response to a mouse click. “Selection of List Items” beginning on page 4-9 explains the different customizations you can make. Figure 4-18 illustrates the bits in the `selFlags` field.

Figure 4-18 Selection flags



The List Manager defines constants for each flag:

```
CONST
    lOnlyOne      = -128;  {allow only 1 item to be selected at once}
    lExtendDrag  = 64;    {enable selection of multiple items }
                    { by dragging without Shift}
    lNoDisjoint  = 32;    {prevent discontinuous selections }
                    { using Command key}
    lNoExtend    = 16;    {deselect all items before }
                    { responding to Shift-click}
    lNoRect      = 8;     {select all items in cursor's path }
                    { during Shift-drag}
    lUseSense    = 4;     {allow user to use Shift key to }
                    { deselect one or more items}
    lNoNilHilite= 2;     {disable highlighting of empty cells}
```

List Manager

When you create a list, the List Manager clears all bits in the `selFlags` fields. To change any of these defaults, set the appropriate bits in the `selFlags` field. For example, this code sets the `selFlags` field so that only one selection is allowed in a list:

```
myList^^.selFlags := lOnlyOne;
```

Many of the constants are often used additively. For example, your application might allow the user to select a new range of cells simply by dragging over them, as shown in the following code:

```
myList^^.selFlags := lExtendDrag + lNoDisjoint + lNoExtend
                    + lNoRect + lUseSense;
```

The `lExtendDrag` constant allows users to select a range of items simply by dragging the cursor. Ordinarily, if the user clicks one cell and drags the cursor to another, only the last cell remains set.

The `lNoDisjoint` constant ensures that only one range of cells can be selected.

The `lNoExtend` constant disables the List Manager feature that responds to a Shift-click by selecting all cells in the range of the newly clicked cell and the first (or last) selected cell. Instead, the List Manager simply deselects all cells in the range if this bit is set.

To allow the user to select a number of cells simply by moving the cursor over them, you can set the bit corresponding to the `lNoRect` constant. This prevents the deselection of cells should the user drag the cursor first in one direction and then the other.

You can set the bit corresponding to the `lUseSense` constant so that if a user Shift-clicks a selected cell, the cell is deselected. Ordinarily, Shift-clicking a selected cell has no effect.

You might also wish to make the Shift key work just like the Command key in your application. You can accomplish that with the following code:

```
myList^^.selFlags := lNoRect + lNoExtend + lUseSense;
```

The `lNoNilHilite` constant is somewhat different from the others, in that it affects the display of a list, not the way that the List Manager selects items in response to a click. If the bit corresponding to this constant is set, then the List Manager does not select or highlight cells that do not contain any data.

Manipulating List Cells

In addition to the `LSetCell` procedure, the List Manager provides four procedures, `LAddToCell`, `LClrCell`, `LGetCellDataLocation`, and `LGetCell`, that allow you to manipulate cell item data. You can use the `LAddToCell` procedure to append data to list items and the `LClrCell` procedure to remove all data from a list item. The `LGetCellDataLocation` procedure indicates the location of the beginning of a cell's data within the `cells` field of the list record as well as the length of that data, and the `LGetCell` procedure copies a cell's data to a buffer that your application specifies.

Listing 4-11 illustrates the use of `LClrCell` to clear the data from all cells in a list.

Listing 4-11 Clearing all cell data

```
PROCEDURE MyClearAllCellData (myList: ListHandle);
VAR
    aCell: Cell;
BEGIN
    SetPt(aCell, 0, 0);
    REPEAT
        LClrCell(aCell, myList);
    UNTIL NOT LNextCell(TRUE, TRUE, aCell, myList);
END;
```

Because `LClrCell` simply does nothing if passed a cell not in the list, the `MyClearAllCellData` procedure defined in Listing 4-11 will not crash when attempting to clear the first cell even if there are no cells in the list.

Listing 4-12 uses the `LGetCell` procedure to return the data of a specific cell.

Listing 4-12 Getting a copy of the data of a cell

```
PROCEDURE MyGetCellData (dataPtr: Ptr; VAR dataLen: Integer;
                        aCell: Cell; myList: ListHandle);
BEGIN
    LGetCell(dataPtr, dataLen, aCell, myList);
END;
```

The `LGetCell` procedure copies cell data to memory beginning at the location specified by `dataPtr`. It copies only the number of bytes specified by the value passed in the `dataLen` parameter; it returns in that parameter the number of bytes actually copied.

Because the `LGetCell` procedure duplicates existing bytes of memory, if your application needs to access a cell's data but does not need to manipulate the data, then it should use the `LGetCellDataLocation` procedure to access cell data directly. Listing 4-13 uses the `LGetCellDataLocation` procedure to get a cell's data.

Listing 4-13 Directly accessing a cell's data

```
PROCEDURE MyGetDirectAccessToCellData
    (VAR offset: Integer; VAR len: Integer;
     aCell: Cell; myList: ListHandle);
BEGIN
    LGetCellDataLocation(offset, len, aCell, myList);
END;
```

The `LGetCellDataLocation` procedure simply returns in the `offset` and `len` parameters the offset and length of the appropriate cell's data within the `cells` field of the list record.

Listing 4-14 shows an application-defined procedure that uses `LGetCellDataLocation` in conjunction with the `LSetCell` procedure and the `LAddRow` function to add a new string to a one-column, alphabetical text-only list. To compare two strings, the procedure uses the Text Utilities `CompareText` function, which requires that data be specified by a pointer and length, thus making `LGetCellDataLocation` perfect for this purpose. For more information on the `CompareText` function, see *Inside Macintosh: Text*.

Listing 4-14 Adding an item to a one-column, alphabetical text list

```

PROCEDURE MyAddItemAlphabetically (myList: ListHandle; myString: Str255);
VAR
    found:      Boolean;           {flag variable}
    myRows:     Integer;          {number of rows in list}
    currentRow: Integer;          {row being examined}
    cellDataOffset, cellDataLength: Integer; {data being compared to string}
    aCell:      Cell;             {cell coordinates}
BEGIN
    found := FALSE;               {initialize flag variable}
    WITH myList^.dataBounds DO
        myRows := bottom - top;   {compute number of rows}
    currentRow := -1;             {start before first row}
    WHILE NOT found DO
        BEGIN                     {try to insert before next row}
            currentRow := currentRow + 1; {move to next row}
            IF currentRow = myRows THEN {past the end of the list?}
                found := TRUE;        {insert string at this row}
            ELSE
                BEGIN
                    SetPt(aCell, 0, currentRow); {prepare to check cell data}
                                                    {find location of data}
                    LGetCellDataLocation(cellDataOffset, cellDataLength, aCell, myList);
                    HLockHi(Handle(myList^.cells)); {lock list data in memory}
                    IF CompareText(@myString[1], {skip length byte of string}
                        Ptr(ORD4(myList^.cells^) + cellDataOffset),
                        Length(myString), cellDataLength, gitl2Hdl) = -1 THEN
                        found := TRUE; {new string should precede }
                                       { this row's string}
                    HUnlock(Handle(myList^.cells)); {unlock list data}
                END;
        END;
        {add new row for string}
        currentRow := LAddRow(1, currentRow, myList);
        SetPt(aCell, 0, currentRow); {prepare to set cell data}
                                       {set data}
        LSetCell(@myString[1], Length(myString), aCell, myList);
    END;
END;

```

The `MyAddItemAlphabetically` procedure defined in Listing 4-14 simply compares a string to the text in each row of a list, until the string follows the row text alphabetically or until there are no more rows, that is, the row number (which is 0-based) is equal to the number of rows, in which case the string is appended to the end of the list.

Searching a List for a Particular Item

Sometimes, your application might need to search through a list for a particular item. For example, your application might need to search a list of pictures to see which cell contains a certain picture, or your application might wish to search for an item that matches a certain string. You can use the `LSearch` function and specify your own match function to make this possible.

The `LSearch` function returns `TRUE` if it is able to find the specified data in a cell greater than or equal to the specified cell. If it does find the data, it also returns the coordinates of the cell that contains the data.

In addition to specifying the cell to search, your application also specifies a pointer to a match function, the data to search for, and the length of the data, as parameters to the `LSearch` function.

If your application specifies `NIL` for the match function, the `LSearch` function searches the list for the first cell whose data matches the specified data. In particular, the `LSearch` function calls the Text Utilities `IUMagIDString` function to compare each cell's data with the specified data until `IUMagIDString` returns 0. Because `IUMagIDString` compares strings for equality without regard for secondary ordering, using this default match function is useful only for text-only lists. For more information on `IUMagIDString`, see *Inside Macintosh: Text*.

Your application can use a different match function from `IUMagIDString` as long as it is defined just like `IUMagIDString`. For example, your application could use the `IUMagString` function so that secondary ordering is taken into consideration. To do so, your application might use the following code:

```
found := LSearch(myData, myLength, @IUMagString, myCell, myList);
```

You can also write your own match function. Listing 4-15 shows an example match function.

Listing 4-15 A match function

```
FUNCTION MySearchPartialMatch
    (cellDataPtr, searchDataPtr: Ptr;
     cellDataLen, searchDataLen: Integer): Integer;
BEGIN
    IF (cellDataLen > 0) AND (cellDataLen >= searchDataLen) THEN
        MySearchPartialMatch :=
            IUMagIDString(cellDataPtr, searchDataPtr,
                          searchDataLen, searchDataLen)
    ELSE
        MySearchPartialMatch := 1;
    END;
```

List Manager

Your match function should return 0 if it finds a match and 1 otherwise. The match function defined in Listing 4-15 works just like the default match function but allows the cell data to be longer than the data being searched for. For example, a search for the text “rose” would match a cell containing the text “Rosebud”.

Listing 4-16 defines a more complex but potentially more useful match function for text-only lists.

Listing 4-16 Searching a list for a cell containing certain text or the next cell alphabetically

```

FUNCTION MyMatchNextAlphabetically
    (cellDataPtr, searchDataPtr: Ptr;
     cellDataLen, searchDataLen: Integer): Integer;
BEGIN
    MyMatchNextAlphabetically := 1;    {set default return value}
    IF (cellDataLen > 0) THEN
    BEGIN
        IF IUMagIDString(cellDataPtr, searchDataPtr,
                        searchDataLen, searchDataLen) = 0 THEN
            MyMatchNextAlphabetically := 0 {strings are equal}
        ELSE IF IUMagString(cellDataPtr, searchDataPtr,
                           cellDataLen, searchDataLen) = 1 THEN
            MyMatchNextAlphabetically := 0; {search data is after }
                                           { cell data}
        END;
    END;
END;

```

Using the `LSearch` function with the `MyMatchNextAlphabetically` function defined in Listing 4-16 results in the finding of the cell that is alphabetically greater than or equal to the search text. For example, if you use the `LSearch` function with this match function to search a list of the 50 states (not including the District of Columbia) for the text “Washington, D.C.”, then the `LSearch` function returns the coordinates of the cell containing the text “West Virginia”.

Note

The `MyMatchNextAlphabetically` function defined in Listing 4-16 works only for lists that are alphabetically arranged. u

Supporting Keyboard Navigation of Lists

This section discusses how your application can support keyboard navigation of lists. In particular, this section first shows how your application can respond to the user's typing to select an item in a text-only list. Second, this section shows how your application can respond to the user's pressing of the arrow keys.

Supporting Type Selection of List Items

To support type selection of list items, your application must keep a record of the characters the user has typed, the time when the user last typed a character, and which list the last typed character affected. For example, the SurfWriter application defines the following four variables to keep track of this information:

```
VAR
    gListNavigateString:      {current string being searched}
                            Str255;
    gTSThresh:               Integer;   {ticks before type selection resets}
    gLastKeyTime:            LongInt;   {time in ticks of last click time}
    gLastListHit:            ListHandle; {last list type selection affected}
```

The `gListNavigateString` variable stores the current status of the type selection. For example, if the user types 'h' and then 'e' and then 'l' and then 'l' and then 'o', this string should be 'hello'.

The `gTSThresh` variable stores the number of ticks after which type selection resets. For example, if the user has typed 'hello' but then waits more than this amount of time before typing 'g', the SurfWriter application sets `gListNavigateString` to 'g', not to 'hellog'. The value of `gTSThresh` is dependent on the value the user sets for "Delay Until Repeat" in the Keyboard control panel. SurfWriter also resets the type selection if the user begins typing in a different list from the list last typed in. Thus, if the difference between the current tick count and the `gLastKeyTime` variable is greater than `gTSThresh`, or if `gLastListHit` is not equal to the current list, then the SurfWriter application must reset the type selection.

List Manager

Listing 4-17 shows how the SurfWriter application initializes or resets its type-selection variables.

Listing 4-17 Resetting variables related to type selection

```
PROCEDURE MyResetTypeSelection;
CONST
    KeyThresh = $18E;    {location of low-memory word}
    kMaxKeyThresh = 120; {120 ticks = 2 seconds}
TYPE
    IntPtr = ^Integer;  {for accessing low memory}
BEGIN
    gListNavigateString := '';           {reset navigation string}
    gLastListHit := NIL;                 {remember active list}
    gLastKeyTime := 0;                   {no keys yet hit}
    gTSThresh := 2 * IntPtr(KeyThresh)^; {update type-selection }
                                         { threshold}

    IF gTSThresh > kMaxKeyThresh THEN
        gTSThresh := kMaxKeyThresh;     {set threshold to maximum}
END;
```

The `MyResetTypeSelection` procedure defined in Listing 4-17 initializes three of the variables to default values and sets the `gTSThresh` variable to twice the value of the system global variable `KeyThresh`, up to a maximum of 120 ticks. By using the same formula as `MyResetTypeSelection` for computing the type-selection threshold, you make sure your application is consistent with other applications as well as with the Finder. The SurfWriter application calls the `MyResetTypeSelection` procedure when it starts up and when it wishes to reset the type selection because the type-selection threshold has expired. It also calls the procedure whenever it receives a resume event, because the user might have used the Keyboard control panel, in which case SurfWriter needs to update the value of the type-selection threshold.

Having initialized variables related to type selection, the SurfWriter application needs to respond to appropriate key-down events. Listing 4-18 illustrates an application-defined procedure that does this.

Listing 4-18 Selecting an item in response to a key-down event

```

PROCEDURE MyKeySearchInList (theList: ListHandle; theEvent: EventRecord);
VAR
    newChar: Char;           {character to add to search string}
    theCell: Cell;          {cell containing found string}
BEGIN
    newChar := CHR(BAnd(theEvent.message, charCodeMask));
    IF (gLastListHit <> theList) OR
        (theEvent.when - gLastKeyTime >= gTSThresh) OR
        (Length(gListNavigateString) = 255) THEN
        MyResetTypeSelection;
        gLastListHit := theList;           {remember list keyed in}
        gLastKeyTime := theEvent.when;    {record time of key-down event}
        gListNavigateString[0] := Char(Length(gListNavigateString) + 1);
                                           {set length of string}
                                           {add character to string}
        gListNavigateString[Length(gListNavigateString)] := newChar;

        SetPt(theCell, 0, 0);
        IF LSearch(@gListNavigateString[1], Length(gListNavigateString),
            @MyMatchNextAlphabetically, theCell, theList) THEN
        BEGIN
            {deselect all cells but new cell}
            MySelectOneCell(theList, theCell);
            {make sure new selection is visible}
            MyMakeCellVisible(theList, theCell);
        END;
    END;
END;

```

List Manager

The `MyKeySearchInList` procedure defined in Listing 4-18 first updates variables related to type selection. Then it searches through the list for a cell containing the current search string or for the next cell alphabetically. It searches using the `LSearch` function in conjunction with a custom match function defined in Listing 4-15 on page 4-43. The procedure also uses the `MySelectOneCell` procedure defined in Listing 4-9 on page 4-36 and the `MyMakeCellVisible` procedure defined in Listing 4-10 on page 4-37.

Note

If your compiler enforces range checking, you may need to disable it before using the code in Listing 4-18, because the code accesses the length byte of a string directly. See your development system's documentation for more information on range checking. `u`

Supporting Arrow-Key Navigation of Lists

This section discusses how your application can support the use of arrow keys to move the current selection or to extend the current selection using a simple extension algorithm. For information on implementing a more complex anchor algorithm for extending the selection, read this section and then the next section, beginning on page 4-52.

The following constants define the ASCII character codes for the various arrow keys. These ASCII values for these keys are the same for U.S. and international keyboards.

```
CONST
    kLeftArrow    = Char(28);           {move left}
    kRightArrow   = Char(29);           {move right}
    kUpArrow      = Char(30);           {move up}
    kDownArrow    = Char(31);           {move down}
```

To support both the moving of a selection (the user's pressing an arrow key without pressing the Shift key) and the extending of a selection (the user's pressing of an arrow key while pressing the Shift key), your application needs to define a routine that computes a new selection location given an old one. For example, if the user presses Command-Left Arrow, the routine should find the cell as far to the left of the first currently selected cell as possible. Listing 4-19 illustrates an application-defined procedure that does this.

Listing 4-19 Determining the location of a new cell in response to an arrow-key event

```

PROCEDURE MyFindNewCellLoc
    (theList: ListHandle; oldCellLoc: Cell;
     VAR newCellLoc: Cell; keyHit: Char;
     moveToExtreme: Boolean);
VAR
    listRows, listColumns: Integer;      {list dimensions}
BEGIN
    WITH theList^^.dataBounds DO
    BEGIN
        listRows := bottom - top;        {number of rows in list}
        listColumns := right - left;     {number of columns in list}
    END;
    newCellLoc := oldCellLoc;
    IF moveToExtreme THEN
        CASE keyHit OF
            kUpArrow:
                newCellLoc.v := 0;        {move to row 0}
            kDownArrow:
                newCellLoc.v := listRows - 1; {move to last row}
            kLeftArrow:
                newCellLoc.h := 0;        {move to column 0}
            kRightArrow:
                newCellLoc.h := listColumns - 1; {move to last column}
        END
    ELSE
        CASE keyHit OF
            kUpArrow:
                IF oldCellLoc.v <> 0 THEN
                    newCellLoc.v := oldCellLoc.v - 1; {row up}
            kDownArrow:
                IF oldCellLoc.v <> listRows - 1 THEN
                    newCellLoc.v := oldCellLoc.v + 1; {row down}
            kLeftArrow:
                IF oldCellLoc.h <> 0 THEN
                    newCellLoc.h := oldCellLoc.h - 1; {column left}
            kRightArrow:
                IF oldCellLoc.h <> listColumns - 1 THEN
                    newCellLoc.h := oldCellLoc.h + 1; {column right}
        END;
    END;
END;

```

List Manager

The `MyFindNewCellLoc` procedure defined in Listing 4-19 computes the coordinates of the cell referenced by the `newCellLoc` parameter based on the coordinates of the `oldCellLoc` parameter and the direction of the arrow key pressed. The `oldCellLoc` parameter contains the coordinates of the first or last cell in a selection, depending on which arrow key was pressed. The behavior of `MyFindNewCellLoc` also depends on the value passed in the `moveToExtreme` parameter. For example, if the user pressed the Command key while pressing an arrow key, the SurfWriter application passes `TRUE`; otherwise, it passes `FALSE`. If `moveToExtreme` is `TRUE`, then `MyFindNewCellLoc` returns in `newCellLoc` a cell that is as far as possible from the cell specified in `oldCellLoc`. Otherwise, it returns a cell that is within one cell of `oldCellLoc`. If a cell cannot be moved in the direction specified by the arrow key, `newCellLoc` is equivalent on exit to `oldCellLoc`.

Having defined the `MyFindNewCellLoc` procedure, it is easy to move or extend a selection in response to an arrow-key event. Listing 4-20 illustrates an application-defined procedure that moves the selection in response to the user's pressing an arrow key without pressing the Shift key.

Listing 4-20 Moving the selection in response to an arrow-key event

```
PROCEDURE MyArrowKeyMoveSelection (theList: ListHandle;
                                   keyHit: Char;
                                   moveToExtreme: Boolean);

VAR
  currentSelection:    Cell;
  newSelection:       Cell;
BEGIN
  IF MyGetFirstSelectedCell(theList, currentSelection) THEN
    BEGIN
      IF (keyHit = kRightArrow) OR (keyHit = kDownArrow) THEN
        {find last selected cell}
        MyGetLastSelectedCell(theList, currentSelection);
        {move relative to appropriate cell}
        MyFindNewCellLoc(theList, currentSelection,
                          newSelection, keyHit, moveToExtreme);
        {make this cell the selection}
        MySelectOneCell(theList, newSelection);
        {make sure new selection is visible}
        MyMakeCellVisible(theList, newSelection);
      END;
    END;
END;
```

List Manager

The `MyArrowKeyMoveSelection` procedure defined in Listing 4-20 calls the `MyFindNewCellLoc` procedure defined in Listing 4-19 to find the coordinates of a cell to select. It computes the coordinates of that new cell relative to the first selected cell if the user pressed a Left Arrow or Up Arrow key; otherwise, it computes the coordinates of the new cell relative to the last selected cell. After computing the coordinates of the new cell, `MyArrowKeyMoveSelection` selects it by calling routines defined in Listing 4-9 and Listing 4-10.

Listing 4-21 illustrates an application-defined procedure that extends the selection in response to the user's pressing an arrow key while pressing the Shift key.

Listing 4-21 Extending the selection in response to an arrow-key event

```
PROCEDURE MyArrowKeyExtendSelection (theList: ListHandle;
                                     keyHit: Char;
                                     moveToExtreme: Boolean);

VAR
    currentSelection: Cell;
    newSelection:     Cell;
BEGIN
    IF MyGetFirstSelectedCell(theList, currentSelection) THEN
        BEGIN
            IF (keyHit = kRightArrow) OR (keyHit = kDownArrow) THEN
                {find last selected cell}
                MyGetLastSelectedCell(theList, currentSelection);
                {move relative to appropriate cell}
                MyFindNewCellLoc(theList, currentSelection,
                                newSelection, keyHit, moveToExtreme);
                {add a new cell to the selection}
            IF NOT LGetSelect(FALSE, newSelection, theList) THEN
                LSetSelect(TRUE, newSelection, theList);
                {make sure new selection is visible}
                MyMakeCellVisible(theList, newSelection);
            END;
        END;
END;
```

The `MyArrowKeyExtendSelection` procedure defined in Listing 4-21 works just like the `MyArrowKeyMoveSelection` procedure defined in Listing 4-20, but it does not deselect all other cells besides the newly selected cell.

Listing 4-22 shows an application-defined procedure that takes advantage of the code listings provided in this section. The SurfWriter application calls the procedure in Listing 4-22 every time it receives an arrow-key event that affects a list.

Listing 4-22 Processing an arrow-key event

```
PROCEDURE MyArrowKeyInList (theList: ListHandle; theEvent: EventRecord;
                           allowExtendedSelections: Boolean);
BEGIN
  IF (NOT allowExtendedSelections) OR
     (BAnd(theEvent.modifiers, shiftKey) = 0) THEN
    MyArrowKeyMoveSelection(theList,
                           CHR(BAnd(theEvent.message, charCodeMask)),
                           BAnd(theEvent.modifiers, cmdKey) <> 0)
  ELSE
    MyArrowKeyExtendSelection(theList,
                              CHR(BAnd(theEvent.message, charCodeMask)),
                              BAnd(theEvent.modifiers, cmdKey) <> 0);
END;
```

The `MyArrowKeyInList` procedure defined in Listing 4-22 takes three parameters, the third of which is a Boolean variable that indicates whether the application supports the use of Shift–arrow key combinations to extend the current selection. If the application does support this and the user held down the Shift key, the `MyArrowKeyInList` procedure calls the procedure in Listing 4-21 to extend the selection. Otherwise, it calls the procedure in Listing 4-20 to move the selection. Either way, it checks the status of the Command key to determine whether the appropriate procedure should move as far in the direction of the arrow key as possible before selecting a new cell.

Supporting the Anchor Algorithm for Extending Lists With Arrow Keys

This section summarizes how your application can support the anchor method for extending lists with arrow keys. Implementing this method takes a lot of work, but the extra work may pay off if you expect many users of your application's lists to make range selections or if your application uses multicolumn lists. For a comparison between the anchor algorithm and the extension algorithm illustrated in the previous section, see "Extension of a Selection With Arrow Keys" on page 4-16.

To support the anchor algorithm, your application must keep track of several types of information between Shift–arrow key events. Most importantly, your application must store information about which cell in a list is the anchor cell and which cell is the moving cell. In response to a Shift–arrow key event, your application should change the location of the moving cell. It should then highlight all cells in the rectangle whose corners are

List Manager

the anchor cell and the moving cell. This permits the user to use several consecutive Shift-arrow key combinations to move a rectangular range of cells around the anchor cell.

Your application must thus save the location of the anchor cell the first time the user uses a Shift-arrow key combination to affect a certain rectangular range of cells. For example, if the user presses Shift-Right Arrow and the user has not before used a Shift-arrow key combination, then your application should store as the anchor cell the upper-left cell in the rectangular range of cells to be affected. The moving cell is then one cell to the right of what was the lower-right corner of this range.

Your application can determine what rectangular range of cells a Shift-arrow key combination is meant to affect by using the `LLastClick` function, which returns the coordinates of the last cell that was clicked. (If your application relies on this function, it must always update the `lastClick` field of the list record in response to keyboard selection of any list item, since keyboard selection of a list item is functionally equivalent to clicking.) Your application must check the selection status of adjacent cells to find as big a rectangular range of selected cells surrounding this cell as possible.

Your application can check whether a Shift-arrow key event is affecting a new range of cells simply by checking the `clickTime` field of the list record. (Your application must thus also update this field in response to keyboard selection of any list item.) If the last click time changes between Shift-arrow key events, your application knows that the user has clicked the list or used the keyboard to change the selection. In this case, your application must compute a new anchor cell and moving cell based on the `LLastClick` function and the direction of the arrow key pressed. Otherwise, your application can keep the same anchor cell, move the moving cell in the direction specified by the arrow key, and highlight cells in the rectangular range of the anchor cell and the moving cell.

In summary, if your application is to support the anchor algorithm for extending a list selection, it must keep track of an anchor cell, a moving cell, and the time of the last click in a list. (Your application might store a handle to a relocatable block containing this information in the `userHandle` field of the list record.) Whenever a Shift-arrow key event is meant to affect a new range of cells, your application updates all three of these variables. Otherwise, it only changes the coordinates of the moving cell from one Shift-arrow key event to the next.

Outlining the Current List

If a window in your application contains two lists, or contains one list and an editable text item, then your application should place a 2-pixel outline around a list whenever the list is the current list and active, that is, whenever typing would affect the list. Your application should outline the current list so that the user knows that typing affects the list.

List Manager

Listing 4-23 shows an application-defined procedure that checks whether a list is the current list. If it is both current and active, it draws a 2-pixel outline around the list. Otherwise, it draws in the background color of the dialog box to remove the outline.

Listing 4-23 Drawing an outline around a list

```

PROCEDURE MyDrawOutline (myList: ListHandle);
CONST
    kScrollBarWidth = 15;           {width of scroll bar}
VAR
    myOutlineRect:   Rect;         {rectangle for outline border}
    myPenState:     PenState;     {current status of pen}
BEGIN
    {get list's visible rectangle}
    myOutlineRect := myList^.rView;
    {compensate for scroll bars}
    IF myList^.vScroll <> NIL THEN
        myOutlineRect.right := myOutlineRect.right
                               + kScrollBarWidth;
    IF myList^.hScroll <> NIL THEN
        myOutlineRect.bottom := myOutlineRect.bottom
                               + kScrollBarWidth;
    {draw 2-pixel outline 3 pixels from border}
    SetPort(myList^.port);         {set port to list's port}
                                   {move out 4 pixels}
    InsetRect(myOutlineRect, -4, -4);
    GetPenState(myPenState);       {store pen state}
    IF (myList = gCurrentList) AND myList^.lActive THEN
        PenPat(black)              {draw border}
    ELSE
        PenPat(white);             {remove border}
        PenSize(2, 2);             {use 2-pixel pen}
        FrameRect(myOutlineRect);  {draw outline}
        SetPenState(myPenState);   {restore old pen state}
END;

```

The `MyDrawOutline` procedure defined in Listing 4-23 determines the rectangle to draw in by adjusting the list's visible rectangle to compensate for scroll bars and by then moving each side of the rectangle 4 pixels. (One pixel is already taken by the list border, an additional pixel is needed for space between the border and the outline, and the pen size for the outline is 2 pixels.) The list determines whether to draw or remove a list by

List Manager

comparing the list passed in with an application-defined global variable, `gCurrentList`. If the variable indicates that a list is the current list, and the `MyDrawOutline` procedure determines that the list is active, then it draws the outline; otherwise, it removes it.

Your application can use the `refCon` field of the list record to create a linked ring list of all of the lists in a window to make it easier to support outlining. That is, the `refCon` field of the first list in a window contains a handle to the second list in a window; the `refCon` field of the second list in a window contains a handle to the third, and so on, until the `refCon` field of the last list in a window contains a handle to the first.

The advantage of implementing such a ring list is that it makes it easy to change which list is the current list. In response to a Tab-key event, your application need only find the next list in a window by looking at the current list's `refCon` field and setting the `gCurrentList` variable to the list referenced by that field. Without using such a strategy, your application would need to examine the `gCurrentList` variable, determine which of a window's lists the variable corresponded to, determine which list in the window is the next list, and then set the `gCurrentList` variable to this next list.

Listing 4-24 shows an application-defined procedure that adds a list to a ring being maintained for a particular window.

Listing 4-24 Adding a list to the ring

```
PROCEDURE MyTrackList (myList: ListHandle);
VAR
    aList:      ListHandle;
BEGIN
    aList := gCurrentList;
    IF aList = NIL THEN
        gCurrentList := myList      {first ListHandle to be tracked}
    ELSE
        BEGIN
            {look for last ListHandle in ring}
            WHILE (ListHandle(aList^^.refCon) <> gCurrentList) DO
                {move to next ListHandle in ring}
                aList^^.refCon := ListHandle(aList^^.refCon)^^.refCon;
            {insert myList into ring}
            ListHandle(aList^^.refCon) := myList;
        END;
        {add link from myList to current list}
        ListHandle(myList^^.refCon) := gCurrentList;
    END;
```

List Manager

The SurfWriter application calls the `MyTrackList` procedure defined in Listing 4-24 once for each list in a window when it first opens that window. The first list added to the ring is automatically set to be the current list. SurfWriter initializes the `gCurrentList` variable to `NIL` before creating a ring for each window that uses multiple lists. In addition, SurfWriter stores the value of the `gCurrentList` variable whenever a window containing multiple lists is deactivated and then resets it when the window is activated again. That way, the `gCurrentList` variable always stores a handle to the current list of the active window.

Once all the lists in a window are linked in a ring, it is easy to write a routine that ensures that only the current list is outlined. Listing 4-25 illustrates such a routine.

Listing 4-25 Updating the outline of all lists in a window

```
PROCEDURE MyUpdateListOutlines;
VAR
  listToUpdate: ListHandle;
BEGIN
  listToUpdate := gCurrentList;
  IF listToUpdate <> NIL THEN
    REPEAT
      {move to next list in ring}
      listToUpdate := ListHandle(listToUpdate^^.refCon);
      MyDrawOutline(listToUpdate);
    UNTIL listToUpdate = gCurrentList;
  END;
```

The `MyUpdateListOutlines` procedure defined in Listing 4-25 simply calls the `MyDrawOutline` procedure for each list in the active window's ring of lists. The SurfWriter application calls this procedure each time your application changes which list is current.

List Manager

Listing 4-26 shows an application-defined procedure that responds to the user's pressing the Tab key when the Shift key is not also pressed.

Listing 4-26 Moving the outline to the next list in a window

```
PROCEDURE MyOutlineNextList;
BEGIN
    gCurrentList := ListHandle(gCurrentList^^.refCon);
    MyUpdateListOutlines;
END;
```

If the user presses Shift-Tab, your application should respond by changing the current list to the previous list. Listing 4-27 shows an application-defined procedure that does this.

Listing 4-27 Moving the outline to the previous list in a window

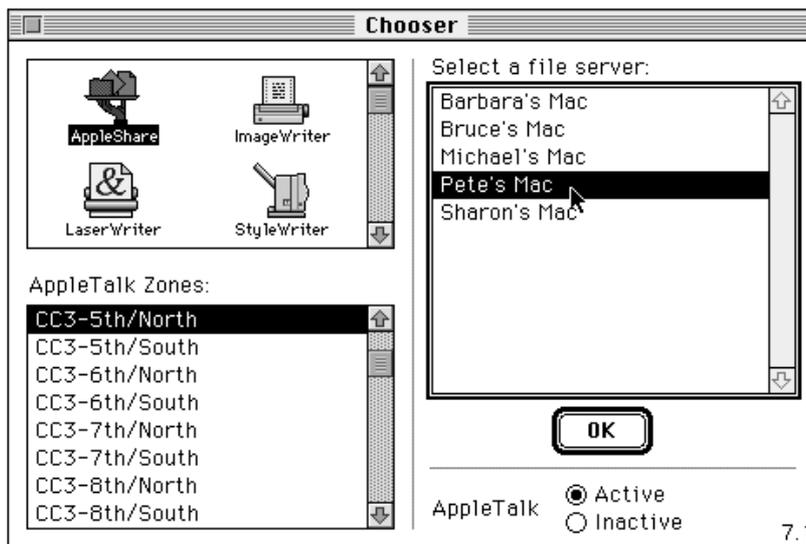
```
PROCEDURE MyOutlinePreviousList;
VAR
    previousList: ListHandle;
BEGIN
    {compute the coordinates of the list before the current list}
    previousList := gCurrentList;
    WHILE (ListHandle(previousList^^.refCon) <> gCurrentList) DO
        previousList := ListHandle(previousList^^.refCon);
    {now switch the outline to this list}
    gCurrentList := previousList;
    MyUpdateListOutlines;
END;
```

The `MyOutlineNextList` and `MyOutlinePreviousList` procedures defined in Listing 4-26 and Listing 4-27 work the same if a window contains exactly two lists.

Writing Your Own List Definition Procedure

The default list definition procedure supports only the display of unstyled text. If your application needs to display items graphically, you can create your own list definition procedure. For example, the Chooser desk accessory uses its own list definition procedure to display icons and names corresponding to Chooser extensions. Figure 4-19 illustrates the Chooser's use of a custom list definition procedure.

Figure 4-19 The Chooser's use of a custom list definition procedure



This section explains how you can write a list definition procedure. After writing a list definition procedure, you must compile it as a resource of type 'LDEF' and store it in the resource fork of any application that uses the list definition procedure.

This section provides code for a list definition procedure that supports the display of QuickDraw pictures. It works by requiring the application that uses it to store as cell data variables of type `PicHandle`. That way, each cell stores only 4 bytes of data, and the List Manager's 32 KB limit is not at risk of being approached for small lists. This list definition procedure provides enough versatility to display virtually any type of image.

You can write your own list definition procedure to store some type of data other than unstyled text. You can give your list definition procedure any name you choose, but it must be defined like this:

```
PROCEDURE MyLDEF (message: Integer; selected: Boolean;
                 VAR cellRect: Rect; theCell: Cell;
                 dataOffset: Integer; dataLen: Integer;
                 theList: ListHandle);
```

List Manager

The List Manager can send four types of messages to your list definition procedure, as indicated by a value passed in the `message` parameter. The following constants define the different types of messages:

```
CONST
    lInitMsg          = 0;      {do any special list initialization}
    lDrawMsg          = 1;      {draw the cell}
    lHiliteMsg        = 2;      {invert cell's highlight state}
    lCloseMsg         = 3;      {take any special disposal action}
```

Of the second through seventh parameters to a list definition procedure, only the `theList` parameter, which contains a handle to a list record, can be accessed by your list definition procedure in response to all four messages.

The `selected`, `cellRect`, `theCell`, `dataOffset`, and `dataLen` parameters pass information to your list definition procedure only when the value in the `message` parameter contains the `lDrawMsg` or the `lHiliteMsg` constant. These parameters provide information about the cell to be affected by the message. The `selected` parameter indicates whether the cell should be highlighted. The `cellRect` and `theCell` parameters indicate the cell's rectangle and coordinates. Finally, the `dataOffset` and `dataLen` parameters specify the offset and length of the cell's data within the relocatable block referenced by the `cells` field of the list record.

Listing 4-28 shows a list definition procedure that processes messages sent to it by the List Manager.

Listing 4-28 Processing messages to a list definition procedure

```
PROCEDURE MyLDEF (message: Integer; selected: Boolean;
                 VAR cellRect: Rect; theCell: Cell;
                 dataOffset: Integer; dataLen: Integer;
                 theList: ListHandle);
BEGIN
    CASE message OF
        lInitMsg:
            MyLDEFInit(theList);
        lDrawMsg:
            MyLDEFDraw(selected, cellRect, theCell, dataOffset,
                       dataLen, theList);
        lHiliteMsg:
            MyLDEFHighlight(selected, cellRect, theCell,
                            dataOffset, dataLen, theList);
        lCloseMsg:
            MyLDEFClose(theList);
    END;
END;
```

List Manager

The `MyLDEF` procedure defined in Listing 4-28 calls procedures defined later in this section to handle the various messages specified by the `message` parameter. It passes all relevant parameters to these message-handling procedures. Thus, it passes only the `theList` parameter to the procedures that handle the initialization and close messages.

Responding to the Initialization Message

The List Manager automatically allocates memory for a list and fills out the fields of a list record before calling your list definition procedure with a `lInitMsg` message. Your application might respond to the initialization message by changing fields of the list record, such as the `cellSize` and `indent` fields. (These fields are by default set according to a formula discussed in “About the List Manager” beginning on page 4-22.)

Many list definition procedures do not need to perform any action in response to the initialization message. For example, the list definition procedure that allows the Standard File Package to display small icons next to the names of files uses the standard cell size and thus does not need to perform any special initialization.

Since pictures can come in a variety of sizes, the pictures list definition procedure introduced in Listing 4-28 does not need to perform any special initialization either; it depends on the application that uses the list definition procedure to define the correct cell size. Thus, Listing 4-29 shows how the pictures list definition procedure responds to the initialization method.

Listing 4-29 Using the default initialization method

```
PROCEDURE MyLDEFInit (theList: ListHandle);
BEGIN
END;
```

Note

Your list definition procedure does not actually need to call a procedure that responds to the initialization message if it does not need to perform any special action. ◻

Responding to the Draw Message

Your list definition procedure must respond to the `lDrawMsg` message by examining the specified cell's data and drawing the cell as appropriate. At the same time, your list definition procedure must ensure that it does not alter the characteristics of the drawing environment.

Listing 4-30 shows how the pictures list definition procedure responds to the draw message.

Listing 4-30 Responding to the `lDrawMsg` message

```

PROCEDURE MyLDEFDraw (selected: Boolean; cellRect: Rect;
                    theCell: Cell; dataOffset: Integer;
                    dataLen: Integer; theList: ListHandle);
VAR
    savedPort:      GrafPtr;           {old graphics port}
    savedClip:      RgnHandle;         {old clip region}
    savedPenState:  PenState;          {old pen state}
    myPicture:      PicHandle;         {handle to a picture}
BEGIN
    {set up the drawing environment}
    GetPort(savedPort);                {remember the port}
    SetPort(theList^.port);            {set port to list's port}
    savedClip := NewRgn;               {create new region}
    GetClip(savedClip);               {set region to clip region}
    ClipRect(cellRect);               {set clip region to cell }
                                    { rectangle}
    GetPenState(savedPenState);        {remember pen state}
    PenNormal;                         {use normal pen type}
    {draw the cell if it contains data}
    EraseRect(cellRect);              {erase before drawing}
    IF dataLen = SizeOf(PicHandle) THEN
    BEGIN
                                                {get handle to picture}
        LGetCell(@myPicture, dataLen, theCell, theList);
                                                {draw the picture}
        DrawPicture(myPicture, cellRect);
    END;
    {select the cell if necessary}
    IF selected THEN                   {highlight cell}
        MyLDEFHighlight(selected, cellRect, theCell, dataOffset,
                        dataLen, theList);
    {restore graphics environment}
    SetPort(savedPort);                {restore saved port}
    SetClip(savedClip);               {restore clip region}
    DisposeRgn(savedClip);            {free region memory}
    SetPenState(savedPenState);        {restore pen state}
END;

```

List Manager

The `MyLDEFDraw` procedure defined in Listing 4-30 begins by saving characteristics of the current graphics environment, such as the graphics port, the clipping region, and the pen state. It also sets the pen to a normal state, and sets the clipping region to the cell's rectangle. The `MyLDEFDraw` procedure then draws in the cell rectangle by erasing the rectangle, getting the handle stored as the cell's data, and drawing the picture referenced by that handle. Then, if the cell should be selected, it simply calls the `MyLDEFHighlight` procedure defined in the next section. Before returning, `MyLDEFDraw` restores the graphics environment to its previous state and disposes of the memory it used to remember the clipping region.

Note

For more information on the `QuickDraw` routines and data structures used in Listing 4-30, see *Inside Macintosh: Imaging With QuickDraw*. ^u

Responding to the Highlighting Message

Virtually every list definition procedure should respond to the `LHiliteMsg` message in the same way, by inverting the bits in the cell's rectangle. Your list definition procedure would need to respond in a different way if selected list items should not simply be highlighted. For example, in a list of patterns, simply highlighting selected cells could confuse the user because highlighted patterns look just like other patterns.

Listing 4-31 shows how your list definition procedure can respond to the `LHiliteMsg` message in a way that is compatible with all Macintosh models, including models that do not support Color `QuickDraw`.

Listing 4-31 Responding to the `LHiliteMsg` message

```
PROCEDURE MyLDEFHighlight (selected: Boolean; cellRect: Rect;
                          theCell: Cell; dataOffset: Integer;
                          dataLen: Integer; theList: ListHandle);
BEGIN
    {use color highlighting if possible}
    BitClr(Ptr(HiliteMode), pHiliteBit);
    InvertRect(cellRect); {highlight cell rectangle}
END;
```

For more information on highlighting, see *Inside Macintosh: Imaging With QuickDraw*.

Responding to the Close Message

The List Manager sends your list definition procedure the `LCloseMsg` message immediately before disposing of the data occupied by a list. Your list definition procedure needs to respond to the close message only if it needs to perform some special processing before a list is disposed, such as releasing memory associated with a list that would not be released by the `LDispose` procedure.

List Manager

The pictures list definition procedure responds to the close message by freeing memory occupied by the list's pictures, whose handles are stored in the list. While the `LDispose` procedure will dispose of the picture handles themselves, it cannot dispose of the relocatable blocks referenced by the picture handles.

Listing 4-32 shows how the pictures list definition procedure responds to the `lCloseMsg` message.

Listing 4-32 Responding to the `lCloseMsg` message

```
PROCEDURE MyLDEFclose (theList: ListHandle);
  VAR
    aCell:      Cell;           {cell in the list}
    myPicHandle: PicHandle;     {handle stored as cell data}
    myDataLength: Integer;     {length in bytes of cell data}
BEGIN
  SetPt(aCell, 0, 0);
  IF PtInRect(aCell, theList^.dataBounds) THEN
    REPEAT
      {free memory only if cell's data is 4 bytes long}
      myDataLength := SizeOf(PicHandle);
      LGetCell(@myPicHandle, myDataLength, aCell, theList);
      IF myDataLength = SizeOf(PicHandle) THEN
        KillPicture(myPicHandle);
      UNTIL NOT LNextCell(TRUE, TRUE, aCell, theList);
    END;
  END;
```

Using the Pictures List Definition Procedure

The pictures list definition procedure introduced in Listing 4-28 can display a list containing pictures. For example, the SurfWriter application uses it to display a list of icons. SurfWriter first creates a list using the `MyCreateVerticallyScrollingList` function shown in Listing 4-1 on page 4-27. After creating the list, rather than using the default cell size as calculated by the List Manager, the SurfWriter application sets the size of the cells using the `LCellSize` procedure, as shown in Listing 4-33.

Listing 4-33 Setting the cell size of a list

```
PROCEDURE MySetCellSizeForIconList(myCellSize: Point;
                                   myList: ListHandle);
BEGIN
  LCellSize(myCellSize, myList);
END;
```

List Manager

To later add an icon to a list of icons, the SurfWriter application uses the procedure shown in Listing 4-34.

Listing 4-34 Adding an icon to a list of icons

```

PROCEDURE MyAddIconToList(myCellRect: Rect; myPlotRect: Rect;
                          myCell: Cell; theList: ListHandle;
                          VAR myPicHandle: PicHandle;
                          resID: Integer);

CONST
  kIconWidth      = 32; {width of an icon}
  kIconHeight     = 32; {height of an icon}
  kExtraSpace     = 2;  {extra space on top and to left of icon}
VAR
  myIcon: Handle;
BEGIN
  {picture occupies entire cell rectangle}
  SetRect(myCellRect, 0, 0, kIconWidth + kExtraSpace,
          kIconHeight + kExtraSpace);
  {plot icon over portion of rectangle}
  SetRect(myPlotRect, kExtraSpace, kExtraSpace, kIconWidth +
          kExtraSpace, kIconHeight + kExtraSpace);
  {load icon from resource file}
  myIcon := GetIcon(resID);
  {create the picture}
  myPicHandle := OpenPicture(myCellRect);
  PlotIcon(myPlotRect, myIcon);
  ClosePicture;
  {store handle to picture as cell data}
  LSetCell(@myPicHandle, SizeOf(PicHandle), myCell, theList);
  {release icon resource}
  ReleaseResource(myIcon);
END;

```

Note that the `MyAddIconToList` procedure uses the QuickDraw routines `OpenPicture` and `ClosePicture` to bracket the set of drawing commands that it uses to define the picture data for a particular cell. It then stores the handle to the picture as the cell's data, so that the pictures list definition procedure can draw the picture within the cell.

List Manager Reference

This section describes the data structures and routines that are specific to the List Manager. The “Data Structures” section shows the data structures for the cell, the data handle, and the list record. The “List Manager Routines” section beginning on page 4-70 describes the routines that your application can use to create, manipulate, get information about, and dispose of lists. The “Application-Defined Routines” section beginning on page 4-96 describes list definition procedures, match functions, and click-loop procedures.

Data Structures

This section describes the data structures that the List Manager uses to store information about a list.

Your application can use the cell record to specify the coordinates of a cell. For example, your application must specify cell coordinates to the `LAddToCell` procedure to add data to a cell.

The List Manager uses a data handle internally to store information about the contents of a list’s cells. The List Manager provides routines that allow you to access information contained in this data handle.

Finally, the List Manager uses a list record to store a variety of information about a list. To obtain some types of information about a list, your application might need to access the fields of the list record directly.

The Cell Record

A cell record specifies the coordinates of a cell in a list. The `Cell` data type defines a cell record.

```
TYPE Cell = Point;
```

Field descriptions

- `v` The row number of the cell.
- `h` The column number of the cell.

Note that column and row numbers are 0-based. Also note that this chapter designates cells using the notation (*column-1*, *row-1*), so that a cell with coordinates (2,5) is in the third column and sixth row of a list. You specify a cell with coordinates (2,5) by setting the cell’s `h` field to 2 and its `v` field to 5.

The Data Handle

The List Manager uses a data handle to store information about a list. The `DataHandle` data type defines a data handle.

```
TYPE  dataArray          = PACKED ARRAY[0..32000] OF Char;
      DataPtr           = ^dataArray;
      DataHandle        = ^DataPtr;
```

Your application should not change the information in a data handle directly. Your application can, however, read data stored in a list's data handle directly by calling the `GetCellDataLocation` procedure to find the offset of a cell's data into the data handle and the length of the cell's data.

The List Record

The List Manager uses a list record to store many types of information about a list. Usually you access a list record through a handle to the list record defined by the data type `ListHandle`. The `ListRec` data type defines a list record.

```
TYPE ListRec =
RECORD
  rView:      Rect;           {list's display rectangle}
  port:      GrafPtr;        {list's graphics port}
  indent:    Point;          {indent distance for drawing}
  cellSize:  Point;          {size in pixels of a cell}
  visible:   Rect;           {boundary of visible cells}
  vScroll:   ControlHandle;  {vertical scroll bar}
  hScroll:   ControlHandle;  {horizontal scroll bar}
  selFlags:  SignedByte;     {selection flags}
  lActive:   Boolean;        {TRUE if list is active}
  lReserved: SignedByte;     {reserved}
  listFlags: SignedByte;     {automatic scrolling flags}
  cliKTime:  LongInt;        {TickCount at time of last click}
  cliKLoc:   Point;          {position of last click}
  mouseLoc:  Point;          {current mouse location}
  lCliKLoop: Ptr;            {routine called by LClick}
  lastClick: Cell;           {last cell clicked}
  refCon:    LongInt;        {for application use}
  listDefProc:
    Handle;                  {list definition procedure}
```

List Manager

```

    userHandle: Handle;           {for application use}
    dataBounds: Rect;            {boundary of cells allocated}
    cells:      DataHandle;      {cell data}
    maxIndex:   Integer;        {used internally}
    cellArray:  ARRAY[1..1] OF Integer;
END;
    ListPtr     = ^ListRec;      {pointer to a list record}
    ListHandle  = ^ListPtr;      {handle to a list record}

```

Field descriptions

<code>rView</code>	Specifies the rectangle in which the list's visible rectangle is located, in local coordinates of the graphics port specified by the <code>port</code> field. Note that the list's visible rectangle does not include the area needed for the list's scroll bars. The width of a vertical scroll bar (which equals the height of a horizontal scroll bar) is 15 pixels.
<code>port</code>	Specifies the graphics port of the window containing the list.
<code>indent</code>	Defines the location, relative to the upper-left corner of a cell, at which drawing should begin. List definition procedures should set this field to a value appropriate to the type of data that a cell in a list is to contain.
<code>cellSize</code>	Contains the size in pixels of each cell in the list. When your application creates a list, it can either specify the cell size or let the List Manager calculate the cell size. You should not change the <code>cellSize</code> field directly; if you need to change the cell size after creating a list, use the <code>LCellSize</code> procedure.
<code>visible</code>	Specifies the cells in a list that are visible within the area specified by the <code>rView</code> field. The List Manager sets the <code>left</code> and <code>top</code> fields of <code>visible</code> to the coordinates of the first visible cell; however, the List Manager sets the <code>right</code> and <code>bottom</code> fields so that each is 1 greater than the horizontal and vertical coordinates of the last visible cell. For example, if a list contains 4 columns and 10 rows but only the first 2 columns and the first 5 rows are visible (that is, the last visible cell has coordinates (1,4)), the List Manager sets the <code>visible</code> field to (0,0,2,5).
<code>vScroll</code>	Contains a control handle for a list's vertical scroll bar, or <code>NIL</code> if a list does not have a vertical scroll bar.
<code>hScroll</code>	Contains a control handle for a list's horizontal scroll bar, or <code>NIL</code> if a list does not have a vertical scroll bar.

List Manager

`selFlags` Indicates the selection flags for a list. When your application creates a list, the List Manager clears the `selFlags` field to 0. This defines the List Manager's default selection algorithm. To change the default behavior for a particular list, set the desired bits in the list's `selFlags` field.

You can use these constants to refer to bits in this field:

```
CONST
    {allow only one item to be selected at once}
    lOnlyOne      = -128;
    {enable multiple item selection without Shift}
    lExtendDrag   = 64;
    {prevent discontinuous selections}
    lNoDisjoint   = 32;
    {reset list before responding to Shift-click}
    lNoExtend     = 16;
    {Shift-drag selects items passed by cursor}
    lNoRect       = 8;
    {allow use of Shift key to deselect items}
    lUseSense     = 4;
    {disable highlighting of empty cells}
    lNoNilNilite  = 2;
```

`lActive` Indicates whether the list is active (TRUE if active, FALSE if inactive).

`lReserved` Reserved.

`listFlags` Indicates whether the List Manager should automatically scroll the list if the user clicks the list and then drags the cursor outside the list display rectangle.

The following constants define bits in this field that determine whether horizontal autoscrolling and vertical autoscrolling are enabled:

```
CONST
    {allow automatic vertical scrolling}
    lDoVAutoscroll = 2;
    {allow automatic horizontal scrolling}
    lDoHAutoscroll = 1;
```

By default, the List Manager enables horizontal autoscrolling for a list if the list includes a horizontal scroll bar, and enables vertical autoscrolling for a list if the list includes a vertical scroll bar.

List Manager

<code>clickTime</code>	Specifies the time in ticks of the last click in the list. If your application depends on the value contained in this field, then your application should update the field should the application select a list item in response to keyboard input.
<code>clickLoc</code>	Specifies the location in local coordinates of the last click in the list.
<code>mouseLoc</code>	Indicates the current location of the cursor in local coordinates. This value is continuously updated by the <code>LClick</code> function after the user clicks a list.
<code>lClickLoop</code>	Contains a pointer to a click-loop procedure repeatedly called by the <code>LClick</code> function, or <code>NIL</code> if the default click-loop procedure is to be used. For information on click-loop procedures, see “Click-Loop Procedures” beginning on page 4-100.
<code>lastClick</code>	Specifies the coordinates of the last cell in the list that was clicked. This may not be the same as the last cell selected if the user selects a range of cells by Shift-dragging or Command-dragging. If your application depends on the value contained in this field, then your application should update the field whenever your application selects a list item in response to keyboard input.
<code>refCon</code>	Contains 4 bytes for use by your application.
<code>listDefProc</code>	Contains a handle to the code for the list definition procedure that defines how the list is drawn.
<code>userHandle</code>	Contains 4 bytes that your application can use as needed. For example, your application might use this field to store a handle to additional storage associated with the list. However, the <code>LDispose</code> procedure does not automatically release this storage when disposing of the list.
<code>dataBounds</code>	Specifies the range of cells in a list. When your application creates a list, it specifies the initial bounds of the list. As your application adds rows and columns, the List Manager updates this field. The List Manager sets the <code>left</code> and <code>top</code> fields of <code>dataBounds</code> to the coordinates of the first cell in the list; the List Manager sets the <code>right</code> and <code>bottom</code> fields so that each is 1 greater than the horizontal and vertical coordinates of the last cell. For example, if a list contains 4 columns and 10 rows (that is, the last cell in the list has coordinates (3,9)), the List Manager sets the <code>dataBounds</code> field to (0,0,4,10).
<code>cells</code>	Contains a handle to a relocatable block used to store cell data. Your application should not change the contents of this relocatable block directly.
<code>maxIndex</code>	Used internally.
<code>cellArray</code>	Contains offsets to data that indicate the location of different cells' data within the data handle specified by the <code>cells</code> parameter. Your application should not access this field directly.

List Manager Routines

This section describes the routines you can use to

- n create and dispose of lists
- n add and delete rows and columns to and from lists
- n find or change cells' selection status
- n read or change cell data
- n respond to list events
- n affect the display of a list
- n get information about cells
- n change the size of a list or of a cell contained in a list

s WARNING

The List Manager's routines are contained in a resource of resource type 'PACK'. Calling any of the routines described in this section could result in the loading of the package resource and the allocation of memory. Thus, your application should not call any of the routines described in this section at interrupt time. For more information on packages, see *Inside Macintosh: Operating System Utilities*. s

Creating and Disposing of Lists

You can create a list by calling the `LNew` function. When you are through with the list, you can dispose of it by calling the `LDispose` procedure.

LNew

You can use the `LNew` function to create a new list in a window.

```
FUNCTION LNew (rView, dataBounds: Rect; cSize: Point;
              theProc: Integer; theWindow: WindowPtr;
              drawit, hasGrow, scrollHoriz, scrollVert: Boolean)
              : ListHandle;
```

`rView` **The rectangle in which to display the list, in local coordinates of the window specified by the `theWindow` parameter. This rectangle does not include the area to be taken up by the list's scroll bars.**

`dataBounds` **The initial data bounds for the list. By setting the `left` and `top` fields of this rectangle to (0,0) and the `right` and `bottom` fields to (`kMyInitialColumns`, `kMyInitialRows`), your application can create a list that has `kMyInitialColumns` columns and `kMyInitialRows` rows.**

List Manager

<code>cSize</code>	The size of each cell in the list. If your application specifies (0,0) and is using the default list definition procedure, the List Manager sets the <code>v</code> coordinate of this parameter to the sum of the ascent, descent, and leading of the current font, and it sets the <code>h</code> coordinate using the following formula: $\text{cSize.h} := (\text{rView.right} - \text{rView.left}) \text{ DIV } (\text{dataBounds.right} - \text{dataBounds.left})$
<code>theProc</code>	The resource ID of the list definition procedure to use for the list. To use the default list definition procedure, which supports the display of unstyled text, specify a resource ID of 0.
<code>theWindow</code>	A pointer to the window in which to install the list.
<code>drawIt</code>	A Boolean value that indicates whether the List Manager should initially enable the automatic drawing mode. When the automatic drawing mode is enabled, the List Manager automatically redraws the list whenever a change is made to it. You can later change this setting using the <code>LSetDrawingMode</code> procedure. Your application should leave the automatic drawing mode disabled only for short periods of time when making changes to a list (by, for example, adding rows and columns).
<code>hasGrow</code>	A Boolean value that indicates whether the List Manager should leave room for a size box. The List Manager does not actually draw the grow icon. Usually, your application can draw it with the Window Manager's <code>DrawGrowIcon</code> procedure.
<code>scrollHoriz</code>	A Boolean value that indicates whether the list should contain a horizontal scroll bar. Specify <code>TRUE</code> if your list requires a horizontal scroll bar; specify <code>FALSE</code> otherwise.
<code>scrollVert</code>	Indicates whether the list should contain a vertical scroll bar. Specify <code>TRUE</code> if your list requires a vertical scroll bar; specify <code>FALSE</code> otherwise.

DESCRIPTION

The `LNew` function attempts to create a list defined by the function's parameters and returns a handle to the newly created list. If the `LNew` function cannot allocate the list, it returns `NIL`. This might happen if there is not enough memory available or if `LNew` cannot load the resource specified by the `theProc` parameter. If the `LNew` function returns successfully, then all of the fields of the list record referenced by the returned handle are correctly set.

If the list contains a horizontal or vertical scroll bar and the window specified by the parameter `theWindow` is visible, `LNew` draws the scroll bar for the new list in the window just outside the list's visible rectangle specified by the `rView` parameter. The `LNew` function does not, however, draw a 1-pixel border around the list's visible rectangle.

SPECIAL CONSIDERATIONS

You should not call the `LNew` 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 `LNew` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0044</code>

SEE ALSO

See Listing 4-1 on page 4-27 for an example of how to use the `LNew` function.

LDispose

You can use the `LDispose` procedure to dispose of the memory associated with a list that you no longer need.

```
PROCEDURE LDispose (lHandle: ListHandle);
```

`lHandle` The list to be disposed of.

DESCRIPTION

The `LDispose` procedure releases all memory allocated by the List Manager in creating a list. First, `LDispose` issues a close request to the list definition procedure and calls the Control Manager procedure `DisposeControl` for the list's scroll bars (if any).

`LDispose` then uses the Memory Manager to free the memory referenced by the `cells` field, then disposes of the list record itself.

Because `LDispose` disposes of data associated with cells in your list, there is no need to clear the data from list cells or to delete individual rows and columns before calling `LDispose`.

The `LDispose` procedure does not dispose of any memory associated with a list that the List Manager has not allocated. In particular, `LDispose` does not dispose of any memory referenced by the `userHandle` field of the list record. Your application is responsible for deallocating any memory it has allocated through the `userHandle` field before calling `LDispose`.

SPECIAL CONSIDERATIONS

You should not call the `LDispose` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDispose` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0028</code>

Adding and Deleting Columns and Rows To and From a List

You can use the `LAddColumn` and `LAddRow` functions to add one or more columns or rows to a list, and you can use the `LDelColumn` and `LDelRow` procedures to delete one or more columns or rows from a list.

LAddColumn

You can use the `LAddColumn` function to add one or more columns to a list.

```
FUNCTION LAddColumn (count: Integer; colNum: Integer;
                    lHandle: ListHandle): Integer;
```

<code>count</code>	The number of columns to add.
<code>colNum</code>	The column number of the first column to add.
<code>lHandle</code>	The list to which to add the columns.

DESCRIPTION

The `LAddColumn` function inserts into the given list the number of columns specified by the `count` parameter, starting at the column specified by the `colNum` parameter. The `LAddColumn` function returns as its function result the column number of the first column added, which is equal to the value specified by the `colNum` parameter if that value is a valid column number.

If the column number specified by `colNum` is not already in the list, then new last columns are added. The value returned by the `LAddColumn` function thus has significance only in this case.

S WARNING

If there is insufficient memory in the heap to add the new columns, the `LAddColumn` function may fail to add the new columns although it returns a positive function result. Be sure there is enough memory in the heap to allocate the new columns before calling `LAddColumn`.

Columns whose column numbers are initially greater than `colNum` have their column numbers increased by `count`.

If the automatic drawing mode is enabled and the columns added by `LAddColumn` are visible, then the list (including its scroll bars) is updated. New cells created by a call to `LAddColumn` are initially empty.

You may add columns to a list that does not yet have rows. The `dataBounds` field of the list record reflects that the list has columns, but you can only access cells when both rows and columns have been added.

SPECIAL CONSIDERATIONS

You should not call the `LAddColumn` 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 `LAddColumn` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0004</code>

LAddRow

You can use the `LAddRow` function to add one or more rows to a list.

```
FUNCTION LAddRow (count: Integer; rowNum: Integer;
                 lHandle: ListHandle): Integer;
```

<code>count</code>	The number of rows to add.
<code>rowNum</code>	The row number of the first row to add.
<code>lHandle</code>	The list to add the rows to.

DESCRIPTION

The `LAddRow` function inserts into the given list the number of rows specified by the `count` parameter, starting at the row specified by the `rowNum` parameter. The `LAddRow` function returns as its function result the row number of the first row added, which is equal to the value specified by the `rowNum` parameter if that value is a valid row number.

If the row number specified by `rowNum` is not already in the list, then new last rows are added. The value returned by the `LAddRow` function thus has significance only in this case.

S WARNING

If there is insufficient memory in the heap to add the new rows, the `LAddRow` function may fail to add the new rows although it returns a positive function result. Be sure there is enough memory in the heap to allocate the new rows before calling `LAddRow`.

Rows whose row numbers are initially greater than `rowNum` have their row numbers increased by `count`.

List Manager

If the automatic drawing mode is enabled and the rows added by `LAddRow` are visible, then the list (including its scroll bars) is updated. New cells created by a call to `LAddRow` are initially empty.

You may add rows to a list that does not yet have columns. The `dataBounds` field of the list record reflects that the list has rows, but you can only access cells when both rows and columns have been added.

SPECIAL CONSIDERATIONS

You should not call the `LAddRow` 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 `LAddRow` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0008</code>

SEE ALSO

For an example that adds rows to a list, see Listing 4-4 on page 4-31.

LDelColumn

You can use the `LDelColumn` procedure to delete one or more columns from a list.

```
PROCEDURE LDelColumn (count: Integer; colNum: Integer;
                    lHandle: ListHandle);
```

<code>count</code>	The number of columns to delete, or 0 to delete all columns.
<code>colNum</code>	The column number of the first column to delete.
<code>lHandle</code>	The list from which to delete the columns.

DESCRIPTION

The `LDelColumn` procedure deletes the number of columns specified by the `count` parameter, starting at the column specified by the `colNum` parameter.

If the column specified by `colNum` is invalid, then nothing is done.

Your application can quickly delete all columns from a list (and thus delete all cell data) simply by setting the `count` parameter to 0. The number of rows is left unchanged. Your application can achieve the same effect by setting the `colNum` parameter to `lHandle^.dataBounds.left` and setting the `count` parameter to a value greater than `lHandle^.dataBounds.right - lHandle^.dataBounds.left`.

List Manager

Columns whose column numbers are initially greater than `colNum` have their column numbers decreased by `count`.

If the automatic drawing mode is enabled and one or more of the columns deleted by `LDelColumn` are visible, then the list (including its scroll bars) is updated.

SPECIAL CONSIDERATIONS

You should not call the `LDelColumn` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDelColumn` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0020</code>

LDelRow

You can use the `LDelRow` procedure to delete one or more rows from a list.

```
PROCEDURE LDelRow (count: Integer; rowNum: Integer;
                  lHandle: ListHandle);
```

<code>count</code>	The number of rows to delete, or 0 to delete all rows.
<code>rowNum</code>	The row number of the first row to delete.
<code>lHandle</code>	The list from which to delete the rows.

DESCRIPTION

The `LDelRow` procedure deletes the number of rows specified by the `count` parameter, starting at the row specified by the `rowNum` parameter.

If the row specified by `rowNum` is invalid, then nothing is done.

Your application can quickly delete all rows from a list (and thus delete all cell data) simply by setting the `count` parameter to 0. The number of columns is left unchanged. Your application can achieve the same effect by setting the `rowNum` parameter to `lHandle^.dataBounds.top` and setting the `count` parameter to a value greater than `lHandle^.dataBounds.bottom - lHandle^.dataBounds.top`.

Rows whose row numbers are initially greater than `rowNum` have their row numbers decreased by `count`.

If the automatic drawing mode is enabled and one or more of the rows deleted by `LDelRow` are visible, then the list (including its scroll bars) is updated.

SPECIAL CONSIDERATIONS

You should not call the `LDelRow` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDelRow` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0024</code>

Determining or Changing the Selection

Your application can use the `LGetSelect` function to determine whether a certain cell is selected or to find the next selected cell. To select or deselect a specific cell, your application can use the `LSetSelect` procedure.

LGetSelect

You can use the `LGetSelect` function to get information about which cells are selected.

```
FUNCTION LGetSelect (next: Boolean; VAR theCell: Cell;
                   lHandle: ListHandle): Boolean;
```

<code>next</code>	A Boolean value that indicates whether <code>LGetSelect</code> should check only the cell specified by the parameter <code>theCell</code> (<code>next = FALSE</code>), or whether it should try to find the next selected cell (<code>next = TRUE</code>).
<code>theCell</code>	On input, specifies the first cell whose selection status should be checked. If <code>next</code> is <code>TRUE</code> , then this parameter on output indicates the next selected cell greater than or equal to the cell specified on input. Otherwise, this parameter remains unchanged.
<code>lHandle</code>	The list in which the selection is being checked.

DESCRIPTION

The behavior of the `LGetSelect` function depends on the value specified in the `next` parameter.

If `next` is `TRUE`, then `LGetSelect` searches the list for the first selected cell beginning at the cell specified by `theCell`. (In particular, `LGetSelect` first checks cells in row `theCell.v`, and then cells in the next row, and so on.) If it finds a selected cell, `LGetSelect` returns `TRUE` and sets the parameter `theCell` to the coordinates of the selected cell. If it does not find a selected cell, `LGetSelect` returns `FALSE`.

If `next` is `FALSE`, then `LGetSelect` checks only the cell specified by the parameter `theCell`. If the cell is selected, `LGetSelect` returns `TRUE`. Otherwise, it returns `FALSE`.

SPECIAL CONSIDERATIONS

You should not call the `LGetSelect` 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 `LGetSelect` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$003C</code>

SEE ALSO

For examples that determine which items are selected in a list, see “Working With List Selections” beginning on page 4-34.

LSetSelect

You can use the `LSetSelect` procedure to select or deselect a cell.

```
PROCEDURE LSetSelect (setIt: Boolean; theCell: Cell;
                    lHandle: ListHandle);
```

<code>setIt</code>	A Boolean value that indicates whether the <code>LSetSelect</code> procedure should select or deselect the specified cell. Specify <code>TRUE</code> to select the cell; specify <code>FALSE</code> to deselect the cell.
<code>theCell</code>	The cell to be selected or deselected.
<code>lHandle</code>	The list containing the cell to be selected or deselected.

DESCRIPTION

If `setIt` is `TRUE`, then the `LSetSelect` procedure selects the cell specified by the `theCell` parameter in the list specified by `lHandle`. If the cell is already selected, `LSetSelect` does nothing.

If `setIt` is `FALSE`, then `LSetSelect` deselects the cell specified by `theCell`. If the cell is already deselected, `LSetSelect` does nothing.

If a cell's selection status is changed and the cell is visible, `LSetSelect` redraws the cell.

SPECIAL CONSIDERATIONS

You should not call the `LSetSelect` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSetSelect` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$005C</code>

SEE ALSO

For examples that change the items selected in a list, see “Working With List Selections” beginning on page 4-34.

Accessing and Manipulating Cell Data

To change the data contained in a cell, your application ordinarily uses the `LSetCell` procedure. Alternatively, it can use the `LAddToCell` procedure to append data to a cell, or the `LClrCell` procedure to clear all data from a cell. To find the data in a cell, your application can use the `LGetCellDataLocation` procedure to find the location of a cell's data in memory. Or, your application can use the `LGetCell` procedure to copy the data elsewhere in memory.

LSetCell

You can use the `LSetCell` procedure to change the data contained in a cell.

```
PROCEDURE LSetCell (dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                   lHandle: ListHandle);
```

<code>dataPtr</code>	A pointer to the new data for a cell.
<code>dataLen</code>	The length in bytes of the data pointed to by the <code>dataPtr</code> parameter.
<code>theCell</code>	The coordinates of the cell to hold the new data.
<code>lHandle</code>	The list containing the cell given in the <code>theCell</code> parameter.

DESCRIPTION

The `LSetCell` procedure sets the data of the cell specified by the parameter `theCell` to `dataLen` bytes of data beginning at the location specified by `dataPtr`. Any previous cell data in `theCell` is replaced.

If the cell coordinates specified by the `theCell` parameter are invalid, then `LSetCell` does nothing.

S WARNING

If there is insufficient memory in the heap, the `LSetCell` procedure may fail to set the cell's data. s

If the data of a visible cell is changed and the automatic drawing mode is enabled, `LSetCell` updates the list.

SPECIAL CONSIDERATIONS

You should not call the `LSetCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSetCell` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0058</code>

SEE ALSO

For an example that sets the data of cells in a list, see Listing 4-4 on page 4-31.

LAddToCell

You can use the `LAddToCell` procedure to append data to the data already contained in a cell.

```
PROCEDURE LAddToCell (dataPtr: Ptr; dataLen: Integer;
                    theCell: Cell; lHandle: ListHandle);
```

<code>dataPtr</code>	A pointer to the data to be appended.
<code>dataLen</code>	The length in bytes of the data pointed to by the <code>dataPtr</code> parameter.
<code>theCell</code>	The coordinates of the cell to which the data should be appended.
<code>lHandle</code>	The list containing the cell given in the <code>theCell</code> parameter.

DESCRIPTION

The `LAddToCell` procedure appends `dataLen` bytes of data beginning at the location specified by `dataPtr` to data already contained in the cell specified by the parameter `theCell`.

If the cell coordinates specified by the parameter `theCell` are invalid, then the `LAddToCell` procedure does nothing.

If the data of a visible cell is changed and the automatic drawing mode is enabled, `LAddToCell` updates the list.

SPECIAL CONSIDERATIONS

You should not call the `LAddToCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LAddToCell` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$000C</code>

LClrCell

You can use the `LClrCell` procedure to clear the data contained in a cell.

```
PROCEDURE LClrCell (theCell: Cell; lHandle: ListHandle);
```

<code>theCell</code>	The coordinates of the cell to be cleared.
<code>lHandle</code>	The list containing the cell given in the <code>theCell</code> parameter.

DESCRIPTION

The `LClrCell` procedure clears the data contained in the cell specified by the `theCell` parameter.

If the cell coordinates specified by the `theCell` parameter are invalid, then the `LClrCell` procedure does nothing.

If the data of a visible cell is cleared and the automatic drawing mode is enabled, `LClrCell` updates the list.

SPECIAL CONSIDERATIONS

You should not call the `LClrCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LClrCell` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$001C</code>

LGetCellDataLocation

You can find the memory location of cell data by using the `LGetCellDataLocation` procedure. The `LGetCellDataLocation` procedure is also available as the `LFind` procedure.

```
PROCEDURE LGetCellDataLocation (VAR offset, len: Integer;
                               theCell: Cell;
                               lHandle: ListHandle);
```

<code>offset</code>	The <code>LGetCellDataLocation</code> procedure returns in this parameter the offset of the cell's data, specified from the beginning of the data handle referenced by the <code>cells</code> field of the list record.
<code>len</code>	The <code>LGetCellDataLocation</code> procedure returns in this parameter the length of the cell's data in bytes.
<code>theCell</code>	The cell whose data's location is sought.
<code>lHandle</code>	The list containing the cell specified by the parameter <code>theCell</code> .

DESCRIPTION

Your application can use the `LGetCellDataLocation` procedure to read cell data. The `cells` field of the list record contains a handle to a relocatable block used to store all cell data. When `LGetCellDataLocation` returns, the `offset` parameter contains the offset of the specified cell's data in this relocatable block, and the `len` parameter specifies the length in bytes of the cell's data. In other words, the first byte of cell data is located at `Ptr(ORD4(lHandle^.cells) + offset)`, and the last byte of cell data is located at `Ptr(ORD4(lHandle^.cells) + offset + len)`.

If the cell coordinates specified by the parameter `theCell` are invalid, then `LGetCellDataLocation` sets the `offset` and `len` parameters to `-1`.

S WARNING

Your application should not modify the contents of the `cells` field directly. To change a cell's data, use the `LSetCell` procedure or the `LAddToCell` procedure. s

SPECIAL CONSIDERATIONS

You should not call the `LGetCellDataLocation` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LGetCellDataLocation` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0034</code>

SEE ALSO

For an example that uses the `LGetCellDataLocation` procedure to get the data of a cell, see Listing 4-13 on page 4-41.

LGetCell

You can use the `LGetCell` procedure to copy a cell's data.

```
PROCEDURE LGetCell (dataPtr: Ptr; VAR dataLen: Integer;
                  theCell: Cell; lHandle: ListHandle);
```

<code>dataPtr</code>	A pointer to the location to which to copy the cell's data.
<code>dataLen</code>	On entry, specifies the maximum number of bytes to copy. On exit, indicates the number of bytes actually copied.
<code>theCell</code>	The cell whose data is to be copied.
<code>lHandle</code>	The list containing the cell specified by the parameter <code>theCell</code> .

DESCRIPTION

The `LGetCell` procedure copies up to `dataLen` bytes of the data of the cell specified by `theCell` to the memory location pointed to by `dataPtr`. If the cell data is longer than `dataLen`, only `dataLen` bytes are copied and the `dataLen` parameter is unchanged. If the cell data is shorter than `dataLen`, then `LGetCell` sets `dataLen` to the length in bytes of the cell's data.

SPECIAL CONSIDERATIONS

You should not call the `LGetCell` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LGetCell` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0038</code>

Responding to Events Affecting Lists

Your application can respond to mouse-down events in a list, activate events for a window containing a list, and update events for a window containing a list simply by calling the `LClick` function, the `LActivate` procedure, and the `LUpdate` procedure, respectively. The List Manager does not include a routine that automatically responds to keyboard events; for information on responding to those, see “Supporting Keyboard Navigation of Lists” beginning on page 4-45.

LClick

To process a mouse-down event in a list, use the `LClick` function.

```
FUNCTION LClick (pt: Point; modifiers: Integer;
                lHandle: ListHandle): Boolean;
```

<code>pt</code>	The location in local coordinates of the mouse-down event. Your application can simply call <code>GlobalToLocal(myEvent.where)</code> and then pass <code>myEvent.where</code> in this parameter.
<code>modifiers</code>	An integer value corresponding to the <code>modifiers</code> field of the event record.
<code>lHandle</code>	The list in which the mouse-down event occurred.

DESCRIPTION

The `LClick` function responds to the mouse-down event whose location and modifiers are specified by the `pt` and `modifiers` parameters. The `LClick` function handles all user interaction until the user releases the mouse button. The `LClick` function returns `TRUE` if the click was a double-click, or `FALSE` otherwise.

If the `pt` parameter specifies a portion of the list’s visible rectangle, then cells are selected with an algorithm that depends on the list’s selection flags and on the `modifiers` parameter. If the user drags the cursor above or below the list’s visible rectangle and vertical autoscrolling is enabled, then the List Manager vertically autoscrolls the list. If the user drags the cursor to the right or the left of the list’s visible rectangle and horizontal autoscrolling is enabled, then the List Manager horizontally autoscrolls the list.

If the `pt` parameter specifies a point within the list’s scroll bar, then the List Manager calls the scroll bar’s control definition procedure to track the cursor and it scrolls the list appropriately.

SPECIAL CONSIDERATIONS

You should not call the `LClick` 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 `LClick` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0018</code>

SEE ALSO

For information on enabling and disabling autoscrolling, see “About the List Manager” beginning on page 4-22. For information on responding to mouse-down events, see “Responding to Events Affecting a List” on page 4-32.

LActivate

When your application receives an activate event for a window containing a list, it should activate or deactivate the list as appropriate. You can use the `LActivate` procedure to perform highlighting of the cells and to show or hide any scroll bars.

```
PROCEDURE LActivate (act: Boolean; lHandle: ListHandle);
```

<code>act</code>	A Boolean value that indicates whether the list should be activated. Specify <code>TRUE</code> to activate the list. Specify <code>FALSE</code> to deactivate the list.
<code>lHandle</code>	The list to be activated or deactivated.

DESCRIPTION

The `LActivate` procedure activates the list specified by the `lHandle` parameter if `act` is `TRUE` and deactivates it otherwise.

If a list is being deactivated, `LActivate` removes highlighting from selected cells and hides the scroll bars. If a list is being activated, `LActivate` highlights selected cells and shows the scroll bars.

The `LActivate` procedure has no effect on a list's size box, if one exists.

SPECIAL CONSIDERATIONS

You should not call the `LActivate` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LActivate` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0000</code>

SEE ALSO

For information on responding to activate events in lists, see “Responding to Events Affecting a List” beginning on page 4-32. For general information on events, see the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

LUpdate

To respond to an update event, use the `LUpdate` procedure.

```
PROCEDURE LUpdate (theRgn: RgnHandle; lHandle: ListHandle);
```

`theRgn` The visible region of the list’s port after a call to the Window Manager’s `BeginUpdate` procedure.

`lHandle` The list to be updated.

DESCRIPTION

The `LUpdate` procedure redraws all visible cells in the list specified by the `lHandle` parameter that intersect the region specified by the parameter `theRgn`. It also redraws the scroll bars if they intersect the region.

You should bracket calls to `LUpdate` by calls to the Window Manager procedures `BeginUpdate` and `EndUpdate`.

SPECIAL CONSIDERATIONS

You should not call the `LUpdate` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LUpdate` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0064</code>

SEE ALSO

For information on responding to update events in lists, see “Responding to Events Affecting a List” beginning on page 4-32. For general information on events, see the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Modifying a List's Appearance

Your application can use the `LSetDrawingMode` procedure to enable or disable automatic drawing of lists. If your application uses `LSetDrawingMode` to temporarily disable list drawing, then it must call the `LDraw` procedure to draw a cell when its appearance changes, or when new rows or columns are added to the list. To automatically scroll a list so that the first selected cell is the first cell visible, your application can use the `LAutoScroll` procedure. To scroll a list a certain number of cells horizontally and vertically, your application can use the `LScroll` procedure.

LSetDrawingMode

You can use the `LSetDrawingMode` procedure to change the automatic drawing mode specified when creating a list. The `LSetDrawingMode` procedure is also available as the `LDoDraw` procedure.

```
PROCEDURE LSetDrawingMode (drawIt: Boolean; lHandle: ListHandle);
```

`drawIt` A Boolean value that indicates whether the List Manager should enable the automatic drawing mode. Specify `TRUE` to enable the automatic drawing mode. Specify `FALSE` to disable the automatic drawing mode.

`lHandle` The list whose drawing mode is being changed.

DESCRIPTION

The `LSetDrawingMode` procedure sets the List Manager's drawing mode for the list specified by the `lHandle` parameter to the state specified by the `drawIt` parameter.

While the automatic drawing mode is turned off, all cell drawing and highlighting are disabled, and the scroll bar does not function properly. Thus, your application should disable the automatic drawing mode only for short periods of time. After enabling it, your application should ensure that the list is redrawn.

SPECIAL CONSIDERATIONS

You should not call the `LSetDrawingMode` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSetDrawingMode` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$002C</code>

SEE ALSO

For an example that disables and then reenables the automatic drawing mode, see “Adding Rows and Columns to a List” beginning on page 4-30.

LDraw

You can use the `LDraw` procedure to draw a cell in a list. Ordinarily, you should only need to use `LDraw` when the automatic drawing mode has been disabled.

```
PROCEDURE LDraw (theCell: Cell; lHandle: ListHandle);
```

`theCell` The cell to draw.

`lHandle` The list containing the cell identified by the parameter `theCell`.

DESCRIPTION

The `LDraw` procedure draws the cell specified by the parameter `theCell`. The List Manager makes the list's graphics port the current port, sets the clipping region to the cell's rectangle, and calls the list definition procedure to draw the cell. It restores the clipping region and port before exiting.

SPECIAL CONSIDERATIONS

You should not call the `LDraw` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LDraw` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0030</code>

LAutoScroll

You can use the `LAutoScroll` procedure to make the first selected cell visible.

```
PROCEDURE LAutoScroll (lHandle: ListHandle);
```

`lHandle` The list to be scrolled.

DESCRIPTION

The `LAutoScroll` procedure scrolls the list specified by the `lHandle` parameter so that the first selected cell is in the upper-left corner of the list's visible rectangle.

SPECIAL CONSIDERATIONS

You should not call the `LAutoScroll` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LAutoScroll` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0010</code>

LScroll

You can use the `LScroll` procedure to scroll a list a specified number of rows and columns.

```
PROCEDURE LScroll (dCols: Integer; dRows: Integer;
                  lHandle: ListHandle);
```

<code>dCols</code>	The number of columns to scroll. Specify a positive number to scroll down (that is, each cell moves up), and a negative number to scroll up.
<code>dRows</code>	The number of rows to scroll. Specify a positive number to scroll right (that is, each cell moves left), and a negative number to scroll left.
<code>lHandle</code>	The list to be scrolled.

DESCRIPTION

The `LScroll` procedure scrolls the list specified by the `lHandle` procedure the number of columns and rows specified by `dCols` and `dRows`. The List Manager will not scroll beyond the data bounds of the list.

If the automatic drawing mode is enabled, `LScroll` does all necessary updating of the list.

SPECIAL CONSIDERATIONS

You should not call the `LScroll` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LScroll` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0050</code>

Searching a List for a Particular Item

You can use the `LSearch` function to search a list for a particular item.

LSearch

You can use the `LSearch` function to find a cell whose data matches data that you specify.

```
FUNCTION LSearch (dataPtr: Ptr; dataLen: Integer; searchProc: Ptr;
                 VAR theCell: Cell; lHandle: ListHandle)
                 : Boolean;
```

<code>dataPtr</code>	A pointer to the data being searched for.
<code>dataLen</code>	The length in bytes of the data being searched for.
<code>searchProc</code>	A pointer to a function to be used to compare the data being searched for with cell data. If <code>NIL</code> , the Text Utilities Package function <code>IUMagIDString</code> is used.
<code>theCell</code>	The first cell to be searched. If <code>LSearch</code> finds a match, it returns in this parameter the coordinates of the first cell whose data matches the data being searched for.
<code>lHandle</code>	The list to be searched.

DESCRIPTION

Your application can use the `LSearch` function to search the list specified by the `lHandle` parameter beginning at the cell specified by the parameter `theCell` for a match. If `LSearch` finds a match, it returns `TRUE` and sets the parameter `theCell` to the coordinates of the first cell whose data matches the data specified by the `dataPtr` and `dataLen` parameters. Otherwise, `LSearch` returns `FALSE`.

The `LSearch` function determines whether a cell's data matches the search data by calling the `IUMagIDString` function, or the function specified by the `searchProc` parameter. If that function returns 0, `LSearch` has found a match; otherwise, `LSearch` checks the next cell in the list.

SPECIAL CONSIDERATIONS

You should not call the `LSearch` 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 `LSearch` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0054</code>

SEE ALSO

For examples of use of the `LSearch` function, see “Searching a List for a Particular Item” beginning on page 4-43. For information on the syntax of a custom match function, see “Match Functions” beginning on page 4-99. For information on the `IUMagIDString` function, see *Inside Macintosh: Text*.

Changing the Size of Cells and Lists

Usually, once your application creates a list, it should not need to change the cell size or the size of the list itself. However, there may be instances in which changing one or both is desirable. For example, if your list is on the lower-right side of a window that is resizable, then your application must resize the list if the window containing it is resized. Your application can do that with the `LSize` procedure. To resize the cells in a list, your application can use the `LCellSize` procedure.

LSize

You can change the size of a list by using the `LSize` procedure. Usually, you need to do this only after calling the Window Manager procedure `SizeWindow`.

```
PROCEDURE LSize (listWidth: Integer; listHeight: Integer;
                lHandle: ListHandle);
```

`listWidth` The new width (in pixels) of the list’s visible rectangle.

`listHeight`

 The new height (in pixels) of the list’s visible rectangle.

`lHandle` The list whose size is being changed.

DESCRIPTION

The `LSize` procedure adjusts the lower-right side of the list specified by the `lHandle` parameter so that the list's visible rectangle is the width and height specified by the `listWidth` and `listHeight` parameters.

Because the list's visible rectangle does not include room for the scroll bars, your application should make `listWidth` 15 pixels less than the desired width of the list if it contains a vertical scroll bar, and it should make `listHeight` 15 pixels less than the desired height of the list if it contains a horizontal scroll bar.

The contents of the list and the scroll bars are adjusted and redrawn as necessary. However, `LSize` does not draw a border around the list's rectangle. Also, it does not erase any portions of the old list that may still be visible. However, this approach should not be a problem if your application only calls `LSize` after the user resizes a window containing a list in its lower-right corner.

SPECIAL CONSIDERATIONS

You should not call the `LSize` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LSize` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0060</code>

SEE ALSO

For information on the Window Manager's `SizeWindow` procedure, see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

LCellSize

You can change the size of cells by using the `LCellSize` procedure. All cells in a list must be the same size, however.

```
PROCEDURE LCellSize (cSize: Point; lHandle: ListHandle);
```

<code>cSize</code>	The new size of each cell in the list.
<code>lHandle</code>	The list whose cells' size is being changed.

DESCRIPTION

The `LCellSize` procedure sets the `cellSize` field of the list record referenced by the `lHandle` parameter to the value of the `cSize` parameter. That is, the list's new cells will be of width `cSize.h` and of height `cSize.v`.

The `LCellSize` procedure updates the list's visible rectangle to contain cells of the specified size. However, `LCellSize` does not redraw any cells.

SPECIAL CONSIDERATIONS

You should not call the `LCellSize` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LCellSize` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0014</code>

Getting Information About Cells

The List Manager provides three routines that allow your application to obtain information related to cells. Your application can use the `LNextCell` function to find the next cell in a list; this is useful, for example, when performing some operation on all cells in a list. To find the local QuickDraw coordinates of a cell's rectangle, your application can use the `LRect` procedure. Finally, to determine the cell coordinates of the last cell clicked, your application can use the `LLastClick` function.

LNextCell

You can use the `LNextCell` function to find the next cell in a given row, in a given column, or in an entire list.

```
FUNCTION LNextCell (hNext: Boolean; vNext: Boolean;
                  VAR theCell: Cell; lHandle: ListHandle)
                  : Boolean;
```

<code>hNext</code>	A Boolean value that indicates whether <code>LNextCell</code> should check columns other than the current column.
<code>vNext</code>	A Boolean value that indicates whether <code>LNextCell</code> should check rows other than the current row.
<code>theCell</code>	The coordinates of the current cell.
<code>lHandle</code>	The list in which to find the next cell.

DESCRIPTION

The behavior of the `LNextCell` function hinges on the values of the `hNext` and `vNext` parameters.

If `hNext` is `TRUE` and `vNext` is `FALSE`, then `LNextCell` tries to find a cell whose coordinates are greater than those of the cell specified in the `theCell` parameter but that is in the same row as `theCell`. If successful, `LNextCell` sets the value of the `theCell` parameter to the first such cell and returns `TRUE`. If the cell initially specified by `theCell` is the last cell in its row, however, `LNextCell` returns `FALSE`.

If `hNext` is `FALSE` and `vNext` is `TRUE`, then `LNextCell` tries to find a cell whose coordinates are greater than those of the cell specified in the `theCell` parameter but that is in the same column as `theCell`. If successful, `LNextCell` sets the value of the `theCell` parameter to the first such cell and returns `TRUE`. If, however, the cell initially specified by `theCell` is the last cell in its column, `LNextCell` returns `FALSE`.

If both `hNext` and `vNext` are `TRUE`, then `LNextCell` tries to find a cell whose coordinates are greater than those of the cell specified in the parameter `theCell`. If successful, `LNextCell` sets the value of the `theCell` parameter to the first such cell and returns `TRUE`. If, however, the cell initially specified by `theCell` is the last cell in the list, `LNextCell` returns `FALSE`.

Finally, if both `hNext` and `vNext` are `FALSE`, `LNextCell` simply returns `FALSE`.

SPECIAL CONSIDERATIONS

You should not call the `LNextCell` 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 `LNextCell` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0048</code>

SEE ALSO

Listing 4-7 on page 4-34 and Listing 4-8 on page 4-35 show how to find the first and last selected cell in a list.

LRect

You can use the `LRect` procedure to find a rectangle that encloses a cell. Because the List Manager automatically draws cells, few applications need to call this procedure directly.

```
PROCEDURE LRect (VAR cellRect: Rect; theCell: Cell;
                lHandle: ListHandle);
```

`cellRect` The `LRect` procedure returns in this parameter the rectangle enclosing the cell, specified in local coordinates of the list's graphics port. This rectangle is not necessarily within the list's rectangle.

`theCell` The cell for which an enclosing rectangle is sought.

`lHandle` The list containing the cell specified by the parameter `theCell`.

DESCRIPTION

The `LRect` procedure calculates the coordinates of the rectangle enclosing the cell specified by the `theCell` parameter. The procedure does not check whether the cell is actually contained within the list's visible rectangle.

If the `theCell` parameter specifies cell coordinates not contained within the list, the `LRect` procedure sets the `cellRect` parameter to (0,0,0,0).

SPECIAL CONSIDERATIONS

You should not call the `LRect` procedure from within an interrupt, such as in a completion routine or VBL task.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LRect` procedure are

Trap macro	Selector
<code>_Pack0</code>	<code>\$004C</code>

LLastClick

You can use the `LLastClick` function to determine the coordinates of the last cell clicked in a particular list.

```
FUNCTION LLastClick (lHandle: ListHandle): Cell;
```

`lHandle` The list to be checked for the last cell clicked.

DESCRIPTION

The `LLastClick` function returns the cell coordinates of the last cell clicked. If the user has not clicked a cell since the creation of the list, then both the `h` and `v` fields of the cell returned contain negative numbers.

Note that the last cell clicked is not necessarily the last cell selected. The user could Shift-click in one cell and then drag the cursor to select other cells.

SPECIAL CONSIDERATIONS

You should not call the `LLastClick` 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 `LLastClick` function are

Trap macro	Selector
<code>_Pack0</code>	<code>\$0040</code>

Application-Defined Routines

The List Manager provides several ways that your application can customize its routines. First, your application can define a list definition procedure to create a list that displays cells graphically. Second, your application can create a custom match function to search for a particular item in a list. Finally, you can override the default click-loop procedure by providing a custom click-loop procedure.

List Definition Procedures

Your application can write a list definition procedure to customize list display. For example, you can write a list definition procedure to support the display of color icons. A custom list definition procedure must be compiled as a code resource of type 'LDEF' and added to the resource file of the application that needs to use it.

MyLDEF

A list definition procedure has the following syntax:

```
PROCEDURE MyLDEF (message: Integer; selected: Boolean;
                 VAR cellRect: Rect; theCell: Cell;
                 dataOffset: Integer; dataLen: Integer;
                 theList: ListHandle);
```

message A value that identifies the operation to be performed. These constants specify the four types of messages:

```
CONST
    lInitMsg      = 0;  {do any special initialization}
    lDrawMsg      = 1;  {draw the cell}
    lHiliteMsg    = 2;  {invert cell's highlight state}
    lCloseMsg     = 3;  {do any special disposal action}
```

selected A Boolean value that indicates whether the cell specified by the `theCell` parameter should be highlighted. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

cellRect The rectangle (in local coordinates of the list's graphics port) that encloses the cell specified by the `theCell` parameter. Although this parameter is defined as a `VAR` parameter, your list definition procedure must not change the coordinates of the rectangle. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

theCell The coordinates of the cell to be drawn or highlighted. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

dataOffset The location of the cell data associated with the cell specified by the `theCell` parameter. The location is specified as an offset from the beginning of the relocatable block referenced by the `cells` field of the list record. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

dataLen The length in bytes of the cell data associated with the cell specified by the `theCell` parameter. This parameter is defined only for the `lDrawMessage` and `lHiliteMsg` messages.

theList The list for which a message is being sent. Your application can access the list's list record, or it can call List Manager routines to manipulate the list.

DESCRIPTION

The List Manager calls your list definition procedure whenever an application using the procedure creates a new list with the `LNew` function, needs a cell to be drawn, needs a cell's highlighting state to be reversed, or has called the `LDispose` procedure to dispose of a list.

List Manager

In response to the `lInitMsg` message, your list definition procedure should perform any special initialization needed for a list. For example, the procedure might set fields of the list record, such as the `cellSize` and `indent` fields, to appropriate values. Your list definition procedure does not necessarily need to do anything in response to the initialization message. If it does nothing, then memory is still allocated for the list, and fields of the list record are set to the same values as they would be set to if the default list definition procedure were being used. (For more information on those values, see “About the List Manager” beginning on page 4-22.)

Your list definition procedure should draw the cell specified by the `theCell` parameter after receiving an `lDrawMsg` message. The procedure must ensure that it does not draw anywhere but within the rectangle specified by the `cellRect` parameter. If the `selected` parameter is `TRUE`, then your list definition procedure should draw the cell in its highlighted state; otherwise, it should draw the cell without highlighting. When drawing, your list definition procedure should take care not to permanently change any characteristics of the drawing environment.

Your list definition procedure should respond to the `lHiliteMsg` message by reversing the selection status of the cell contained within the rectangle specified by the `cellRect` parameter. If a cell is highlighted, your list definition procedure should remove the highlighting; if a cell is not highlighted, your list definition procedure should highlight it.

The List Manager sends your list definition procedure an `lCloseMsg` message before it disposes of a list and its data. Your list definition procedure need only respond to this message if additional memory has been allocated for the list. For example, your list definition procedure might allocate a relocatable block in response to the `lInitMsg` message. In this case, your list definition procedure would need to dispose of this relocatable block in response to the `lCloseMsg` message. Or, if your list definition procedure defines cells simply to contain pointers or handles to data stored elsewhere in memory, it would need to dispose of that memory in response to the `lCloseMsg` message.

SPECIAL CONSIDERATIONS

You must compile a list definition procedure as a resource of type 'LDEF' before it can be used by an application.

Because a list definition procedure is stored in a code resource, it cannot have its own global variables that it accesses through the A5 register. (Some development systems, however, may allow code resources to access global variables through some other register, such as A4. See your development system's documentation for more information.) If your list definition procedure needs access to global data, it might store a handle to such data in the `refCon` or `userHandle` fields of the list record; however, applications would not then be able to use these fields for their own purposes.

ASSEMBLY-LANGUAGE INFORMATION

The entry point of a list definition procedure must be at the beginning.

SEE ALSO

For an example of a list definition procedure, see “Writing Your Own List Definition Procedure” beginning on page 4-58.

Match Functions

You can pass a pointer to a custom match function as the third parameter to the `LSearch` function. Alternatively, your application can specify `NIL` to use the Text Utilities function `IUMagIDString`, the default match function.

MyMatchFunction

A match function must have the following syntax:

```
FUNCTION MyMatchFunction (cellDataPtr, searchDataPtr: Ptr;
                          cellDataLen, searchDataLen: Integer)
                          : Integer;
```

`cellDataPtr`

A pointer to the data contained in a cell.

`searchDataPtr`

A pointer to the data being searched for.

`cellDataLen`

The number of bytes of data contained in the cell specified by the `cellDataPtr` parameter.

`searchDataLen`

The number of bytes of data contained in the cell specified by the `searchDataPtr` parameter.

DESCRIPTION

A custom match function must compare the data defined by the `cellDataPtr` and `cellDataLen` parameters with the data defined by the `searchDataPtr` and `searchDataLen` parameters. If the cell data matches the search data, your match function should return 0. Otherwise, your match function should return 1.

List Manager

Your match function can use any technique you choose to compare the data. For example, your match function might consider the search data to be equivalent to the cell data if both are the same length. Or, your match function might only report a match if the search data can be found somewhere within the cell data.

The default match function, `IUMagIDString`, returns 0 if the search data exactly matches the cell data, but `IUMagIDString` considers the strings 'Rose' and 'rosé' to be equivalent. If your application simply needs a match function that works like `IUMagIDString` but considers 'Rose' to be different from 'rosé', you do not need to write a custom match function. Instead, your application can simply pass `@IUMagString` as the third parameter to the `LSearch` function.

SPECIAL CONSIDERATIONS

A custom match function does not execute at interrupt time. Instead, it is called directly by the `LSearch` function. Thus, a match function can allocate memory, and it does not need to adjust the value contained in the A5 register.

SEE ALSO

For information on the `IUMagIDString` function and the `IUMagString` function, see *Inside Macintosh: Operating System Utilities*.

For examples of match functions, see “Searching a List for a Particular Item” beginning on page 4-43.

Click-Loop Procedures

The List Manager supports the use of custom click-loop procedures to allow you to override the standard click-loop procedure that is used to select cells and automatically scroll a list. To define a custom click-loop procedure, specify a pointer to your procedure in the `lClickLoop` field of the list record. Because the `selFlags` field of the list record (described in “Customizing Cell Highlighting” beginning on page 4-38) already provides a means of customizing the algorithm the List Manager uses to highlight list cells, in most cases you should not need to define a custom click-loop procedure.

MyClickLoop

A click-loop procedure must have the following syntax:

```
PROCEDURE MyClickLoop;
```

DESCRIPTION

If your application defines a custom click-loop procedure, then the `LClick` function repeatedly calls the procedure until the user releases the mouse button. A click-loop procedure may perform any processing desired when it is executed.

Because no parameters are passed to the click-loop procedure, your click-loop procedure probably needs to access a global variable that contains a handle to the list record, which contains information about the location of the cursor and other information potentially of interest to a click-loop procedure. You might also create a global variable that stores the state of the modifier keys immediately before a call to the `LClick` function. You would need to set these global variables immediately before calling `LClick`.

A click-loop procedure should ordinarily set the Z flag to 1 just before returning. If a click-loop procedure sets the Z flag to 0, then the `LClick` function immediately returns.

SPECIAL CONSIDERATIONS

A click-loop procedure does not execute at interrupt time. Instead, it is called directly by the `LClick` function. Thus, a click-loop procedure can allocate memory, and it does not need to adjust the value contained in the A5 register.

ASSEMBLY-LANGUAGE INFORMATION

Your click-loop procedure should ordinarily set register D0 to 1. To stop the `LClick` function from calling your procedure for the current mouse-down event, set register D0 to 0.

For your convenience, register D5 contains the current mouse location.

Summary of the List Manager

Pascal Summary

Constants

CONST

```

{masks for listFlags field of list record}
lDoVAutoScroll = 2;           {allow vertical autoscrolling}
lDoHAutoScroll = 1;           {allow horizontal autoscrolling}

{masks for selFlags field of list record}
lOnlyOne       = -128;        {allow only one item to be selected at once}
lExtendDrag    = 64;          {enable multiple item selection without Shift}
lNoDisjoint    = 32;          {prevent discontinuous selections}
lNoExtend      = 16;          {reset list before responding to Shift-click}
lNoRect        = 8;           {Shift-drag selects items passed by cursor}
lUseSense      = 4;           {allow use of Shift key to deselect items}
lNoNilHilite   = 2;           {disable highlighting of empty cells}

{messages to list definition procedure}
lInitMsg       = 0;           {do any special list initialization}
lDrawMsg       = 1;           {draw the cell}
lHiliteMsg     = 2;           {invert cell's highlight state}
lCloseMsg      = 3;           {take any special disposal action}

```

Data Types

TYPE

```

Cell = Point;                  {cell.v contains row number}
                                   {cell.h contains column number}

DataArray = PACKED ARRAY[0..32000] OF Char;
DataPtr   = ^DataArray;
DataHandle = ^DataPtr;

```

List Manager

```

ListRec =
RECORD
    rView:      Rect;           {list's display rectangle}
    port:       GrafPtr;       {list's graphics port}
    indent:     Point;         {indent distance for drawing}
    cellSize:   Point;         {size in pixels of a cell}
    visible:    Rect;          {boundary of visible cells}
    vScroll:    ControlHandle; {vertical scroll bar}
    hScroll:    ControlHandle; {horizontal scroll bar}
    selFlags:   SignedByte;    {selection flags}
    lActive:    Boolean;       {TRUE if list is active}
    lReserved:  SignedByte;    {reserved}
    listFlags:  SignedByte;    {automatic scrolling flags}
    clikTime:   LongInt;       {TickCount at time of last click}
    clikLoc:    Point;         {position of last click}
    mouseLoc:   Point;         {current mouse location}
    lClikLoop:  Ptr;           {routine called by LClik}
    lastClick:  Cell;          {last cell clicked}
    refCon:     LongInt;       {for application use}
    listDefProc:
                    Handle;    {list definition procedure}
    userHandle: Handle;        {for application use}
    dataBounds: Rect;         {boundary of cells allocated}
    cells:      DataHandle;    {cell data}
    maxIndex:   Integer;       {used internally}
    cellArray:  ARRAY[1..1] OF Integer;
END;

ListPtr      = ^ListRec;      {pointer to a list record}
ListHandle   = ^ListPtr;     {handle to a list record}

```

List Manager Routines

Creating and Disposing of Lists

```

FUNCTION LNew (rView: Rect; dataBounds: Rect; cSize: Point;
              theProc: Integer; theWindow: WindowPtr;
              drawIt, hasGrow, scrollHoriz,
              scrollVert: Boolean): ListHandle;

PROCEDURE LDispose (lHandle: ListHandle);

```

Adding and Deleting Columns and Rows To and From a List

```

FUNCTION LAddColumn      (count: Integer; colNum: Integer;
                        lHandle: ListHandle): Integer;

FUNCTION LAddRow        (count: Integer; rowNum: Integer;
                        lHandle: ListHandle): Integer;

PROCEDURE LDelColumn   (count: Integer; colNum: Integer;
                        lHandle: ListHandle);

PROCEDURE LDelRow     (count: Integer; rowNum: Integer;
                        lHandle: ListHandle);

```

Determining or Changing the Selection

```

FUNCTION LGetSelect     (next: Boolean; VAR theCell: Cell;
                        lHandle: ListHandle): Boolean;

PROCEDURE LSetSelect   (setIt: Boolean; theCell: Cell;
                        lHandle: ListHandle);

```

Accessing and Manipulating Cell Data

```

PROCEDURE LSetCell     (dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                        lHandle: ListHandle);

PROCEDURE LAddToCell  (dataPtr: Ptr; dataLen: Integer; theCell: Cell;
                        lHandle: ListHandle);

PROCEDURE LClrCell    (theCell: Cell; lHandle: ListHandle);
{the LGetCellDataLocation procedure is also available as the LFind procedure}
PROCEDURE LGetCellDataLocation
                        (VAR offset, len: Integer; theCell: Cell;
                        lHandle: ListHandle);

PROCEDURE LGetCell    (dataPtr: Ptr; VAR dataLen: Integer;
                        theCell: Cell; lHandle: ListHandle);

```

Responding to Events Affecting Lists

```

FUNCTION LClick        (pt: Point; modifiers: Integer;
                        lHandle: ListHandle): Boolean;

PROCEDURE LActivate   (act: Boolean; lHandle: ListHandle);

PROCEDURE LUpdate     (theRgn: RgnHandle; lHandle: ListHandle);

```

Modifying a List's Appearance

{the LSetDrawingMode procedure is also available as the LDoDraw procedure}

```
PROCEDURE LSetDrawingMode (drawIt: Boolean; lHandle: ListHandle);
PROCEDURE LDraw (theCell: Cell; lHandle: ListHandle);
PROCEDURE LAutoScroll (lHandle: ListHandle);
PROCEDURE LScroll (dCols: Integer; dRows: Integer;
                  lHandle: ListHandle);
```

Searching a List for a Particular Item

```
FUNCTION LSearch (dataPtr: Ptr; dataLen: Integer;
                searchProc: Ptr; VAR theCell: Cell;
                lHandle: ListHandle): Boolean;
```

Changing the Size of Cells and Lists

```
PROCEDURE LSize (listWidth: Integer; listHeight: Integer;
               lHandle: ListHandle);
PROCEDURE LCellSize (cSize: Point; lHandle: ListHandle);
```

Getting Information About Cells

```
FUNCTION LNextCell (hNext: Boolean; vNext: Boolean;
                  VAR theCell: Cell;
                  lHandle: ListHandle): Boolean;
PROCEDURE LRect (VAR cellRect: Rect; theCell: Cell;
               lHandle: ListHandle);
FUNCTION LLastClick (lHandle: ListHandle): Cell;
```

Application-Defined Routines

```
PROCEDURE MyLDEF (message: Integer; selected: Boolean;
                VAR cellRect: Rect; theCell: Cell;
                dataOffset: Integer; dataLen: Integer;
                theList: ListHandle);
FUNCTION MyMatchFunction (cellDataPtr, searchDataPtr: Ptr;
                        cellDataLen, searchDataLen: Integer): Integer;
PROCEDURE MyClickLoop;
```

C Summary

Constants

```

/*masks for listFlags field of list record*/
enum {
    lDoVAutoScroll = 2,      /*allow vertical autoscrolling*/
    lDoHAutoScroll = 1,      /*allow horizontal autoscrolling*/

/*masks for selFlags field of list record*/
    lOnlyOne = -128,         /*allow only one item to be selected at once*/
    lExtendDrag = 64,        /*enable multiple item selection without Shift*/
    lNoDisjoint = 32,        /*prevent discontinuous selections*/
    lNoExtend = 16,          /*reset list before responding to Shift-click*/
    lNoRect = 8,             /*Shift-drag selects items passed by cursor*/
    lUseSense = 4,           /*allow use of Shift key to deselect items*/
    lNoNilHilite = 2,        /*disable highlighting of empty cells*/

/*messages to list definition procedure*/
    lInitMsg = 0,            /*do any special list initialization*/
    lDrawMsg = 1,            /*draw the cell*/
    lHiliteMsg = 2,          /*invert cell's highlight state*/
    lCloseMsg = 3            /*take any special disposal action*/
};

```

Data Types

```

typedef Point Cell;          /*cell.v contains row number*/
                             /*cell.h contains column number*/

typedef char DataArray[32001], *DataPtr, **DataHandle;

```

List Manager

```

struct ListRec {
    Rect        rView;           /*list's display rectangle*/
    GrafPtr     ptr;            /*list's graphics port*/
    Point       indent;         /*indent distance for drawing*/
    Point       cellSize;       /*size in pixels of a cell*/
    Rect        visible;        /*boundary of visible cells*/
    ControlHandle vScroll;      /*vertical scroll bar*/
    ControlHandle hScroll;      /*horizontal scroll bar*/
    char        selFlags;       /*selection flags*/
    Boolean      lActive;        /*TRUE if list is active*/
    char        lReserved;      /*reserved*/
    char        listFlags;      /*automatic scrolling flags*/
    long        klikTime;       /*TickCount at time of last click*/
    Point       klikLoc;        /*position of last click*/
    Point       mouseLoc;       /*current mouse location*/
    ProcPtr     lClikLoop;      /*routine called by LClik*/
    Cell        lastClick;      /*last cell clicked*/
    long        refCon;         /*for application use*/
    Handle      listDefProc;    /*list definition procedure*/
    Handle      userHandle;     /*for application use*/
    Rect        dataBounds;     /*boundary of cells allocated*/
    DataHandle  cells;          /*cell data*/
    short       maxIndex;       /*used internally*/
    short       cellArray[1];   /*offsets to data*/
};

```

```

typedef struct ListRect ListRect;
typedef ListRect *ListPtr, **ListHandle;

```

List Manager Routines

Creating and Disposing of Lists

```

pascal ListHandle LNew      (const Rect *rView, Rect *dataBounds,
                             Point *cSize, short theProc,
                             WindowPtr theWindow, Boolean drawIt,
                             Boolean hasGrow, Boolean scrollHoriz,
                             Boolean scrollVert);

pascal void LDispose      (ListHandle lHandle);

```

Adding and Deleting Columns and Rows To and From a List

```

pascal short LAddColumn      (short count, short colNum, ListHandle lHandle);
pascal short LAddRow        (short count, short rowNum, ListHandle lHandle);
pascal void LDelColumn      (short count, short colNum, ListHandle lHandle);
pascal void LDelRow         (short count, short rowNum, ListHandle lHandle);

```

Determining or Changing the Selection

```

pascal Boolean LGetSelect    (Boolean next, Cell *theCell,
                             ListHandle lHandle);
pascal void LSetSelect      (Boolean setIt, Cell theCell,
                             ListHandle lHandle);

```

Accessing and Manipulating Cell Data

```

pascal void LSetCell        (const void *dataPtr, short dataLen,
                             Cell theCell, ListHandle lHandle);
pascal void LAddToCell      (const void *dataPtr, short dataLen,
                             Cell theCell, ListHandle lHandle);
pascal void LClrCell        (Cell theCell, ListHandle lHandle);
/*the LGetCellDataLocation procedure is also available as */
/* the LFind procedure*/
pascal void LGetCellDataLocation
                             (short *offset, short *len, Cell theCell,
                             ListHandle lHandle);
pascal void LGetCell        (void *dataPtr, short *dataLen, Cell theCell,
                             ListHandle lHandle);

```

Responding to Events Affecting Lists

```

pascal Boolean LClick       (Point pt, short modifiers, ListHandle lHandle);
pascal void LActivate      (Boolean act, ListHandle lHandle);
pascal void LUpdate        (RgnHandle theRgn, ListHandle lHandle);

```

Modifying a List's Appearance

```

/*the LSetDrawingMode procedure is also available as the LDoDraw procedure*/
pascal void LSetDrawingMode
                                (Boolean drawIt, ListHandle lHandle);
pascal void LDraw                (Cell theCell, ListHandle lHandle);
pascal void LAutoScroll          (ListHandle lHandle);
pascal void LScroll             (short dCols, short dRows, ListHandle lHandle);

```

Searching for a List Containing a Particular Item

```

pascal Boolean LSearch          (const void *dataPtr, short dataLen,
                                SearchProcPtr searchProc, Cell *theCell,
                                ListHandle lHandle);

```

Changing the Size of Cells and Lists

```

pascal void LSize              (short listWidth, short listHeight,
                                ListHandle lHandle);
pascal void LCellSize          (Point cSize, ListHandle lHandle);

```

Getting Information About Cells

```

pascal Boolean LNextCell       (Boolean hNext, Boolean vNext, Cell *theCell,
                                ListHandle lHandle);
pascal void LRect              (Rect *cellRect, Cell theCell,
                                ListHandle lHandle);
pascal Cell LLastClick         (ListHandle lHandle);

```

Application-Defined Routines

```

pascal void MyLDEF             (short message, Boolean selected,
                                Rect *cellRect, Cell theCell,
                                short dataOffset, short dataLen,
                                ListHandle theList);
pascal short MyMatchFunction   (Ptr cellDataPtr, Ptr searchDataPtr,
                                short cellDataLen, short searchDataLen);
pascal void MyClickLoop        (void);

```

Assembly-Language Summary

Data Structures

ListRect Data Structure

0	rView	8 bytes	list's display rectangle
8	port	long	list's graphics port
12	indent	4 bytes	indent distance for drawing
16	cellSize	4 bytes	size in pixels of a cell
20	visible	8 bytes	boundary of visible cells
28	vScroll	long	vertical scroll bar
32	hScroll	long	horizontal scroll bar
36	selFlags	byte	selection flags
37	lActive	byte	nonzero if list is active
38	lReserved	byte	reserved
39	listFlags	byte	automatic scrolling flags
40	clikTime	long	ticks at time of last click
44	clikLoc	4 bytes	position of last click
48	mouseLoc	4 bytes	current mouse location
52	lClikLoop	long	pointer to routine called by LClick
56	lastClick	4 bytes	last cell clicked
60	refCon	long	for application use
64	listDefProc	long	handle to code for list definition procedure
68	userHandle	long	for application use
72	dataBounds	8 bytes	boundary of cells allocated
80	cells	long	handle to cell data
84	maxIndex	word	used internally
86	cellArray	variable	offsets to data

Trap Macros

Trap Macros Requiring Routine Selectors`_Pack0`

Selector	Routine
\$0000	LActivate
\$0004	LAddColumn
\$0008	LAddRow
\$000C	LAddToCell
\$0010	LAutoScroll
\$0014	LCellSize
\$0018	LClick
\$001C	LClrCell
\$0020	LDelColumn
\$0024	LDelRow
\$0028	LDispose
\$002C	LSetDrawingMode
\$0030	LDraw
\$0034	LGetCellDataLocation
\$0038	LGetCell
\$003C	LGetSelect
\$0040	LLastClick
\$0044	LNew
\$0048	LNextCell
\$004C	LRect
\$0050	LScroll
\$0054	LSearch
\$0058	LSetCell
\$005C	LSetSelect
\$0060	LSize
\$0064	LUpdate

Icon Utilities

Contents

Introduction to the Icon Utilities	5-3
About the Icon Utilities	5-6
Using the Icon Utilities	5-7
Drawing Icons in an Icon Family	5-8
Drawing an Icon Directly From a Resource	5-10
Getting an Icon Suite and Drawing One of Its Icons	5-11
Drawing Specific Icons From an Icon Family	5-12
Manipulating Icons	5-13
Drawing Icons That Are Not Part of an Icon Family	5-13
Icon Utilities Reference	5-17
Data Structure	5-17
The Color Icon Record	5-17
Icon Utilities Routines	5-18
Drawing Icons From Resources	5-19
Getting Icons From Resources That Don't Belong to an Icon Family	5-28
Disposing of Icons	5-30
Creating an Icon Suite	5-30
Getting Icons From an Icon Suite	5-34
Drawing Icons From an Icon Suite	5-35
Performing Operations on Icons in an Icon Suite	5-38
Getting and Setting the Label for an Icon Suite	5-40
Getting Label Information	5-41
Disposing of Icon Suites	5-42
Converting an Icon Mask to a Region	5-43
Determining Whether a Point or Rectangle Is Within an Icon	5-46
Working With Icon Caches	5-53
Application-Defined Routines	5-57
Icon Action Functions	5-57
Icon Getter Functions	5-58

CHAPTER 5

Summary of the Icon Utilities	5-60
Pascal Summary	5-60
Constants	5-60
Data Types	5-62
Icon Utilities Routines	5-62
Application-Defined Routines	5-65
C Summary	5-65
Constants	5-65
Data Types	5-67
Icon Utilities Routines	5-68
Application-Defined Routines	5-71
Assembly-Language Summary	5-71
Data Structure	5-71
Trap Macros	5-72
Result Codes	5-73

This chapter describes how your application can use the Icon Utilities to draw icons, including small, large, black-and-white, and color icons. The Finder draws and manages the icons that a user sees on the desktop, but if your application needs to display icons within its windows, it can use Icon Utilities routines to draw them.

For information on how to create icons and associate them with your application and its document, see the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*. For information on how to design icons, see the chapter “Icons” in *Macintosh Human Interface Guidelines*.

This chapter begins with a brief overview of the various kinds of icons you can provide. The rest of the chapter describes how you can draw each kind of icon.

Introduction to the Icon Utilities

An **icon** on a Macintosh screen is an image that graphically represents some object, such as a file, a folder, or the Trash. On the desktop, the Finder displays icons representing your application and the documents it creates. The Finder also allows users to manipulate icons on the desktop and in folders.

If necessary, your application can also display icons in its menus, dialog boxes, or windows. You define an icon for a menu item by providing the icon's icon number in the 'MENU' resource that describes the menu item. If you define an icon for a menu item in this manner, the Menu Manager automatically displays the icon whenever you display the menu using the `MenuSelect` function.

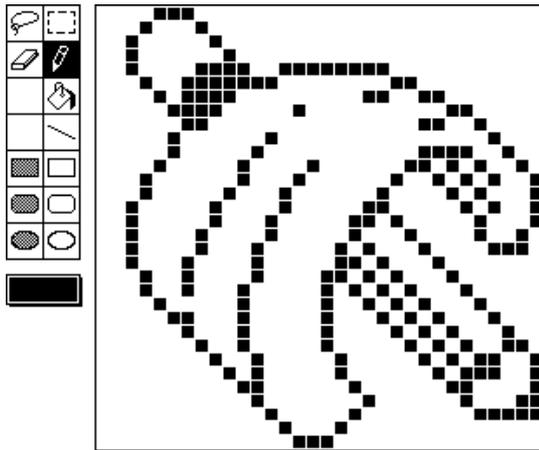
You usually define icons in dialog boxes by defining an item of type `Icon` and providing the resource ID of the icon in the item list ('DITL') resource that describes the dialog. If you define an icon for a dialog item in this manner, the Dialog Manager automatically displays the icon whenever you display the dialog box using Dialog Manager routines.

Both the Menu Manager and Dialog Manager allow you to display icons of resource type 'ICON' or 'cicn'. The Menu Manager also allows you to display icons of resource type 'SICN'. To display other types of icons in your menu items, you can write your own menu definition procedure and use the routines described in this chapter to draw the icons. To display other types of icons in your dialog items, define items of type `userItem` and use the routines in this chapter to draw your icons.

To display icons of any kind in your windows, use Icon Utilities routines. Icons in windows can be useful for representing files and folders in certain applications, such as archiving applications, groupware, and electronic mail applications. Other programs, such as games, might allow users to move or manipulate icons in windows for a variety of purposes.

Whenever you design an icon, you should generally begin by creating a black-and-white icon and then add color using the resource types that define color icons. Typically you use a high-level tool such as the ResEdit application to design icons. Figure 5-1 shows the ResEdit view of a black-and-white icon. When you are satisfied with the appearance of your icons, you can use the DeRez decompiler to convert them into Rez input.

Figure 5-1 The ResEdit view of an icon



For more information about designing and creating icons, see *Macintosh Human Interface Guidelines* and the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.

To display an icon most effectively at different sizes and on display devices with different bit depths, you should create an icon family for each icon you wish to use. An **icon family** is the set of icons that represent a single object. 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 icon:

- n a large (32-by-32 pixel) black-and-white icon and mask—both of which you define in an icon list ('ICN#') resource
- n a small (16-by-16 pixel) black-and-white icon and mask—both of which you define in a small icon list ('ics#') resource
- n 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 ('ic14') resource

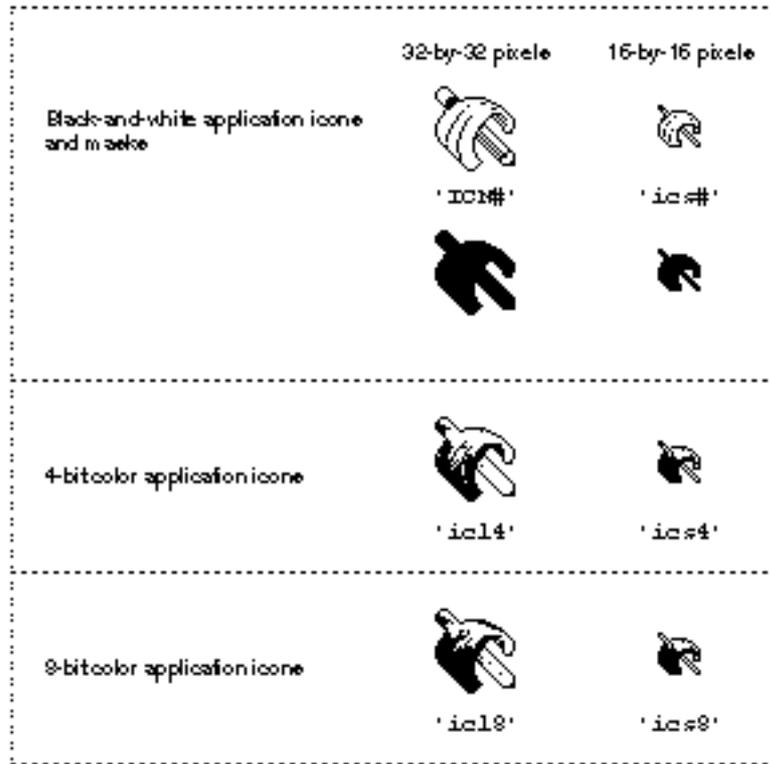
Icon Utilities

- n 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
- n 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
- n 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

An icon family can contain only one icon of each resource type listed.

Figure 5-2 shows the icon family for the icon that represents the SurfWriter application. To see these icons in color, see Plate 3 in *Inside Macintosh: Macintosh Toolbox Essentials*.

Figure 5-2 An icon family



Icon Utilities

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. As previously discussed, both the Menu Manager and Dialog Manager allow you to display icons with the resource type 'ICON' or 'cicn', and the Menu Manager also allows you to display icons of resource type 'SICN'. These are the only kinds of icons you can use in menu items and dialog boxes if you want the Menu Manager and Dialog Manager to display the icons automatically for you. If you provide a color icon ('cicn') resource with the same resource ID as an icon ('ICON') resource, the Menu Manager and the Dialog Manager display the color icon instead of the black-and-white icon.

The icon ('ICON') resource contains a bitmap for a 32-by-32 pixel black-and-white icon. Because it is always displayed on a white background, and never in the Finder, it doesn't need a mask.

The color icon ('cicn') resource has a special format that includes a pixel map, a bitmap, and a mask. You can use it to define a color icon of any size without a mask or a 32-by-32 pixel color icon with a mask. You can also define the bit depth for a color icon resource. For information about the format of a 'cicn' resource, see *Inside Macintosh: Imaging With QuickDraw*.

Many of the icons in the System file are available in a small size; these icons are stored in 'SICN' resources. The icons in an 'SICN' resource are 12 by 16 pixels, even though they are stored in the resource as 16-by-16 pixel bitmaps. An 'SICN' resource consists of a list of 16-by-16 pixel bitmaps for black-and-white icons; by convention, the list includes only two bitmaps, and the second bitmap is considered a mask. The Menu Manager lets you use an 'SICN' resource as an icon in a menu item; however, you cannot use the Dialog Manager to display an 'SICN' icon in a dialog box.

The Finder does *not* use or display any resources that you create of type 'ICON', 'cicn', or 'SICN'. To create an icon for display by the Finder, create one or more of the icons in an icon family.

About the Icon Utilities

The Icon Utilities allow your application (and system software) to manipulate and draw icons of any standard resource type in windows and if necessary in menus or dialog boxes. You need to use these routines only if you wish to draw icons in your application's windows or to draw icons whose resource types are not recognized by the Menu Manager and Dialog Manager in menus and dialog boxes.

To display an icon most effectively at a variety of sizes and bit depths, you should provide an icon family. You can then draw the appropriate member of the family for a given size and bit depth either by passing the family's resource ID to an Icon Utilities routine or by reading the family's icon resources into memory as an icon suite and passing the suite's handle to Icon Utilities routines.

Icon Utilities

The next section, “Using the Icon Utilities,” begins by describing how to draw icons in an icon family. After a brief overview of icon families, icon suites, icon caches, and related Icon Utilities routines, it describes in detail how to

- n draw the most appropriate icon for a given destination rectangle and bit depth directly from an icon family member’s resource
- n get an icon suite and draw the most appropriate icon from that suite for a given destination rectangle and bit depth
- n draw specific icons from an icon family or suite
- n get a handle to an icon suite member’s icon data so you can manipulate it
- n draw icons that are not part of an icon family

You can use also Icon Utilities routines to

- n perform operations on icons in an icon suite
- n manipulate labels associated with specific icon suites
- n dispose of icon suites and color icon records
- n convert an icon mask to a region and perform hit-testing for an icon
- n create an icon cache by associating an icon suite with an icon getter function and a pointer to data that you can use as a reference constant

For detailed descriptions of all Icon Utilities routines, including those used to perform these tasks, see “Icon Utilities Reference” beginning on page 5-17.

In addition to the resource types described earlier in this chapter, some Icon Utilities routines operate on icons of resource types `'icm#'`, `'icm4'`, and `'icm8'`. These **mini icons** are 12-by-16 pixel icons. Like the icons in an icon family, the three resource types for mini icons identify the icon list, 4-bit color icons, and 8-bit color icons, respectively.

Using the Icon Utilities

This section explains how you can use routines in the Icon Utilities to draw icons in your application’s windows (or dialog boxes and menu items if needed).

Most of the Icon Utilities routines are available only in System 7 and later. To determine whether they are available, call the `Gestalt` function with the `gestaltIconUtilitiesAttr` selector and check the value of the response parameter. If the bit indicated by the constant `gestaltIconUtilitiesPresent` is set, then the Icon Utilities are available.

```
CONST
gestaltIconUtilitiesAttr    = 'icon';    {Icon Utils attributes}
gestaltIconUtilitiesPresent = 0;        {check this bit in the }
                                { response parameter}
```

The `GetIcon`, `PlotIcon`, `GetCIcon`, `PlotCIcon`, and `DisposeCIcon` routines are available in both System 6 and System 7.

Drawing Icons in an Icon Family

You can define different versions of an icon for specific sizes and bit depths as part of a single icon family whose members share the same resource ID. If you define all your application's icons in icon families, you can use Icon Utilities routines to draw the icon using the icon family member that is best suited for the destination rectangle and the current bit depth of the display device. When your application uses Icon Utilities routines like `PlotIconSuite` or `PlotIconID` to plot icons, it doesn't have to determine which icon in the icon family is best suited for a given destination rectangle and bit depth; instead, the routines automatically display the appropriate icon.

You can also define individual icons of resource type `'ICON'`, `'cicn'`, or `'SICN'` that are not part of an icon family and use Icon Utilities routines to draw them when necessary. For information about drawing these types of icons, see the section "Drawing Icons That Are Not Part of an Icon Family" beginning on page 5-13.

You can use the Icon Utilities to draw icons using modes or **transforms** that alter the icon's appearance in standard ways that are analogous to Finder states for icons. For example, the Finder draws a selected icon differently than it draws one that is not selected; to do so, the Finder specifies the transform constant `ttSelected` when it calls Icon Utilities routines to draw a selected icon. If you need to apply a particular transform to an icon, some Icon Utilities routines allow you to apply transforms for both standard Finder states and Finder label colors when you draw the icon.

Many of the Icon Utilities routines can also automatically align an icon within its destination rectangle. For example, the generic document icon that appears in the Finder is taller than it is wide. Some Icon Utilities routines allow you to draw such an icon without any special alignment, align it at the left or right of the destination rectangle, or use various other alignments.

Depending on the size of the rectangle, the Icon Utilities routines may stretch or shrink the icon to fit. To draw icons without stretching them, these routines require that the destination rectangle have the exact dimensions of a standard icon: that is, depending on the icon resource type, 32 by 32 pixels, 16 by 16 pixels, or 12 by 16 pixels. If you use destination rectangles of other sizes, these routines stretch or shrink the icons to fit the rectangles.

An icon family is a collection of icons representing a single object. Each icon in the family shares the same resource ID as other icons in the family but has its own resource type identifying the icon data it contains. The simplest way to draw an icon from an icon family is to pass the family's resource ID to the `PlotIconID` function, which draws the appropriate icon from the family for the specified destination rectangle and bit depth. The next section, "Drawing an Icon Directly From a Resource," describes how to use `PlotIconID`.

Alternatively, you can first use the `GetIconSuite` function to read the resource data for some or all icons in an icon family into memory. Given a resource ID and one or more resource types, the `GetIconSuite` function reads in resource data for each icon with the specified resource ID and resource types and collects handles to the resource data in an icon suite. An **icon suite** typically consists of one or more handles to icon resources from a single icon family that have been read into memory. The `GetIconSuite` function returns a handle for the requested icon suite; you can pass this handle to `PlotIconSuite` and other Icon Utilities routines. Like `PlotIconID`, `PlotIconSuite` draws the appropriate icon from an icon suite for the specified destination rectangle and bit depth. The section “Getting an Icon Suite and Drawing One of Its Icons” on page 5-11 describes how to use the `GetIconSuite` and `PlotIconSuite` routines.

An icon suite can in turn contain handles to each of the six icon resources that an icon family can contain, or it can contain handles to only a subset of the icon resources in an icon family. However, for best results, an icon suite should always include a black-and-white icon and icon mask for any icons you provide; that is, it should include a resource of type `'ICN#'` in addition to any other large icons you provide as well as a resource of type `'ics#'` in addition to any other small icons you provide. When you create an icon suite from icon family resources, the associated resource file should remain open while you use Icon Utilities routines.

Two types of handles exist in an icon suite: handles to icon data associated with a resource and handles to icon data that isn't associated with a resource. You typically use `GetIconSuite` to fill an icon suite with handles to icon resource data. You typically use `AddIconToSuite` to add to an icon suite handles to icon data. When you use `AddIconToSuite`, the handles that you add to the suite do not have to be associated with a resource fork. For example, your application might get icon data from the desktop database rather than reading it from a resource, or your application might read icon data from a resource and then detach it. In either case, you can provide a handle to the icon data and use `AddIconToSuite` to add the handle to the icon suite.

An **icon cache** is like an icon suite, except that an icon cache also contains a pointer to an application-defined **icon getter function** and a pointer to data that is associated with the icon suite. You can pass a handle to an icon cache to any of the Icon Utilities routines that accept a handle to an icon suite. An icon cache typically does not contain handles to the icon resources for all icon family members. Instead, if the icon cache does not contain an entry for a specific type of icon in an icon family, the Icon Utilities routines call your application's icon getter function to retrieve the data for that icon type. The icon getter function should return either a handle to the icon data or `NIL` to indicate that no icon data exists for the specified icon type.

Drawing an Icon Directly From a Resource

To draw an icon from an icon family without first creating an icon suite, use the `PlotIconID` function. Listing 5-1 shows an application-defined procedure that draws an icon from an icon family. Given a resource ID, the `PlotIconID` function determines which member of the icon family to draw and then draws the icon in the given rectangle with the specified transform and alignment.

Listing 5-1 Drawing the icon from an icon family that is best suited to the user's display

```
PROCEDURE MyDrawIconFromFamily (resID: Integer; destRect: Rect);
VAR
    align:          IconAlignmentType;
    transform:      IconTransformType;
    myErr:          OSErr;
BEGIN
    align := atAbsoluteCenter; {specify alignment (centered)}
    transform := ttNone;      {specify no special transforms}
    {draw the icon, using the icon type best suited for the }
    { destination rect and current bit depth of the display device}
    myErr := PlotIconID(destRect, align, transform, resID);
END;
```

The `PlotIconID` function determines, from the size of the specified destination rectangle and the current bit depth of the display device, which icon of a given size from an icon family to draw. For example, if the coordinates of the destination rectangle are (100,100,116,116) and the display device is set to 4-bit color, the `PlotIconID` function draws the icon of type 'ics4' if that icon is available in the icon family.

If the width or height of a destination rectangle is greater than or equal to 32, `PlotIconID` uses the 32-by-32 pixel icon with the appropriate bit depth for the display device. If the destination rectangle is less than 32 by 32 pixels and greater than 16 pixels wide or 12 pixels high, `PlotIconID` uses the 16-by-16 pixel icon with the appropriate bit depth. If the destination rectangle's height is less than or equal to 12 pixels or its width is less than or equal to 16 pixels, `PlotIconID` uses the 12-by-16 pixel icon with the appropriate bit depth. (Typically only the Finder and the Standard File Package use 12-by-16 pixel icons.)

Depending on the size of the rectangle, the `PlotIconID` function may stretch or shrink the icon to fit. To draw icons without stretching them, `PlotIconID` requires that the destination rectangle have the exact dimensions of a standard icon: that is, depending on the icon resource type, 32 by 32 pixels, 16 by 16 pixels, or 12 by 16 pixels. If you use destination rectangles of other sizes, `PlotIconID` stretches or shrinks the icons to fit the rectangles.

Getting an Icon Suite and Drawing One of Its Icons

Listing 5-2 shows how you can use the `GetIconSuite` and `PlotIconSuite` functions to get an icon suite and then draw the icon from the suite that is best suited to the destination rectangle and the current bit depth of the display device.

Listing 5-2 Drawing the icon from an icon suite that is best suited to the display device

```
PROCEDURE MyDrawIconInSuite (resID: Integer; destRect: Rect;
                             VAR iconSuiteHdl: Handle);
VAR
  iconType:      IconSelectorValue;
  align:         IconAlignmentType;
  transform:     IconTransformType;
  myErr:        OSErr;
BEGIN
  iconType := svAllAvailable;  {get all icons in icon family}
  myErr := GetIconSuite(iconSuiteHdl, resID, iconType);
  IF iconSuiteHdl <> NIL THEN
    BEGIN
      align := atAbsoluteCenter; {specify alignment (centered)}
      transform := ttNone;      {specify no special transforms}
      {draw the icon, using the icon type best suited for the }
      { destination rect & current bit depth of display device}
      myErr := PlotIconSuite(destRect, align, transform,
                             iconSuiteHdl);

    END;
  END;
```

The application-defined procedure `MyDrawIconInSuite` shown in Listing 5-2 first uses the `GetIconSuite` function, specifying the constant `svAllAvailable` in the third parameter, to get all icons from the icon family with the specified resource ID and to collect the handles to the data for each icon into an icon suite. (You can use other constants in the third parameter of `GetIconSuite` to request only certain members of an icon family for an icon suite.) The `MyDrawIconInSuite` procedure then draws an icon from this suite using the `PlotIconSuite` function.

Like the `PlotIconID` function described in the previous section, the `PlotIconSuite` function determines, from the size of the specified destination rectangle and the current bit depth of the display device, which icon from the icon suite to draw.

You can also specify various transforms and alignments to `PlotIconSuite`. For example, the code in Listing 5-2 specifies that `PlotIconSuite` should center the icon within the destination rectangle.

Drawing Specific Icons From an Icon Family

In most cases you should use `PlotIconID` or `PlotIconSuite` to draw an icon from an icon family, because these routines automatically select the best version of an icon to display for a given destination rectangle and bit depth. The preceding sections, “Drawing an Icon Directly From a Resource” and “Getting an Icon Suite and Drawing One of Its Icons,” describe how to use these routines.

If you need to plot a specific icon from an icon family rather than using the Icon Utilities to select a family member, you must first create an icon suite that contains only the icon from the desired resource type and its corresponding mask. You can then use `PlotIconSuite` to plot the icon. In this case `PlotIconSuite` still attempts to use the best icon available for the given destination rectangle and bit depth; however, by limiting the icon resources available in the icon suite, you can force `PlotIconSuite` to plot either the black-and-white icon from the 'ICN#' resource or just one of the other available resources. Listing 5-3 demonstrates how to do this.

Listing 5-3 Drawing a specific icon from an icon family or icon suite

```
PROCEDURE MyDrawThisIcon (destRect: Rect; resID: Integer;
                        VAR iconSuiteHdl: Handle);
VAR
    align:           IconAlignmentType;
    transform:       IconTransformType;
    myErr:           OSErr;
BEGIN
    {get only the 'ICN#' and 'icl4' icons and collect them in an }
    { icon suite}
    myErr := GetIconSuite(iconSuiteHdl, resID,
                        svLarge1Bit + svLarge4Bit);
    IF iconSuiteHdl <> NIL THEN
        BEGIN
            align := atAbsoluteCenter; {specify alignment (centered)}
            transform := ttNone;       {specify no special transforms}
            {draw the best icon from the suite referenced by the icon }
            { suite handle; since the suite contains only 'ICN#' and }
            { 'icl4' icons, PlotIconSuite draws the best of the two}
            myErr := PlotIconSuite(destRect, align, transform,
                                iconSuiteHdl);
        END;
    END;
```

The application-defined procedure `MyDrawThisIcon` passes the constants `svLarge1Bit` and `svLarge4Bit` to `GetIconSuite`. In response, `GetIconSuite` reads only the 'ICN#' and 'icl4' resources into memory, storing handles to the icon

Icon Utilities

resource data in the icon suite. `MyDrawThisIcon` then uses `PlotIconSuite` to plot the best available icon from the suite.

If the bit depth of the display device is 1, the `PlotIconSuite` function in Listing 5-3 displays the black-and-white version of the icon from the `'ICN#'` resource, regardless of the size of the destination rectangle. If the bit depth of the display device is greater than 1, `PlotIconSuite` draws the icon from the `'ic14'` resource, regardless of the size of the destination rectangle.

Manipulating Icons

You can use the `GetIconFromSuite` function to get a handle to the pixel data for a specific icon from an icon suite. You can use the handle returned by the function `GetIconFromSuite` to manipulate the icon data—for example, to alter its color or add three-dimensional shading—but not to draw the icon with other Icon Utilities routines such as `PlotIconHandle`.

Listing 5-4 provides an example of an application-defined procedure, `MyGetIconData`, that calls `GetIconFromSuite` and manipulates the icon data.

Listing 5-4 Manipulating icon data in memory

```
PROCEDURE MyGetIconData (iconType: ResType; iconSuite: Handle;
                        VAR iconHandle: Handle);
VAR
    myErr: OSErr;
BEGIN
    {get the data for the icon with iconType from the suite}
    myErr := GetIconFromSuite(iconHandle, iconSuite, iconType);
    {do whatever with the data}
    myErr := MyManipulateIconData(iconHandle, iconType);
END;
```

The Icon Utilities also include routines that allow you to perform an action on one or more icons in an icon suite and to perform hit-testing on icons. For information about these routines, see “Performing Operations on Icons in an Icon Suite” and “Determining Whether a Point or Rectangle Is Within an Icon” beginning on page 5-38 and page 5-46, respectively.

Drawing Icons That Are Not Part of an Icon Family

To draw icons of resource type `'ICON'` or `'icicn'` in menus and dialog boxes, you can use the Menu Manager and Dialog Manager as described in *Inside Macintosh: Macintosh Toolbox Essentials*. You can also use Menu Manager routines to draw resources of type `'SICN'`.

Icon Utilities

To draw resources of resource type 'ICON', 'cicn', or 'SICN' in your application's windows, you can use these routines:

Resource type	Routines
'ICON'	PlotIconHandle PlotIcon
'cicn'	PlotCIconHandle PlotCIcon
'SICN'	PlotSICNHandle

The routines in this list that end in `Handle` allow you to specify alignment and transforms for the icons. You are responsible for disposing of the handle you pass to any of these routines.

Note

Unlike `PlotCIcon`, `PlotCIconHandle` doesn't honor the current foreground and background colors. [u](#)

The listings that follow provide examples of how to draw each of the three icon resource types that are not part of an icon family.

Listing 5-5 shows how to use `PlotIcon` to draw an icon of resource type 'ICON' without specifying alignment or transforms. The application-defined procedure `MyPlotAnICON` uses `GetIcon` to get a handle to the data for the desired icon and then passes the destination rectangle and the handle to `PlotIcon`.

Listing 5-5 Drawing an icon of resource type 'ICON'

```
PROCEDURE MyPlotAnICON (resID: Integer; destRect: Rect;
                       VAR myIcon: Handle);
BEGIN
    myIcon := GetIcon(resID);
    PlotIcon(destRect, myIcon);
END;
```

IMPORTANT

When you are finished using a handle obtained from `GetIcon`, use the `ReleaseResource` procedure to release the memory occupied by the icon resource data; for more information about `ReleaseResource`, see the chapter "Resource Manager" in this book. [s](#)

Icon Utilities

Listing 5-6 shows how to use `PlotIconHandle` to draw an icon of resource type 'ICON' with a specific alignment and transform. The application-defined procedure `MyPlotAnICONWithAlignAndTransform` uses `GetIcon` to get a handle to the data for the desired icon and then passes the destination rectangle, alignment, transform, and handle to `PlotIconHandle`.

Listing 5-6 Drawing an icon of resource type 'ICON' with a specific alignment and transform

```
PROCEDURE MyPlotAnICONWithAlignAndTransform
    (resID: Integer; destRect: Rect;
    align: IconAlignmentType;
    transform: IconTransformType; VAR myIcon: Handle);
VAR
    myErr: OSErr;
BEGIN
    myIcon := GetIcon(resID);
    myErr := PlotIconHandle(destRect, align, transform, myIcon);
END;
```

For the `PlotIconHandle` function in Listing 5-6 to draw the icon without stretching it, the destination rectangle passed in the `destRect` parameter of `MyPlotAnICONWithAlignAndTransform` must be exactly 32 by 32 pixels. If the destination rectangle is not 32 by 32 pixels, `PlotIconHandle` expands or shrinks the icon to fit.

Listing 5-7 shows how to use `PlotCIcon` to draw an icon of resource type 'cicn' without specifying alignment or transform. The `MyPlotAcicn` procedure uses `GetCIcon` to get a handle to the color icon record of the desired icon and then passes the destination rectangle and handle to `PlotCIcon`.

Listing 5-7 Drawing an icon of resource type 'cicn'

```
PROCEDURE MyPlotAcicn (resID: Integer; destRect: Rect;
    VAR myCicnIcon: CIconHandle);
BEGIN
    myCicnIcon := GetCIcon(resID);
    PlotCIcon(destRect, myCicnIcon);
END;
```

Icon Utilities

Listing 5-8 shows how to use `PlotCIconHandle` to draw an icon of resource type 'cicn' with a specific alignment and transform. Listing 5-8 uses `GetCIcon` to get a handle to the color icon record of the desired icon and then passes the destination rectangle, alignment, transform, and handle to `PlotCIconHandle`.

Listing 5-8 Drawing an icon of resource type 'cicn' with a specific alignment and transform

```
PROCEDURE MyPlotAcicnWithAlignAndTransform
    (resID: Integer; destRect: Rect;
    align: IconAlignmentType;
    transform: IconTransformType;
    VAR myCicnIcon: CIconHandle);
VAR
    myErr: OSErr;
BEGIN
    myCicnIcon := GetCIcon(resID);
    myErr := PlotCIconHandle(destRect, align, transform,
                            myCicnIcon);
END;
```

Listing 5-9 shows how to use `PlotSICNHandle` to draw an icon of resource type 'SICN' with a specific alignment and transform. The application-defined procedure `MyPlotAnSICNWithAlignAndTransform` uses `GetResource` to get a handle to the data for the desired icon and then passes the destination rectangle, alignment, transform, and handle to `PlotSICNHandle`.

Listing 5-9 Drawing an icon of resource type 'SICN' with a specific alignment and transform

```
PROCEDURE MyPlotAnSICNWithAlignAndTransform
    (resID: Integer; destRect: Rect;
    align: IconAlignmentType;
    transform: IconTransformType; VAR myIcon: Handle);
VAR
    myErr: OSErr;
BEGIN
    myIcon := GetResource('SICN', resID);
    myErr := PlotSICNHandle(destRect, align, transform, myIcon);
END;
```

For the `PlotSICNHandle` function in Listing 5-9 to draw the icon without stretching it, the destination rectangle passed in the `destRect` parameter of `MyPlotAnSICNWithAlignAndTransform` must be exactly 16 by 16 pixels. If the destination rectangle is not this size, `PlotSICNHandle` expands or shrinks the icon to fit.

Icon Utilities Reference

The sections that follow describe the data structure and routines provided by the Icon Utilities.

The first section, “Data Structure,” describes the color icon record. “Icon Utilities Routines” beginning on page 5-18 describes the routines for drawing and manipulating icons. “Application-Defined Routines” beginning on page 5-57 describes the syntax of the icon action and icon getter functions that your application can provide for use by Icon Utilities routines.

Data Structure

This section describes the color icon record. Note that you use the color icon record only for icons of resource type `'cicn'`; you do not need to use the color icon record for any of the color icons in an icon family.

The Color Icon Record

The `GetCIcon` function reads in a **color icon resource**—that is, an icon resource of type `'cicn'`—and returns a handle to a color icon record. A **color icon record** is defined by the `CIcon` data type.

```

TYPE
  CIcon =
  RECORD
    iconPMap:      PixMap;      {the icon's pixel map}
    iconMask:      BitMap;      {the icon's mask}
    iconBMap:      BitMap;      {the icon's bitmap}
    iconData:      Handle;      {handle to the icon's data}
    iconMaskData: ARRAY[0..0] OF Integer;
  END;
  CIconPtr      = ^CIcon;      {pointer to color icon record}
  CIconHandle = ^CIconPtr;    {handle to color icon record}

```

Icon Utilities

Field descriptions

<code>iconPMap</code>	The pixel map describing the icon. Note that this is a pixel map record, not a handle to a pixel map record.
<code>iconMask</code>	A bitmap of the icon's mask.
<code>iconBMap</code>	A bitmap of the icon.
<code>iconData</code>	A handle to the icon's pixel image.
<code>iconMaskData</code>	An array containing the icon's mask data followed by the icon's bitmap data. This is used only when the icon is stored as a resource.

Your application can load a color icon resource into memory using the `GetCIcon` function. All color icon resources should be marked purgeable. To draw a color icon, you can use the `PlotCIcon` or `PlotCIconHandle` function. When your application has finished using a color icon, it can dispose of the color icon record by calling the `DisposeCIcon` function.

You can use icons of resource type `'cicn'` in menus the same way that you use resources of type `'ICON'`. If a menu item specifies an icon number, the menu definition procedure first tries to load in a `'cicn'` resource with the specified resource ID. If it doesn't find one, the menu definition procedure tries to load in an `'ICON'` resource with the same ID. The Dialog Manager also uses a `'cicn'` resource instead of an `'ICON'` resource if it finds one with the same resource ID. For more information, see *Inside Macintosh: Macintosh Toolbox Essentials*.

For information about the format of a color icon resource, see *Inside Macintosh: Imaging With QuickDraw*.

Icon Utilities Routines

This section describes the Icon Utilities routines. You can use these routines to draw icons in windows and, if necessary, in menus and dialog boxes. You can also use Icon Utilities routines to perform operations on icons in an icon suite, get and set labels associated with specific icon suites, dispose of icon suites and color icon records, convert an icon mask to a region, perform hit-testing, and create and manipulate icon caches. Note that you can pass a handle to an icon cache to any of the Icon Utilities routines that accept a handle to an icon suite.

Most of the Icon Utilities routines are available only in System 7 and later. To determine whether they are available, call the `Gestalt` function with the `gestaltIconUtilitiesAttr` selector and check the value of the response parameter. If the bit indicated by the constant `gestaltIconUtilitiesPresent` is set, then the Icon Utilities are available. The `GetIcon`, `PlotIcon`, `GetCIcon`, `PlotCIcon`, and `DisposeCIcon` routines are available in both System 6 and System 7.

Note

The Icon Utilities routines do not place any restrictions on whether icon resources are purgeable or nonpurgeable; however, in general, you should specify your icon resources as purgeable. [u](#)

This section first describes the routines you can use to draw icons from an icon family and then describes the routines that work with icon suites and icon caches.

IMPORTANT

All of the Icon Utilities routines may move or purge memory blocks in the application heap or for some other reason should not be called from within an interrupt. Your application should not call Icon Utilities routines at interrupt time. [s](#)

Assembly-Language Note

You can invoke Icon Utilities routines by using the trap `_IconDispatch` with the appropriate routine selector. The routine selectors are listed in “Assembly-Language Summary” beginning on page 5-71. [u](#)

Drawing Icons From Resources

The routines described in this section allow you to plot an icon directly from a resource without first creating an icon suite.

To draw an icon from an icon family (that is, those resources of type 'ICN#', 'ics#', 'icl4', 'icl8', 'ics4', or 'ics8' that share the same resource ID), use the `PlotIconID` function. This function gets the icon's data from its resource and also allows you to specify transforms and alignment. The `PlotIconID` function also determines, from the destination rectangle in which the icon is to be drawn and the current bit depth of the display device, which resource type to get from the icon family.

To draw an icon obtained with the aid of an icon getter function, use the `PlotIconMethod` function. For information about icon getter functions, see “Icon Getter Functions” beginning on page 5-58.

To plot an icon of resource types 'ICON' and 'cicn' from an icon handle previously obtained from the `GetIcon` or `GetCIcon` function, use the `PlotIconHandle` and `PlotCIconHandle` functions, respectively. These functions allow you to specify transforms and alignment.

You can also plot an icon of resource types 'ICON' and 'cicn' using the `PlotIcon` and `PlotCIcon` procedures, respectively. However, neither of these procedures allow you to specify transforms and alignment.

To plot an icon of resource type 'SICN', use the `PlotSICNHandle` function. This function allows you to specify transforms and alignment.

PlotIconID

You can use the `PlotIconID` function to draw the icon described by an icon family. From the icon family, `PlotIconID` selects the most appropriate icon resource for the current bit depth of the display device and the rectangle in which the icon is to be drawn.

```
FUNCTION PlotIconID (theRect: Rect; align: IconAlignmentType;
                    transform: IconTransformType;
                    theResID: Integer): OSErr;
```

<code>theRect</code>	The rectangle, specified in local coordinates of the current graphics port, in which to draw the icon. The <code>PlotIconID</code> function determines, from the size of the specified destination rectangle and the current bit depth of the display device, which icon of a given size to draw from an icon family.
<code>align</code>	A value that specifies how <code>PlotIconID</code> should align the icon within the rectangle. For example, you can specify that <code>PlotIconID</code> center the icon within the rectangle or align it at one side or the other. See the description that follows for a list of constants you can use in this parameter.
<code>transform</code>	A value that specifies how <code>PlotIconID</code> should modify the appearance of the icon. See the description that follows for a list of constants you can use in this parameter.
<code>theResID</code>	The resource ID of the icon to draw. The icon resource must be of resource type <code>'ICN#'</code> , <code>'ics#'</code> , <code>'icl4'</code> , <code>'icl8'</code> , <code>'ics4'</code> , or <code>'ics8'</code> .

DESCRIPTION

The `PlotIconID` function plots a single icon from the icon family specified by `theResID`. You cannot determine which icon from the family it will draw; `PlotIconID` bases this decision on the size of the specified destination rectangle and the current bit depth of the display device. For example, if the destination rectangle has the coordinates (100,100,116,116) and the display device is set to 4-bit color, the `PlotIconID` function draws the icon of type `'ics4'` if that icon is available in the icon family.

If the width or height of a destination rectangle is greater than or equal to 32, `PlotIconID` uses the 32-by-32 pixel icon with the appropriate bit depth for the display device. If the destination rectangle is less than 32 by 32 pixels and greater than 16 pixels wide or 12 pixels high, `PlotIconID` uses the 16-by-16 pixel icon with the appropriate bit depth. If the destination rectangle's height is less than or equal to 12 pixels or its width is less than or equal to 16 pixels, `PlotIconID` uses the 12-by-16 pixel icon with the appropriate bit depth. (Typically only the Finder and Standard File Package use 12-by-16 pixel icons.)

Icon Utilities

You can use these constants in the `align` parameter to specify the alignment of the icon within the rectangle specified by the `theRect` parameter:

```
CONST
    atNone           = $0; {no special alignment}
    atVerticalCenter = $1; {centered vertically}
    atTop            = $2; {top aligned}
    atBottom        = $3; {bottom aligned}
    atHorizontalCenter = $4; {centered horizontally}
    atLeft          = $8; {left aligned}
    atRight         = $C; {right aligned}
    atAbsoluteCenter = (atVerticalCenter + atHorizontalCenter);
    atCenterTop     = (atTop + atHorizontalCenter);
    atCenterBottom  = (atBottom + atHorizontalCenter);
    atCenterLeft    = (atVerticalCenter + atLeft);
    atTopLeft       = (atTop + atLeft);
    atBottomLeft    = (atBottom + atLeft);
    atCenterRight   = (atVerticalCenter + atRight);
    atTopRight      = (atTop + atRight);
    atBottomRight   = (atBottom + atRight);
```

The destination rectangle passed in the `theRect` parameter of `PlotIconID` must be exactly 32 by 32 pixels, 16 by 16 pixels, or 12 by 16 pixels for `PlotIconID` to draw the icon without stretching it. If the destination rectangle is not one of these standard sizes, `PlotIconID` expands or shrinks the icon to fit. After stretching or shrinking the icon, the `PlotIconID` function aligns the icon according to the value specified in the `align` parameter, moving the icon so that the edges of its mask align with the specified side or direction.

You can pass constants in the `transform` parameter to specify how you want the icon modified, if at all, when plotted by `PlotIconID`. If you don't want to specify any transform constants, specify `ttNone` in the `transform` parameter.

```
CONST ttNone      = $0;
```

You can use these constants in the `transform` parameter to transform the icon in a manner analogous to certain Finder states for icons:

```
CONST
    ttDisabled      = $1;
    ttOffline       = $2;
    ttOpen          = $3;
    ttSelected      = $4000;
    ttSelectedDisabled = (ttSelected + ttDisabled);
    ttSelectedOffline = (ttSelected + ttOffline);
    ttSelectedOpen  = (ttSelected + ttOpen);
```

Icon Utilities

You can use another group of constants to color the icon using the Finder label colors. To determine the appropriate label for a file's icon, you can check bits 1–3 of the `fdFlags` field in the file's file information record. These bits contain a number from 0 to 7 indicating the label setting (0 indicates no label). Simply add the corresponding constant from this list to the `transform` parameter when you call `PlotIconID`:

```
CONST
    ttLabel1      = $0100;
    ttLabel2      = $0200;
    ttLabel3      = $0300;
    ttLabel4      = $0400;
    ttLabel5      = $0500;
    ttLabel6      = $0600;
    ttLabel7      = $0700;
```

RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>noMaskFoundErr</code>	-1000	No mask found

SEE ALSO

For an example of the use of the `PlotIconID` function, see Listing 5-1 on page 5-10.

To restrict the icons from an icon family that are available for use by the Icon Utilities, see “Drawing Specific Icons From an Icon Family” on page 5-12.

For information about the file information record, see the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.

PlotIconMethod

You can use the `PlotIconMethod` function to plot an icon obtained with the aid of an icon getter function for a specified destination rectangle and alignment.

```
FUNCTION PlotIconMethod (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theMethod: IconGetter;
                        yourDataPtr: UNIV Ptr): OSErr;
```

<code>theRect</code>	The rectangle in which to draw the icon, specified in local coordinates of the current graphics port.
<code>align</code>	A value that specifies how to align the icon within the rectangle specified by <code>theRect</code> . See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.

Icon Utilities

<code>transform</code>	A value that specifies how <code>PlotIconMethod</code> should modify the appearance of the icon. See the description of <code>PlotIconID</code> beginning on page 5-20 for a list of constants you can use in this parameter.
<code>theMethod</code>	A pointer to an icon getter function.
<code>yourDataPtr</code>	A pointer to data that is passed to your icon getter function.

DESCRIPTION

The `PlotIconMethod` function uses your icon getter function to obtain the icon to draw. Then `PlotIconMethod` draws this icon in the specified destination rectangle, with the specified transform and alignment.

`PlotIconMethod` passes to your icon getter function the type of the icon to draw and the value specified in the `yourDataPtr` parameter. The `PlotIconMethod` function examines the current bit depth of the display devices and calls your icon getter function once for each display device that intersects the rectangle specified in the parameter `theRect`. Your icon getter function should return a handle to the requested icon's data. Your icon getter function can get the icon data using whatever method is appropriate to your application. For example, your application might maintain its own cache of icons or use its icon getter function to get an icon from the desktop database.

RESULT CODES

<code>noErr</code>	0	No error
<code>noMaskFoundErr</code>	-1000	No mask found

SEE ALSO

For more information about icon getter functions, see page 5-58.

PlotIcon

You can use the `PlotIcon` procedure to plot an icon of resource type 'ICON'. You must have previously obtained a handle to the icon using `GetIcon` (or `GetResource` or other Resource Manager routines).

```
PROCEDURE PlotIcon (theRect: Rect; theIcon: Handle);
```

<code>theRect</code>	The rectangle in which to draw the icon, specified in local coordinates of the current graphics port.
<code>theIcon</code>	A handle to the icon to draw.

DESCRIPTION

The `PlotIcon` procedure draws the icon specified by the given handle. Unlike `PlotIconHandle`, `PlotIcon` does not allow you to specify any transforms or alignment. The `PlotIcon` procedure uses the `QuickDraw` procedure `CopyBits` with the `srcCopy` transfer mode.

If the destination rectangle is not 32 by 32 pixels, `PlotIcon` stretches or shrinks the icon to fit.

To plot an icon of resource type 'ICON' with a specified transform and alignment, use `PlotIconHandle` (described next).

SEE ALSO

For an example of the use of the `PlotIcon` procedure, see Listing 5-5 on page 5-14. For information on `GetIcon`, see page 5-28. For information on the `QuickDraw` procedure `CopyBits`, see *Inside Macintosh: Imaging With QuickDraw*.

PlotIconHandle

You can use the `PlotIconHandle` function to plot an icon of resource type 'ICON' or 'ICN#'. You must have previously obtained a handle to the icon using `GetIcon` (or `GetResource` or other Resource Manager routines).

```
FUNCTION PlotIconHandle (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theIcon: Handle): OSErr;
```

<code>theRect</code>	The rectangle in which to draw the icon, specified in local coordinates of the current graphics port.
<code>align</code>	A value that specifies how <code>PlotIconHandle</code> should align the icon within the rectangle. See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>transform</code>	A value that specifies how <code>PlotIconHandle</code> should modify the appearance of the icon. See the description of <code>PlotIconID</code> beginning on page 5-20 for a list of constants you can use in this parameter.
<code>theIcon</code>	A handle to the icon to draw.

DESCRIPTION

The `PlotIconHandle` function draws the icon specified by the `theIcon` parameter with the transform and alignment specified by the `transform` and `align` parameters.

IMPORTANT

To plot an icon from an icon suite, you should normally use `PlotIconSuite`. The `PlotIconHandle` function may not draw the icon correctly if you pass it the handle returned in the `theIconData` parameter of `GetIconFromSuite`.

RESULT CODES

<code>noErr</code>	0	No error
<code>noMaskFoundErr</code>	-1000	No mask found

SEE ALSO

For an example of the use of the `PlotIconHandle` function, see Listing 5-6 on page 5-15. For information on `GetIcon`, see page 5-28.

PlotCIcon

You can plot a color icon of resource type 'cicn' using the `PlotCIcon` procedure. You must have previously obtained a handle to the icon using `GetCIcon` (or `GetResource` or other Resource Manager routines).

```
PROCEDURE PlotCIcon (theRect: Rect; theIcon: CIconHandle);
```

`theRect` The rectangle in which to draw the icon, specified in local coordinates of the current graphics port.

`theIcon` A handle to the color icon record of the color icon to draw.

DESCRIPTION

The `PlotCIcon` procedure draws the color icon specified by the given handle. The `iconMask` field of the color icon record determines which pixels in the `iconPMap` field are drawn and which are not. Only pixels with 1s in corresponding positions in the `iconMask` field are drawn. If the screen depth is 1 or 2 bits per pixel, `PlotCIcon` uses the `iconBMap` field instead of the `iconPMap` field (unless the `rowBytes` field of `IconBMap` contains 0, indicating that there is no bitmap for the icon).

Icon Utilities

When `PlotCIcon` draws the icon, it uses the `bounds` field of `iconPMap` as the source rectangle of the image. If the destination rectangle is not the same size as the icon or its mask, `PlotCIcon` stretches or shrinks the icon to fit. The icon's pixels are remapped to the current depth and color table, if necessary. The `bounds` fields of `iconPMap`, `iconBMap`, and `iconMask` are expected to be equal in size.

Unlike `PlotIconHandle`, `PlotCIcon` does not allow you to specify any transforms or alignment. The `PlotCIcon` procedure uses the QuickDraw procedure `CopyMask` and doesn't send any of its drawing commands through QuickDraw bottleneck routines. Therefore, calls to `PlotCIcon` are not recorded as pictures.

RESULT CODE

`noErr` 0 No error

SEE ALSO

For a description of the color icon record, see "The Color Icon Record" on page 5-17. For information on `GetCIcon`, see page 5-29. For information on the QuickDraw procedure `CopyMask`, see *Inside Macintosh: Imaging With QuickDraw*.

For an example of the use of the `PlotCIcon` procedure, see Listing 5-7 on page 5-15.

PlotCIconHandle

You can use the `PlotCIconHandle` function to plot an icon of resource type `'icn'`. You must have previously obtained a handle to the icon using `GetCIcon` (or `GetResource` or other Resource Manager routines).

```
FUNCTION PlotCIconHandle (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theCIcon: CIconHandle): OSErr;
```

<code>theRect</code>	The rectangle in which to draw the icon, specified in local coordinates of the current graphics port.
<code>align</code>	A value that specifies how <code>PlotCIconHandle</code> should align the icon within the rectangle. See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>transform</code>	A value that specifies how <code>PlotCIconHandle</code> should modify the appearance of the icon. See the description of <code>PlotIconID</code> beginning on page 5-20 for a list of constants you can use in this parameter.
<code>theCIcon</code>	A handle to the color icon record of the icon to draw.

DESCRIPTION

The `PlotCIconHandle` function draws the specified color icon with the transform and alignment specified by the `transform` and `align` parameters. Unlike `PlotCIcon`, `PlotCIconHandle` doesn't honor the current foreground and background colors.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Error in parameter list

SEE ALSO

For an example of the use of `PlotCIconHandle`, see Listing 5-8 on page 5-16. For information on `GetCIcon`, see page 5-29. For a description of the color icon record, see page 5-17.

PlotSICNHandle

You can use the `PlotSICNHandle` function to plot a small icon of resource type 'SICN' with a specified transform and alignment. You must have previously obtained a handle to the icon using `GetResource` (or other Resource Manager routines).

```
FUNCTION PlotSICNHandle (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theSICN: Handle): OSErr;
```

<code>theRect</code>	The rectangle in which to draw the icon, specified in local coordinates of the current graphics port.
<code>align</code>	A value that specifies how <code>PlotSICNHandle</code> should align the icon within the rectangle. See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>transform</code>	A value that specifies how <code>PlotSICNHandle</code> should modify the appearance of the icon. See the description of <code>PlotIconID</code> beginning on page 5-20 for a list of constants you can use in this parameter.
<code>theSICN</code>	A handle to the icon to draw.

DESCRIPTION

The `PlotSICNHandle` function draws the specified small icon with the transform and alignment specified by the `transform` and `align` parameters. Only 'SICN' resources with a single member—or with two members, the second of which is a mask for the first—plot correctly.

RESULT CODES

noErr	0	No error
noMaskFoundErr	-1000	No mask found

SEE ALSO

For an example of the use of the `PlotSICNHandle` function, see Listing 5-9 on page 5-16.

Getting Icons From Resources That Don't Belong to an Icon Family

You can get a handle to an 'ICON' or 'icn' resource using the `GetIcon` and `GetCIcon` functions. You can then draw these icons using the routines `PlotIcon`, `PlotCIcon`, `PlotIconHandle`, or `PlotCIconHandle` (see “Drawing Icons From Resources” beginning on page 5-19).

To get a handle to an icon suite for a given icon family, use the routines described in “Creating an Icon Suite” beginning on page 5-30.

GetIcon

You can use the `GetIcon` function to get a handle to an icon resource of type 'ICON'.

```
FUNCTION GetIcon (iconID: Integer): Handle;
```

`iconID` The resource ID for an icon of resource type 'ICON'.

DESCRIPTION

The `GetIcon` function reads in the 'ICON' resource with the specified resource ID and returns a handle to it. The `GetIcon` function searches the current resource chain for the resource. If `GetIcon` finds the resource, it reads the resource and returns a handle to the icon as its function result. If `GetIcon` can't find the resource, it returns `NIL` as its function result.

To draw an icon obtained from `GetIcon` in a specified rectangle, you can use either `PlotIcon` or `PlotIconHandle`. Unlike `PlotIcon`, `PlotIconHandle` allows you to specify transforms and alignments.

When you are finished using a handle obtained from `GetIcon`, use the `ReleaseResource` procedure to release the memory occupied by the icon resource data.

RESULT CODES

noErr	0	No error
resNotFound	-192	Resource not found

SEE ALSO

For a description of the `PlotIcon` procedure and `PlotIconHandle` function, see page 5-23 and page 5-24, respectively. For information about `ReleaseResource`, see the chapter “Resource Manager” in this book.

GetCIcon

You can use `GetCIcon` to get a handle to a color icon of resource type 'cicn'.

```
FUNCTION GetCIcon (iconID: Integer): CIconHandle;
```

`iconID` The resource ID for an icon of resource type 'cicn'.

DESCRIPTION

The `GetCIcon` function reads in the 'cicn' resource with the specified resource ID and returns a handle to it. The `GetCIcon` function searches the current resource chain for the resource. If `GetCIcon` finds the resource, it reads the resource, creates a color icon record for the icon, and initializes the fields of the record according to the information contained in the 'cicn' resource. `GetCIcon` returns a handle to the color icon record as its function result. If `GetCIcon` can't find the resource, it returns `NIL` as its function result.

To draw an icon obtained from `GetCIcon` in a specified rectangle, you can use either the `PlotCIcon` or `PlotCIconHandle` routine. Unlike `PlotCIcon`, `PlotCIconHandle` allows you to specify transforms and alignments.

When you are finished with a handle obtained from `GetCIcon`, use the `DisposeCIcon` procedure to release the memory occupied by the color icon record.

RESULT CODES

noErr	0	No error
resNotFound	-192	Resource not found

SEE ALSO

For information about the color icon record, see “The Color Icon Record” on page 5-17. For information about the format of the 'cicn' resource, see *Inside Macintosh: Imaging With QuickDraw*.

For descriptions of the `PlotCIcon` procedure and `PlotCIconHandle` function, see page 5-25 and page 5-26, respectively. The `DisposeCIcon` procedure is described next.

Disposing of Icons

When you are finished with a handle obtained from `GetCIcon`, use the `DisposeCIcon` procedure to release the memory occupied by the color icon record.

When you are finished using a handle obtained from `GetIcon` or `GetResource`, use the `ReleaseResource` procedure to release the memory occupied by the icon resource data; for more information about `GetResource` and `ReleaseResource`, see the chapter “Resource Manager” in this book.

To dispose of icons in an icon suite, use the `DisposeIconSuite` function described on page 5-42.

DisposeCIcon

You can use the `DisposeCIcon` procedure to release the memory occupied by an icon color record obtained from the `GetCIcon` function. The `DisposeCIcon` procedure is also available as the `DisposCIcon` procedure.

```
PROCEDURE DisposeCIcon (theIcon: CIconHandle);
```

`theIcon` A handle to the color icon record to dispose of.

DESCRIPTION

The `DisposeCIcon` procedure disposes of any structure allocated by `GetCIcon`.

Creating an Icon Suite

You typically create an icon suite by reading all resources from a specific icon family into memory and storing handles to the icon resource data in a new icon suite. You can do this using the `GetIconSuite` function. Alternatively, you can create an empty icon suite using the `NewIconSuite` function and then add icons to it one at a time using the `AddIconToSuite` function.

Although you typically create an icon suite using `GetIconSuite` (which fills the suite with handles to icon resource data), you can also create an icon suite and then add handles to icon data. The handles that you add to the suite do not have to be associated with a resource fork. For example, your application might get icon data from the desktop database rather than reading it from a resource, or your application might read icon data from a resource and then detach it. In either case, you can provide a handle to the icon data and use `AddIconToSuite` to add the handle to the icon suite. You need to release the memory occupied by the icon suite when you’re finished using it. The `DisposeIconSuite` function releases this memory but does not release the memory of any resource handles. You can request `DisposeIconSuite` to release the memory of any other handles to icon data in the suite.

Note

When you create an icon suite from icon family resources, the associated resource file should remain open while you use Icon Utilities routines. u

GetIconSuite

You can use the `GetIconSuite` function to create an icon suite in memory that contains handles to a specified icon family's resources and to return a handle to the icon suite.

```
FUNCTION GetIconSuite (VAR theIconSuite: Handle;
                      theResID: Integer;
                      selector: IconSelectorValue): OSErr;
```

`theIconSuite`

`GetIconSuite` allocates the memory for and returns, in this parameter, a handle to an icon suite for the requested icon family. To release the memory occupied by an icon suite, you must use the `DisposeIconSuite` function.

`theResID`

The resource ID of the icons in the icon family to be read into memory.

`selector`

A value that indicates which icons from the icon family to include in the icon suite. See the description that follows for a list of constants you can use in this parameter.

DESCRIPTION

The `GetIconSuite` function returns a handle to a suite of icons for the icon family whose resource ID is specified in the `theResID` parameter. Use one or more of these constants in the `selector` parameter to specify which members of the family to include in the icon suite:

CONST

```
svLarge1Bit      = $00000001; {'ICN#' resource}
svLarge4Bit      = $00000002; {'icl4' resource}
svLarge8Bit      = $00000004; {'icl8' resource}
svSmall11Bit     = $00000100; {'ics#' resource}
svSmall4Bit      = $00000200; {'ics4' resource}
svSmall8Bit      = $00000400; {'ics8' resource}
svMini1Bit       = $00010000; {'icm#' resource}
svMini4Bit       = $00020000; {'icm4' resource}
svMini8Bit       = $00040000; {'icm8' resource}
svAllLargeData   = $000000FF; {'ICN#', 'icl4', and 'icl8' }
                  { resources}
svAllSmallData   = $0000FF00; {'ics#', 'ics4', and 'ics8' }
                  { resources}
```

Icon Utilities

```

svAllMiniData      = $00FF0000; {'icm#', 'icm4', and 'icm8' }
                    { resources}
svAll11BitData     = (svLarge1Bit + svSmall11Bit + svMini1Bit);
svAll4BitData      = (svLarge4Bit + svSmall4Bit + svMini4Bit);
svAll8BitData      = (svLarge8Bit + svSmall8Bit + svMini8Bit);
svAllAvailableData= $FFFFFFFF; {all resources of given ID}

```

These constants are additive; that is, you can add several constants to include the corresponding family members in the icon suite.

When you create an icon suite using `GetIconSuite`, it sets the default label for the suite to none. To set a new default label for an icon suite, use the `SetSuiteLabel` function.

If you call `SetResLoad` with the `load` parameter set to `FALSE` before you call `GetIconSuite`, the suite is filled with unloaded resource handles.

To perform operations on one or more icons in an icon suite, use the `ForEachIconDo` function.

To draw the icon described by the icon suite using the icon family member that is most suitable for the current bit depth of the display device, use the `PlotIconSuite` function.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory in heap zone

SEE ALSO

For examples of the use of the `GetIconSuite` function, see Listing 5-2 and Listing 5-3 on page 5-11 and page 5-12, respectively.

For a description of the `PlotIconSuite` and `ForEachIconDo` functions, see page 5-35 and page 5-38, respectively. For information on the `DisposeIconSuite` function, see page 5-42.

NewIconSuite

You can use the `NewIconSuite` function to get a handle to an empty icon suite. Then you can use `AddIconToSuite` to add handles to icon data.

```
FUNCTION NewIconSuite (VAR theIconSuite: Handle): OSErr;
```

`theIconSuite`

`NewIconSuite` allocates the memory for a new icon suite and returns, in this parameter, a handle to an empty icon suite.

DESCRIPTION

The `NewIconSuite` function returns a handle to an empty icon suite in the parameter `theIconSuite`. When you create an icon suite using `NewIconSuite`, it sets the default label for the suite to none. To set a new default label for an icon suite, use the `SetSuiteLabel` function. `NewIconSuite` allocates the memory for the icon suite handle. To release the memory occupied by an icon suite, you must use the `DisposeIconSuite` function.

RESULT CODES

<code>noErr</code>	0	No error
<code>memFullErr</code>	-108	Not enough memory in heap zone

SEE ALSO

For information on the `DisposeIconSuite` function, see page 5-42.

AddIconToSuite

You can use the `AddIconToSuite` function to add icons to an icon suite. This function is most often used to read icons into an empty icon suite created with `NewIconSuite`.

```
FUNCTION AddIconToSuite (theIconData: Handle; theSuite: Handle;
                        theType: ResType): OSErr;
```

`theIconData`

A handle to the data for the new icon to be added to the icon suite. You can obtain a handle to icon data using various routines, such as `GetIcon` or `GetResource`.

`theSuite`

A handle to the icon suite to which to add the icon.

`theType`

The resource type of the new icon. The resource type should be that of an icon family member.

DESCRIPTION

The `AddIconToSuite` function adds the handle to the icon data to the specified icon suite at the location reserved for icon data of type `theType`. If the icon suite already includes a handle to icon data for that type, `AddIconToSuite` replaces the handle to the old data without disposing of it. In this case you may want to call `GetIconFromSuite` (described next) first to obtain the old handle so that you can dispose of it.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	No such type in icon family

Getting Icons From an Icon Suite

The `GetIconFromSuite` function returns a handle to the specified icon in an icon suite.

GetIconFromSuite

You can use the `GetIconFromSuite` function to get an icon from an icon suite.

```
FUNCTION GetIconFromSuite (VAR theIconData: Handle;
                          theSuite: Handle;
                          theType: ResType): OSerr;
```

`theIconData`

`GetIconFromSuite` returns a handle to the data for the requested icon in this parameter. If an icon of the specified type does not exist in the given icon suite, `GetIconFromSuite` returns `NIL` in this parameter.

`theSuite`

A handle to the icon suite from which to get the icon.

`theType`

The resource type of the desired icon.

DESCRIPTION

The `GetIconFromSuite` function returns a handle to the data for the icon of type `theType` in the icon suite specified by `theSuite`. If you intend to dispose of the handle, pass a `NIL` handle to the `AddIconToSuite` function to delete the corresponding entry in the suite.

You can use the handle returned by `GetIconFromSuite` to manipulate the icon data, for example, to alter its color or add three-dimensional shading. However, you should not use the returned handle to draw the icon with other Icon Utilities routines.

IMPORTANT

To plot an icon from an icon suite, you should normally use `PlotIconSuite`. The `PlotIconHandle` function may not draw the icon correctly if you pass it the handle returned in the `theIconData` parameter of `GetIconFromSuite`. s

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Requested type not present in suite

SEE ALSO

For an example of the use of the `GetIconFromSuite` function, see Listing 5-4 on page 5-13.

For a description of the `AddIconToSuite` function, see page 5-33. The `PlotIconSuite` function is described next.

Drawing Icons From an Icon Suite

To draw an icon from an icon suite using the icon that is most appropriate for a specified rectangle and the current bit depth of the display device, use the `PlotIconSuite` function.

To draw an icon from a resource, use the routines described in “Drawing Icons From Resources” beginning on page 5-19. For example, to draw an icon from an icon family, use the `PlotIconID` function.

PlotIconSuite

You can use the `PlotIconSuite` function to draw the icon described by an icon suite using the most appropriate icon in the suite for the current bit depth of the display device and the rectangle in which the icon is to be drawn.

```
FUNCTION PlotIconSuite (theRect: Rect;
                       align: IconAlignmentType;
                       transform: IconTransformType;
                       theIconSuite: Handle): OSErr;
```

- `theRect` The rectangle in which to draw the icon. The `PlotIconSuite` function uses the size of the specified destination rectangle and the current bit depth of the display device to determine which icon from an icon suite to draw.
- `align` A value that specifies how `PlotIconSuite` should align the icon within the rectangle. For example, you can specify that `PlotIconSuite` center the icon within the rectangle or align it at one side or the other. See the description that follows for a list of constants you can use in this parameter.
- `transform` A value that specifies how `PlotIconSuite` should modify the appearance of the icon. See the description that follows for a list of constants you can use in this parameter.
- `theIconSuite` A handle to the icon suite from which `PlotIconSuite` gets the icon to draw. You can get a handle to an icon suite using the `GetIconSuite` or `NewIconSuite` function.

DESCRIPTION

The `PlotIconSuite` function plots a single icon from an icon suite in the current graphics port. You cannot determine which icon from a given suite it will draw; `PlotIconSuite` bases this decision on the size of the specified destination rectangle and the current bit depth of the display device. For example, if the destination rectangle has the coordinates (100,100,116,116) and the display device is set to 4-bit color, the `PlotIconSuite` function draws the icon of type 'ics4' if that icon is available in the icon suite.

If the width or height of a destination rectangle is greater than or equal to 32 pixels, `PlotIconSuite` uses the 32-by-32 pixel icon with the appropriate bit depth for the display device. If the destination rectangle is less than 32 by 32 pixels and greater than 16 pixels wide or 12 pixels high, `PlotIconSuite` uses the 16-by-16 pixel icon with the appropriate bit depth. If the destination rectangle's height is less than or equal to 12 pixels or its width is less than or equal to 16 pixels, `PlotIconSuite` uses the 12-by-16 pixel icon with the appropriate bit depth. (Typically, only the Finder and Standard File Package use 12-by-16 pixel icons.)

You can use these constants in the `align` parameter to specify the alignment of the icon within the rectangle specified by the parameter `theRect`:

CONST

```

atNone           = $0; {no special alignment}
atVerticalCenter = $1; {centered vertically}
atTop           = $2; {top aligned}
atBottom        = $3; {bottom aligned}
atHorizontalCenter = $4; {centered horizontally}
atLeft          = $8; {left aligned}
atRight         = $C; {right aligned}
atAbsoluteCenter = (atVerticalCenter + atHorizontalCenter);
atCenterTop     = (atTop + atHorizontalCenter);
atCenterBottom  = (atBottom + atHorizontalCenter);
atCenterLeft    = (atVerticalCenter + atLeft);
atTopLeft       = (atTop + atLeft);
atBottomLeft    = (atBottom + atLeft);
atCenterRight   = (atVerticalCenter + atRight);
atTopRight      = (atTop + atRight);
atBottomRight   = (atBottom + atRight);

```

The destination rectangle passed in the `theRect` parameter of `PlotIconSuite` must be exactly 32 by 32 pixels, 16 by 16 pixels, or 12 by 16 pixels for `PlotIconSuite` to draw the icon without stretching it. If the destination rectangle is not one of these standard sizes, `PlotIconSuite` expands or shrinks the icon to fit. After stretching or shrinking the icon, the `PlotIconSuite` function aligns the icon according to the value specified in the `align` parameter, moving the icon so that the edges of its mask align with the specified side or direction.

Icon Utilities

You can pass constants in the `transform` parameter to specify how you want the icon modified, if at all, when plotted by `PlotIconSuite`. If you don't want to specify any transform constants, specify `ttNone` in the `transform` parameter.

```
CONST ttNone          = $0;
```

You can use these constants in the `transform` parameter to transform the icon in a manner analogous to certain Finder states for icons:

```
CONST
    ttDisabled        = $1;
    ttOffline         = $2;
    ttOpen             = $3;
    ttSelected         = $4000;
    ttSelectedDisabled = (ttSelected + ttDisabled);
    ttSelectedOffline = (ttSelected + ttOffline);
    ttSelectedOpen    = (ttSelected + ttOpen);
```

You can use another group of constants to color the icons using the Finder label colors. To determine the appropriate label for a file's icon, you can check bits 1-3 of the `fdFlags` field in the file's file information record. These bits contain a number from 0 to 7 indicating the label setting (0 indicates no label). Simply add the corresponding constant from this list to the `transform` parameter when you call `PlotIconSuite`:

```
CONST
    ttLabel1          = $0100;
    ttLabel2          = $0200;
    ttLabel3          = $0300;
    ttLabel4          = $0400;
    ttLabel5          = $0500;
    ttLabel6          = $0600;
    ttLabel7          = $0700;
```

If you don't specify a label constant in the `transform` parameter, `PlotIconSuite` displays the icon using the default label for that icon suite. When you create an icon suite using `GetIconSuite` or `NewIconSuite`, these functions set the default label for the suite to none. To set a new default label for an icon suite, use the `SetSuiteLabel` function.

RESULT CODES

<code>noErr</code>	0	No error
<code>noMaskFoundErr</code>	-1000	No mask found

SEE ALSO

For examples of the use of the `PlotIconSuite` function, see Listing 5-2 and Listing 5-3, starting on page 5-11.

For information on the `SetSuiteLabel` function, see page 5-40. See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for more information about the file information record.

Performing Operations on Icons in an Icon Suite

You can perform an action on one or more icons in an icon suite using the `ForEachIconDo` function.

ForEachIconDo

You can use the `ForEachIconDo` function to perform an action on one or more icons in an icon suite.

```
FUNCTION ForEachIconDo (theSuite: Handle;
                       selector: IconSelectorValue;
                       action: IconAction;
                       yourDataPtr: Ptr): OSErr;
```

`theSuite` A handle to an icon suite.

`selector` A long integer whose bits determine which icons in the suite to perform the operation on. See the description that follows for a list of constants you can use in this parameter.

`action` A pointer to your icon action function.

`yourDataPtr` A pointer to data that is passed to your icon action function.

DESCRIPTION

The `ForEachIconDo` function uses the icon action function identified by the `action` parameter to perform an action on the specified icons in the icon suite. You can use these constants in the `selector` parameter to specify the icons on which to perform the action:

```
CONST
svLarge1Bit       = $00000001; {'ICN#' resource}
svLarge4Bit       = $00000002; {'icl4' resource}
svLarge8Bit       = $00000004; {'icl8' resource}
svSmall11Bit      = $00000100; {'ics#' resource}
svSmall4Bit       = $00000200; {'ics4' resource}
```

Icon Utilities

```

svSmall8Bit      = $00000400; {'ics8' resource}
svMini1Bit      = $00010000; {'icm#' resource}
svMini4Bit      = $00020000; {'icm4' resource}
svMini8Bit      = $00040000; {'icm8' resource}
svAllLargeData  = $000000FF; {'ICN#', 'icl4', and 'icl8' }
                  { resources}
svAllSmallData  = $0000FF00; {'ics#', 'ics4', and 'ics8' }
                  { resources}
svAllMiniData   = $00FF0000; {'icm#', 'icm4', and 'icm8' }
                  { resources}
svAll1BitData   = (svLarge1Bit + svSmall1Bit + svMini1Bit);
svAll4BitData   = (svLarge4Bit + svSmall4Bit + svMini4Bit);
svAll8BitData   = (svLarge8Bit + svSmall8Bit + svMini8Bit);
svAllAvailableData= $FFFFFFFF; {all resources of given ID}

```

These constants are additive; that is, you can add several constants to include the corresponding family members in the icon suite.

You can use the `yourDataPtr` parameter to pass a pointer to data or other information required by your icon action function. Typically, you use this parameter to specify which action your icon action function should perform.

`ForEachIconDo` calls your icon action function once for each type of icon specified in the `selector` parameter. `ForEachIconDo` passes to your icon action function a handle to the icon to perform the action on. Your icon action function should perform any action as indicated by the `yourDataPtr` parameter and return a result code. `ForEachIconDo` returns the result code returned by your icon action function. If your icon action function returns a nonzero function result, `ForEachIconDo` immediately returns to the application.

RESULT CODE

```
noErr    0    No error
```

SEE ALSO

See “Icon Action Functions” beginning on page 5-57 for more information about icon action functions.

Getting and Setting the Label for an Icon Suite

The `GetSuiteLabel` and `SetSuiteLabel` functions allow you to get and set the default label associated with an icon suite.

GetSuiteLabel

You can use the `GetSuiteLabel` function to get the default label setting associated with an icon suite.

```
FUNCTION GetSuiteLabel (theSuite: Handle): Integer;
```

`theSuite` A handle to an icon suite.

DESCRIPTION

The `GetSuiteLabel` function returns, as its function result, the default label setting associated with the specified icon suite. The default label setting is an integer from 1 to 7 that specifies which of the label colors shown in the Finder's Label menu is applied to icons of that suite when your application displays them. `GetSuiteLabel` returns 0 if the suite doesn't have a label.

You can override the default label setting for a suite by specifying a label in the `transform` parameter of the `PlotIconSuite` function.

SEE ALSO

To get information about the color and string for a specific label, you can use the `GetLabel` function, which is described on page 5-41.

SetSuiteLabel

You can use the `SetSuiteLabel` function to specify the default label associated with an icon suite.

```
FUNCTION SetSuiteLabel (theSuite: Handle;
                       theLabel: Integer): OSErr;
```

`theSuite` A handle to an icon suite.

`theLabel` An integer from 1 to 7 that specifies a label for the icon suite, or 0 to set the icon suite's label to none.

DESCRIPTION

The `SetSuiteLabel` function sets the label associated with the specified icon suite. The default label setting helps to determine which of the label colors shown in the Finder's Label menu is applied to icons of that suite when your application displays them.

You can override the default label setting for a suite by specifying a label in the `transform` parameter of the `PlotIconSuite` function. For example, suppose the color currently set for the third label displayed in the Finder's Label menu is red, and the color for the fourth label is green. If you set the default label for a suite using `SetSuiteLabel(theSuite, 3)`, then draw an icon from the same suite using `PlotIconSuite` and specifying `ttNone` in the `transform` parameter, the label color red is applied to the icon. However, if you specify `ttLabel4` in the `transform` parameter of the `PlotIconSuite` function, the label color green is applied to the icon.

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>paramErr</code>	<code>-50</code>	The <code>theLabel</code> parameter is greater than 7

SEE ALSO

For a description of the `PlotIconSuite` function, see page 5-35.

Getting Label Information

If you wish to display an icon in your application with the label color and label string associated with a specific label in the Finder, you can use the `GetLabel` function to get the current label information for that label.

GetLabel

You can use the `GetLabel` function to get the color and string used for a given label in the Label menu of the Finder and in the Labels control panel.

```
FUNCTION GetLabel (labelNumber: Integer; VAR labelColor: RGBColor;
                  VAR labelString: Str255): OSErr;
```

`labelNumber`

An integer from 1 to 7 indicating which label's information is requested.

`labelColor`

`GetLabel` returns, in this parameter, the color of the specified label.

`labelString`

`GetLabel` returns, in this parameter, the string associated with the specified label.

DESCRIPTION

The `GetLabel` function returns the color and string used for a specified label in the Label menu of the Finder and in the Labels control panel.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	The <code>labelNumber</code> parameter is greater than 7

SEE ALSO

For information on the `RGBColor` record, see *Inside Macintosh: Imaging With QuickDraw*.

Disposing of Icon Suites

When you are finished with an icon suite, you can release the memory it occupies by calling the `DisposeIconSuite` function.

DisposeIconSuite

You can use the `DisposeIconSuite` function to release the memory occupied by an icon suite.

```
FUNCTION DisposeIconSuite (theIconSuite: Handle;
                          disposeData: Boolean): OSErr;
```

`theIconSuite`

A handle to the icon suite to be disposed of.

`disposeData`

A Boolean value indicating whether or not to dispose of handles in the icon suite that are not associated with a resource fork.

DESCRIPTION

The `DisposeIconSuite` function releases the memory occupied by the specified icon suite. However, `DisposeIconSuite` does not release the memory of any icons explicitly associated with an open resource fork, that is, any handles to icon resource data that your application added to the suite using `GetIconSuite` or `AddIconToSuite`. For handles to icon data that your application added to the icon suite using `AddIconToSuite` (for example, if your application read in an icon resource, detached it, then added the handle to the suite), you can request that `AddIconToSuite` release the memory associated with the handles.

Set `disposeData` to `TRUE` to automatically release icon data that is associated with the specified icon suite but not explicitly associated with a resource fork. If you set `disposeData` to `FALSE`, `DisposeIconSuite` does not dispose of any icon data that is associated with the specified icon suite.

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>memWZErr</code>	<code>-111</code>	Attempt to operate on a free block

SEE ALSO

For more information on icon suites, see “Creating an Icon Suite” beginning on page 5-30.

Converting an Icon Mask to a Region

The `IconSuiteToRgn`, `IconIDToRgn`, and `IconMethodToRgn` functions create a region from an icon’s mask. `IconSuiteToRgn` and `IconIDToRgn` operate on an icon identified by a handle to a suite and an icon ID, respectively. The `IconMethodToRgn` function performs this operation on the icon mask that it obtains with the aid of your icon getter function. Once you have a region that describes the icon mask for a given icon, you can use it to perform accurate hit-testing and outline dragging of the icon in your application.

IconSuiteToRgn

You can use the `IconSuiteToRgn` function to convert, to a region, the icon mask in an icon suite. You specify a rectangle as one of the parameters to this function. `IconSuiteToRgn` determines, from the size of the specified rectangle, which mask from the icon suite to convert. Once it has determined which icon mask to convert, `IconSuiteToRgn` uses the specified rectangle as the bounding box of the region.

```
FUNCTION IconSuiteToRgn (theRgn: RgnHandle; iconRect: Rect;
                        align: IconAlignmentType;
                        theIconSuite: Handle): OSErr;
```

`theRgn` `IconSuiteToRgn` returns a handle to the requested region in this parameter. You must allocate memory for the region handle before calling `IconSuiteToRgn`.

`iconRect` The rectangle in which the icon is to be drawn, specified in local coordinates of the current graphics port. `IconSuiteToRgn` uses this rectangle as the bounding box of the region. `IconSuiteToRgn` determines, from the size of the rectangle specified in this parameter, which icon mask to use from the icon suite.

Icon Utilities

align A value that specifies how `IconSuiteToRgn` should align the region within the rectangle. See the description of `PlotIconSuite` on page 5-35 for a list of constants you can use in this parameter.

theIconSuite A handle to an icon suite.

DESCRIPTION

The `IconSuiteToRgn` function modifies the region referred to by the handle in the `theRgn` parameter. The returned region corresponds to the icon's mask (the mask defined by either an 'ICN#' or 'ics#' entry in an icon suite, according to the rectangle and alignment specified in the `iconRect` and `align` parameters).

RESULT CODES

<code>noErr</code>	0	No error
<code>noMaskFoundErr</code>	-1000	No mask found

IconIDToRgn

You can use the `IconIDToRgn` function to convert, to a region, the icon mask in an icon family. You specify a rectangle as one of the parameters to this function. `IconIDToRgn` determines, from the size of the specified rectangle, which mask from the icon family to convert. Once it has determined which icon mask to convert, `IconIDToRgn` uses the specified rectangle as the bounding box of the region.

```
FUNCTION IconIDToRgn (theRgn: RgnHandle; iconRect: Rect;
                    align: IconAlignmentType;
                    iconID: Integer): OSErr;
```

theRgn `IconIDToRgn` returns a handle to the requested region in this parameter. You must allocate memory for the region handle before calling `IconIDToRgn`.

iconRect The rectangle in which to draw the icon, specified in local coordinates of the current graphics port. `IconIDToRgn` uses this rectangle as the bounding box of the region. `IconIDToRgn` determines, from the size of the rectangle specified in this parameter, which icon mask to use from the icon family specified by `iconID`.

align A value that specifies how `IconIDToRgn` should align the mask within the rectangle. See the description of `PlotIconID` on page 5-20 for a list of constants you can use in this parameter.

iconID The resource ID of the icon for which to create a region.

DESCRIPTION

The `IconIDToRgn` function modifies the region referred to by the handle in the `theRgn` parameter. The returned region corresponds to the icon's mask (the mask defined by either an `'ICN#'` or `'ics#'` resource in an icon family, according to the rectangle and alignment specified in the `iconRect` and `align` parameters).

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>noMaskFoundErr</code>	<code>-1000</code>	No mask found

IconMethodToRgn

You can use the `IconMethodToRgn` function to convert, to a region, the mask for an icon that `IconMethodToRgn` obtains with the aid of your icon getter function.

```
FUNCTION IconMethodToRgn (theRgn: RgnHandle; iconRect: Rect;
                        align: IconAlignmentType;
                        theMethod: IconGetter;
                        yourDataPtr: Ptr): OSErr;
```

<code>theRgn</code>	<code>IconMethodToRgn</code> returns a handle to the requested region in this parameter. You must allocate memory for the region handle before calling <code>IconMethodToRgn</code> .
<code>iconRect</code>	The rectangle in which to draw the icon, specified in local coordinates of the current graphics port. The <code>IconMethodToRgn</code> function obtains the data for the icon mask from your icon getter function and then converts the icon mask to a region. <code>IconMethodToRgn</code> uses the rectangle specified in this parameter as the bounding box of the region.
<code>align</code>	A value that specifies how <code>IconMethodToRgn</code> should align the region within the rectangle. See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>theMethod</code>	A pointer to an icon getter function.
<code>yourDataPtr</code>	A pointer to data that is passed to your icon getter function.

DESCRIPTION

The `IconMethodToRgn` function modifies the region referred to by the handle in the `theRgn` parameter. The region corresponds to the icon's mask (as returned by your icon getter function, and according to the rectangle and alignment specified in the `iconRect` and `align` parameters).

Icon Utilities

`IconMethodToRgn` passes to your icon getter function the type of the icon to get and the value specified in the `yourDataPtr` parameter. The `IconMethodToRgn` function examines the size of the rectangle and requests the appropriate icon from your icon getter function—an icon of icon type `'ICN#'` or `'ics#'`. Your icon getter function should return a handle to the data of the requested icon type. The `IconMethodToRgn` function extracts the mask from the icon data that your icon getter function returns. If your icon getter function returns data that does not correspond to an icon of type `'ICN#'` or type `'ics#'`, `IconMethodToRgn` attempts to generate a mask from the returned data.

Your icon getter function can get the data for the icon and its mask using whatever method is appropriate to your application. For example, your application might maintain its own cache of icons (and pass a pointer to it in the `yourDataPtr` parameter) or use its icon getter function to get an icon from the desktop database.

RESULT CODES

<code>noErr</code>	0	No error
<code>noMaskFoundErr</code>	-1000	No mask found

Determining Whether a Point or Rectangle Is Within an Icon

You can use several Icon Utilities routines to perform hit-testing for points or rectangles against a specified icon. You specify a destination rectangle and alignment of the icon within the rectangle as parameters to these functions. The functions use this information to determine whether a specified point or rectangle is within the icon as it appears in the destination rectangle.

The `PtInIconSuite` and `PtInIconID` functions hit-test a specified point against the appropriate icon mask from an icon suite or icon family. The `PtInIconMethod` function hit-tests a specified point against an icon mask obtained with the aid of your icon getter function.

The `RectInIconSuite` and `RectInIconID` functions hit-test a specified rectangle against the appropriate icon mask from an icon suite or icon family. The `RectInIconMethod` function hit-tests a specified rectangle against an icon mask obtained with the aid of your icon getter function.

PtInIconSuite

You can use the `PtInIconSuite` function to determine whether a specified point is within an icon. (A point is considered to be within an icon if the point is within the icon's mask.) For example, you might use this function to determine whether a user clicked an icon in a window of your application. You specify as parameters to `PtInIconSuite` the same rectangle and alignment that you last used to draw the icon. `PtInIconSuite` uses the size of this rectangle to determine which icon mask from the icon suite to use for the operation. The `PtInIconSuite` function uses the location of this rectangle (along with the alignment) to determine whether a specified point is within the icon.

```
FUNCTION PtInIconSuite (testPt: Point; iconRect: Rect;
                       align: IconAlignmentType;
                       theIconSuite: Handle): Boolean;
```

<code>testPt</code>	The point to be tested, specified in local coordinates of the current graphics port.
<code>iconRect</code>	The rectangle in which the icon appears, specified in local coordinates of the current graphics port. <code>PtInIconSuite</code> determines, from the size of the rectangle specified in this parameter, which icon mask from the icon suite specified by <code>theIconSuite</code> to test the point against. <code>PtInIconSuite</code> then uses the location of this rectangle (and the location of the icon in the rectangle) to determine whether the specified point is within the icon.
<code>align</code>	A value that specifies how the icon against which to hit-test is aligned within the rectangle specified by <code>iconRect</code> . See the description of <code>PlotIconSuite</code> on page 5-35 for a list of constants you can use in this parameter.
<code>theIconSuite</code>	A handle to an icon suite.

DESCRIPTION

The `PtInIconSuite` function hit-tests the point specified by `testPt` against the appropriate icon mask from the specified icon suite. `PtInIconSuite` determines which icon mask to use ('`ICN#`' or '`ics#`') according to the rectangle specified in `iconRect`. The parameters `iconRect` and `align` should be the same as when the icon was last drawn. The `PtInIconSuite` function returns `TRUE` if the point is in the icon mask and `FALSE` if it is not.

PtInIconID

You can use the `PtInIconID` function to determine whether a specified point is within an icon. (A point is considered to be within an icon if the point is within the icon's mask.) For example, you might use this function to determine whether a user clicked an icon in a window of your application. You specify as parameters to `PtInIconID` the same rectangle and alignment that you last used to draw the icon. `PtInIconID` uses the size of this rectangle to determine which icon mask from the icon family to use for the operation. The `PtInIconID` function uses the location of this rectangle (along with the alignment) to determine whether a specified point is within the icon.

```
FUNCTION PtInIconID (testPt: Point; iconRect: Rect;
                    align: IconAlignmentType;
                    iconID: Integer): Boolean;
```

<code>testPt</code>	The point to be tested, specified in local coordinates of the current graphics port.
<code>iconRect</code>	The rectangle in which the icon appears, specified in local coordinates of the current graphics port. <code>PtInIconID</code> determines, from the size of the rectangle specified in this parameter, which icon mask from the icon family specified by <code>iconID</code> to test the point against. <code>PtInIconID</code> then uses the location of this rectangle (and the alignment of the icon in the rectangle) to determine whether the specified point is within the icon.
<code>align</code>	A value that specifies how the icon against which to hit-test is aligned within the rectangle specified by <code>iconRect</code> . See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>iconID</code>	A resource ID for an icon family.

DESCRIPTION

The `PtInIconID` function hit-tests the point specified by `testPt` against the appropriate icon mask from the icon family identified by `iconID`, using the destination rectangle and alignment specified by `iconRect` and `align`. The parameters `iconRect` and `align` should be the same as when the icon was last drawn. The `PtInIconID` function returns `TRUE` if the point is in the icon mask and `FALSE` if it is not.

PtInIconMethod

You can use the `PtInIconMethod` function to determine whether a specified point is within an icon. (A point is considered to be within an icon if the point is within the icon's mask.) The `PtInIconMethod` function obtains the icon to test against with the aid of your icon getter function.

```
FUNCTION PtInIconMethod (testPt: Point; iconRect: Rect;
                        align: IconAlignmentType;
                        theMethod: IconGetter;
                        yourDataPtr: Ptr): Boolean;
```

<code>testPt</code>	The point to be tested, specified in local coordinates of the current graphics port.
<code>iconRect</code>	The rectangle in which the icon appears, specified in local coordinates of the current graphics port.
<code>align</code>	A value that specifies how the icon against which to hit-test is aligned within the rectangle specified by <code>iconRect</code> . See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>theMethod</code>	A pointer to an icon getter function.
<code>yourDataPtr</code>	A pointer to data that is passed to your icon getter function.

DESCRIPTION

The `PtInIconMethod` function hit-tests the point specified by `testPt` against an icon obtained with the aid of an icon getter function, using the destination rectangle and alignment specified by `iconRect` and `align`. The parameters `iconRect` and `align` should be the same as when the icon was last drawn. The `PtInIconMethod` function returns `TRUE` if the point is in the icon mask and `FALSE` if it is not.

`PtInIconMethod` passes to your icon getter function the type of icon your function should retrieve (either `'ICN#'` or `'ics#'`) and also passes the value specified in the `yourDataPtr` parameter. The `PtInIconMethod` function examines the size of the specified rectangle and requests the appropriate icon from your icon getter function. Your icon getter function should return a handle to the requested icon's data. The `PtInIconMethod` function extracts the mask from the icon data that your icon getter function returns. If your icon getter function returns data that does not correspond to an icon of type `'ICN#'` or type `'ics#'`, `PtInIconMethod` attempts to generate a mask from the returned data.

Your icon getter function can get the icon's data using whatever method is appropriate to your application. For example, your application might maintain its own cache of icons (and pass a pointer to it in the `yourDataPtr` parameter) or use its icon getter function to get an icon from the desktop database.

SEE ALSO

For more information about icon getter functions, see page 5-58.

RectInIconSuite

You can use the `RectInIconSuite` function to hit-test a rectangle against the appropriate icon mask from an icon suite for a specified destination rectangle and alignment.

```
FUNCTION RectInIconSuite (testRect: Rect; iconRect: Rect;
                        align: IconAlignmentType;
                        theIconSuite: Handle): Boolean;
```

<code>testRect</code>	The rectangle to be tested, specified in local coordinates of the current graphics port.
<code>iconRect</code>	The rectangle in which the icon appears, specified in local coordinates of the current graphics port. Like <code>PtInIconSuite</code> , <code>RectInIconSuite</code> determines, from the size of the rectangle specified in this parameter, which icon mask from the icon suite specified by <code>theIconSuite</code> to test the <code>testRect</code> parameter against.
<code>align</code>	A value that specifies how the icon against which to hit-test is aligned within the rectangle specified by <code>iconRect</code> . See the description of <code>PlotIconSuite</code> on page 5-35 for a list of constants you can use in this parameter.
<code>theIconSuite</code>	A handle to an icon suite.

DESCRIPTION

The `RectInIconSuite` function hit-tests the rectangle specified by `testRect` against the appropriate icon mask from the icon suite as it appears in the `iconRect` rectangle. The parameters `iconRect` and `align` should be the same as when the icon was last drawn. The `RectInIconSuite` function returns `TRUE` if the rectangle intersects the icon mask and `FALSE` if it doesn't.

For example, if the coordinates of the `iconRect` parameter are (100,100,116,116) and the icon cache contains entries for each icon family member, `RectInIconSuite` uses the icon mask defined by the 'ics#' entry. The function aligns this mask (according to the `align` parameter) within the `iconRect` rectangle. The function then intersects the rectangle specified by `testRect` with the icon mask in the `iconRect` rectangle. Continuing with this example, if the icon mask is left-aligned so that its rightmost pixel appears at coordinates (112,112) and the coordinates of `testRect` are (114,114,130,130), then `RectInIconSuite` returns `FALSE`.

RectInIconID

You can use the `RectInIconID` function to hit-test a rectangle against the appropriate icon mask from an icon family for a specified destination rectangle and alignment.

```
FUNCTION RectInIconID (testRect: Rect; iconRect: Rect;
                    align: IconAlignmentType;
                    iconID: Integer): Boolean;
```

<code>testRect</code>	The rectangle to be tested, specified in local coordinates of the current graphics port.
<code>iconRect</code>	The rectangle in which the icon appears, specified in local coordinates of the current graphics port. Like <code>PtInIconID</code> , <code>RectInIconID</code> determines, from the size of the rectangle specified in this parameter, which icon mask from the icon family to test the <code>testRect</code> parameter against.
<code>align</code>	A value that specifies how the icon against which to hit-test is aligned within the rectangle specified by <code>iconRect</code> . See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>iconID</code>	A resource ID for an icon family.

DESCRIPTION

The `RectInIconID` function hit-tests the rectangle specified by `testRect` against the appropriate icon mask from the icon family as it appears in the `iconRect` rectangle. The parameters `iconRect` and `align` should be the same as when the icon was last drawn. The `RectInIconID` function returns `TRUE` if the rectangle intersects the icon mask and `FALSE` if it doesn't.

RectInIconMethod

You can use the `RectInIconMethod` function to hit-test a rectangle against an icon obtained by your icon getter function for a specified destination rectangle and alignment.

```
FUNCTION RectInIconMethod (testRect: Rect; iconRect: Rect;
                          align: IconAlignmentType;
                          theMethod: IconGetter;
                          yourDataPtr: Ptr): Boolean;
```

<code>testRect</code>	The rectangle to be tested, specified in local coordinates of the current graphics port.
<code>iconRect</code>	The rectangle in which the icon appears, specified in local coordinates of the current graphics port.
<code>align</code>	A value that specifies how the icon against which to hit-test is aligned within the rectangle specified by <code>iconRect</code> . See the description of <code>PlotIconID</code> on page 5-20 for a list of constants you can use in this parameter.
<code>theMethod</code>	A pointer to an icon getter function.
<code>yourDataPtr</code>	A pointer to data that is passed to your icon getter function.

DESCRIPTION

The `RectInIconMethod` function hit-tests the rectangle specified by `testRect` against an icon mask obtained with the aid of an icon getter function and as the icon appears in the destination rectangle. The parameters `iconRect` and `align` should be the same as when the icon was last drawn. The function returns `TRUE` if the rectangle intersects the icon mask and `FALSE` if it doesn't.

`RectInIconMethod` passes to your icon getter function the type of the icon your function should retrieve and the value specified in the `yourDataPtr` parameter. The `RectInIconMethod` function examines the size of the rectangle and requests the appropriate icon from your icon getter function—an icon of icon type `'ICN#'` or `'ics#'`. Your icon getter function should return a handle to the data of the requested icon type. The `RectInIconMethod` function extracts the mask from the icon data that your icon getter function returns. If your icon getter function returns data that does not correspond to an icon of type `'ICN#'` or type `'ics#'`, `RectInIconMethod` attempts to generate a mask from the returned data.

Your icon getter function can get the data for the icon and its mask using whatever method is appropriate to your application. For example, your application might maintain its own cache of icons (and pass a pointer to it in the `yourDataPtr` parameter) or use its icon getter function to get an icon from the desktop database.

SEE ALSO

For more information about icon getter functions, see page 5-58.

Working With Icon Caches

All the Icon Utilities routines that accept a handle to an icon suite also accept a handle to an icon cache. An icon cache is like an icon suite except that it also contains a pointer to an application-defined icon getter function and a pointer to data that can be used as a reference constant. An icon cache typically does not contain handles to the icon resources for all icon family members. Instead, if the icon cache does not contain an entry for a specific type of icon in an icon family, the Icon Utilities routines call your application's icon getter function to retrieve the data for that icon type.

You can use the routines described in this section to create and manipulate icon caches. To create an empty icon cache, you can use the `MakeIconCache` function, much as you use the `NewIconSuite` function to create an empty icon suite. Before drawing an icon in an icon cache, you can use the `LoadIconCache` function to load icon data for a specified destination rectangle, bit depth of the display device, and alignment.

To get and set the data associated with an icon cache or the icon getter function used with an icon cache, you can use the `GetIconCacheData`, `SetIconCacheData`, `GetIconCacheProc`, and `SetIconCacheProc` functions.

MakeIconCache

You can use the `MakeIconCache` function to get a handle to an empty icon cache, to which you can add icon data using the `LoadIconCache` function.

```
FUNCTION MakeIconCache (VAR theHandle: Handle;
                       makeIcon: IconGetter;
                       yourDataPtr: UNIV Ptr): OSerr;
```

`theHandle` `MakeIconCache` allocates memory for a new icon cache and returns a handle to the new icon cache in this parameter.

`makeIcon` A pointer to an icon getter function to associate with the icon cache.

`yourDataPtr` A pointer to the data to associate with the icon cache.

DESCRIPTION

`MakeIconCache` returns a handle to an empty icon cache in the parameter `theHandle`. The `MakeIconCache` function associates the icon getter function and the value specified in the parameters `makeIcon` and `yourDataPtr` with the new icon cache.

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory in heap zone

LoadIconCache

You can use the `LoadIconCache` function to load into an icon cache a handle to the appropriate icon data for a specified destination rectangle and the current bit depth, for drawing later with a specified alignment and transform.

```
FUNCTION LoadIconCache (theRect: Rect; align: IconAlignmentType;
                       transform: IconTransformType;
                       theIconCache: Handle): OSErr;
```

theRect The rectangle in which to draw the icon, specified in local coordinates of the current graphics port. `LoadIconCache` uses the rectangle specified in this parameter and the bit depth of the display device to determine which icon type to load into the cache.

align A value that specifies how to align the icon within the rectangle. See the description of `PlotIconSuite` on page 5-35 for a list of constants you can use in this parameter.

transform A value that specifies how to modify the appearance of the icon. See the description of `PlotIconSuite` beginning on page 5-35 for a list of constants you can use in this parameter.

theIconCache A handle to the icon cache into which to load the icon data.

DESCRIPTION

You can load icon data into an icon cache with the `LoadIconCache` function for drawing at a later time. For example, this can be useful if you suspect that the icon may be drawn at a time not convenient for loading resource data (for instance, when the resource fork isn't in the current resource chain). The `LoadIconCache` function uses the same criteria as `PlotIconSuite` to select the icon to load.

`LoadIconCache` uses the icon getter function associated with the icon cache to get the appropriate icon. The icon getter function returns a handle to the requested icon data, and `LoadIconCache` adds the returned handle to the entry for that icon in the icon cache.

Icon Utilities

After calling `LoadIconCache`, you can pass the same parameters to `PlotIconSuite` to plot the icon data. Note that if you specify an alignment when you call `LoadIconCache`, then call `PlotIconSuite` and specify no alignment, `PlotIconSuite` draws the icon using the alignment that you originally specified to `LoadIconCache`.

RESULT CODES

<code>noErr</code>	0	No error
<code>noMaskFoundErr</code>	-1000	No mask found

SEE ALSO

For a description of the `PlotIconSuite` function, see page 5-35.

GetIconCacheData

You can use the `GetIconCacheData` function to get the data associated with an icon cache.

```
FUNCTION GetIconCacheData (theCache: Handle;
                          VAR theData: Ptr): OSErr;
```

`theCache` A handle to the icon cache whose data is desired.

`theData` `GetIconCacheData` returns, in this parameter, a pointer to the data associated with the icon cache.

DESCRIPTION

The `GetIconCacheData` function returns, in the parameter `theData`, a pointer to the data associated with the specified icon cache. You associate data with an icon cache when you first create the cache using `MakeIconCache`. You can also set this data using `SetIconCacheData`.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	The parameter <code>theCache</code> must be a handle to an icon cache

SetIconCacheData

You can use the `SetIconCacheData` function to set the data associated with an icon cache.

```
FUNCTION SetIconCacheData (theCache: Handle; theData: Ptr): OSErr;
```

`theCache` **A handle to the icon cache whose data is to be set.**

`theData` **A pointer to the data to set.**

DESCRIPTION

The `SetIconCacheData` function sets the data associated with the specified icon cache to the data identified by `theData` parameter.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	The parameter <code>theCache</code> must be a handle to an icon cache

GetIconCacheProc

You can use the `GetIconCacheProc` function to get the icon getter function associated with an icon cache.

```
FUNCTION GetIconCacheProc (theCache: Handle;
                          VAR theProc: IconGetter): OSErr;
```

`theCache` **A handle to the icon cache whose icon getter function is desired.**

`theProc` **`GetIconCacheProc` returns a pointer to the requested icon getter function in this parameter.**

DESCRIPTION

The `GetIconCacheProc` function returns, in the parameter `theProc`, a pointer to the icon getter function currently associated with the specified icon cache.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	The parameter <code>theCache</code> must be a handle to an icon cache

SetIconCacheProc

You can use the `SetIconCacheProc` function to set the icon getter function associated with an icon cache.

```
FUNCTION SetIconCacheProc (theCache: Handle;
                          theProc: IconGetter): OSErr;
```

`theCache` A handle to the icon cache whose icon getter function is to be set.

`theProc` A pointer to the icon getter function to set.

DESCRIPTION

The `SetIconCacheProc` function sets the icon getter function for the specified icon cache to the icon getter function specified by the parameter `theProc`.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	The parameter <code>theCache</code> must be a handle to an icon cache

Application-Defined Routines

Your application can provide two functions for use by Icon Utilities routines. If you want to use the `ForEachIconDo` function to perform operations on icons, you must provide an icon action function. If you use icon caches or use any of the routines that end in `Method`, you must provide at least one icon getter function.

Icon Action Functions

You can perform operations on every icon in an icon suite by providing a pointer to an icon action function as a parameter to the `ForEachIconDo` function. The `ForEachIconDo` function calls your icon action function for specified icon resource types.

MyIconAction

The `action` parameter of `ForEachIconDo` must point to a function that uses this syntax:

```
FUNCTION MyIconAction (theType: ResType; VAR theIcon: Handle;
                      yourDataPtr: Ptr): OSErr;
```

`theType` The resource type of the icon.

`theIcon` A handle to the icon on which to perform the operation.

`yourDataPtr` A pointer to data as specified in the `yourDataPtr` parameter of the `ForEachIconDo` function. When your application calls `ForEachIconDo`, it typically provides in the `yourDataPtr` parameter a value that identifies the action your function should perform.

DESCRIPTION

The `ForEachIconDo` function uses your icon action function to perform actions on specified icons in an icon suite. Your icon action function should return a result code indicating whether it successfully performed the action on the icon.

RESULT CODE

`noErr` 0 No error

SEE ALSO

For a description of the `ForEachIconDo` function, see page 5-38.

Icon Getter Functions

If you use icon caches, you must provide at least one icon getter function. You provide a pointer to an icon getter function as a parameter to the `MakeIconCache` function. Subsequent calls to Icon Utilities routines that use icon types not present in the icon cache use the icon getter function associated with the icon cache to return a handle to the icon data.

You can also specify an icon getter function as a parameter to Icon Utilities routines that end in `Method`. Like Icon Utilities routines that work with icon caches, the icon getter function that you provide as a parameter to `PlotIconMethod` should return a handle to the requested icon's data. Note that the icon getter function that you provide as a parameter to `IconMethodToRgn`, `PtInIconMethod`, and `RectInIconMethod` should also return a handle to the requested icon; these three functions then extract the icon mask from the icon data your icon getter function returns.

MyIconGetter

Here is the syntax of an icon getter function:

```
FUNCTION MyIconGetter (theType: ResType;
                      yourDataPtr: Ptr): Handle;
```

theType The resource type of the icon.

yourDataPtr

If your icon getter was called by an icon cache routine, this parameter contains a pointer to the data associated with the icon cache. Otherwise, this parameter contains the value your application specified in the `yourDataPtr` parameter. For icon caches, you initially set this value when you first create a cache using `MakeIconCache`. You can change this value using `SetIconCacheData`. The icon getter function can use this data as needed.

DESCRIPTION

An icon getter function should return as its function result a handle to the requested icon's data.

The `MakeIconCache` function takes a pointer to an icon getter function for use with a new icon cache. To get and set an existing icon cache's icon getter function, use the `GetIconCacheProc` and `SetIconCacheProc` functions. You can also specify an icon getter function for use by the `PlotIconMethod`, `IconMethodToRgn`, `PtInIconMethod`, and `RectInIconMethod` functions.

SEE ALSO

For descriptions of the `MakeIconCache`, `GetIconCacheProc`, and `SetIconCacheProc` functions, see "Working With Icon Caches" beginning on page 5-53.

For information on the `PlotIconMethod` function, see page 5-22. For a description of the `IconMethodToRgn` function, see "Converting an Icon Mask to a Region" beginning on page 5-43.

For descriptions of the `PtInIconMethod` and `RectInIconMethod` functions, see "Determining Whether a Point or Rectangle Is Within an Icon" beginning on page 5-46.

Summary of the Icon Utilities

Pascal Summary

Constants

CONST

```

gestaltIconUtilitiesAttr  = 'icon'; {Icon Utilities attributes}
gestaltIconUtilitiesPresent= 0;      {check this bit in the }
                                   { response parameter}

{types for icon families}
large1BitMask             = 'ICN#'; {icon list resource for large icons}
large4BitData             = 'icl4'; {large 4-bit color icon resource}
large8BitData             = 'icl8'; {large 8-bit color icon resource}
small11BitMask           = 'ics#'; {icon list resource for small icons}
small4BitData            = 'ics4'; {small 4-bit color icon resource}
small8BitData            = 'ics8'; {small 8-bit color icon resource}
mini1BitMask             = 'icm#'; {icon list resource for mini icons}
mini4BitData             = 'icm4'; {4-bit color mini icon}
mini8BitData             = 'icm8'; {8-bit color mini icon resource}

{IconAlignmentType values}
atNone                   = $0;      {no alignment}
atVerticalCenter         = $1;      {centered vertically}
atTop                    = $2;      {top aligned}
atBottom                 = $3;      {bottom aligned}
atHorizontalCenter       = $4;      {centered horizontally}
atLeft                   = $8;      {left aligned}
atRight                  = $C;      {right aligned}
atAbsoluteCenter         = (atVerticalCenter + atHorizontalCenter);
atCenterTop              = (atTop + atHorizontalCenter);
atCenterBottom           = (atBottom + atHorizontalCenter);
atCenterLeft             = (atVerticalCenter + atLeft);
atTopLeft                = (atTop + atLeft);
atBottomLeft             = (atBottom + atLeft);
atCenterRight            = (atVerticalCenter + atRight);
atTopRight               = (atTop + atRight);
atBottomRight            = (atBottom + atRight);

```

Icon Utilities

```

{IconTransformType values}
ttNone                = $0;
ttDisabled            = $1;
ttOffline             = $2;
ttOpen                = $3;
ttLabel1             = $0100;
ttLabel2             = $0200;
ttLabel3             = $0300;
ttLabel4             = $0400;
ttLabel5             = $0500;
ttLabel6             = $0600;
ttLabel7             = $0700;
ttSelected           = $4000;
ttSelectedDisabled   = (ttSelected + ttDisabled);
ttSelectedOffline    = (ttSelected + ttOffline);
ttSelectedOpen       = (ttSelected + ttOpen);

{IconSelectorValue masks}
svLarge1Bit          = $00000001; {'ICN#' resource}
svLarge4Bit          = $00000002; {'icl4' resource}
svLarge8Bit          = $00000004; {'icl8' resource}
svSmall11Bit         = $00000100; {'ics#' resource}
svSmall4Bit          = $00000200; {'ics4' resource}
svSmall8Bit          = $00000400; {'ics8' resource}
svMini1Bit           = $00010000; {'icm#' resource}
svMini4Bit           = $00020000; {'icm4' resource}
svMini8Bit           = $00040000; {'icm8' resource}
svAllLargeData       = $000000FF; {'ICN#', 'icl4', and 'icl8' }
                        { resources}

svAllSmallData       = $0000FF00; {'ics#', 'ics4', and 'ics8' }
                        { resources}
svAllMiniData        = $00FF0000; {'icm#', 'icm4', and 'icm8' }
                        { resources}

svAll11BitData       = (svLarge1Bit + svSmall11Bit + svMini1Bit);
svAll4BitData        = (svLarge4Bit + svSmall4Bit + svMini4Bit);
svAll8BitData        = (svLarge8Bit + svSmall8Bit + svMini8Bit);
svAllAvailableData   = $FFFFFFFF; {all resources of given ID}

```

Data Types

TYPE

```

CIcon =
RECORD
    iconPMap:      PixMap;          {the icon's pixel map}
    iconMask:      BitMap;          {the icon's mask}
    iconBMap:      BitMap;          {the icon's bitmap}
    iconData:      Handle;          {handle to the icon's data}
    iconMaskData:  ARRAY[0..0] OF Integer;
END;

CIconPtr          = ^CIcon;        {pointer to color icon record}
CIconHandle       = ^CIconPtr;    {handle to color icon record}

IconSelectorValue = LongInt;      {icon selector type}
IconAlignmentType = Integer;      {icon alignment type}
IconTransformType = Integer;      {icon transform type}

IconAction        = ProcPtr;       {pointer to action function}
IconGetter        = ProcPtr;       {pointer to icon getter function}

```

Icon Utilities Routines

Drawing Icons From Resources

```

FUNCTION PlotIconID      (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theResID: Integer): OSErr;

FUNCTION PlotIconMethod  (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theMethod: IconGetter;
                        yourDataPtr: UNIV Ptr): OSErr;

PROCEDURE PlotIcon      (theRect: Rect; theIcon: Handle);

FUNCTION PlotIconHandle  (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theIcon: Handle): OSErr;

PROCEDURE PlotCIcon     (theRect: Rect; theIcon: CIconHandle);

FUNCTION PlotCIconHandle (theRect: Rect; align: IconAlignmentType;
                        transform: IconTransformType;
                        theCIcon: CIconHandle): OSErr;

```

```
FUNCTION PlotSICNHandle (theRect: Rect; align: IconAlignmentType;
                       transform: IconTransformType;
                       theSICN: Handle): OSErr;
```

Getting Icons From Resources That Don't Belong to an Icon Family

```
FUNCTION GetIcon (iconID: Integer): Handle;
FUNCTION GetCIcon (iconID: Integer): CIconHandle;
```

Disposing of Icons

```
PROCEDURE DisposeCIcon (theIcon: CIconHandle);
```

Creating an Icon Suite

```
FUNCTION GetIconSuite (VAR theIconSuite: Handle; theResID: Integer;
                      selector: IconSelectorValue): OSErr;
FUNCTION NewIconSuite (VAR theIconSuite: Handle): OSErr;
FUNCTION AddIconToSuite (theIconData: Handle; theSuite: Handle;
                        theType: ResType): OSErr;
```

Getting Icons From an Icon Suite

```
FUNCTION GetIconFromSuite (VAR theIconData: Handle; theSuite: Handle;
                           theType: ResType): OSErr;
```

Drawing Icons From an Icon Suite

```
FUNCTION PlotIconSuite (theRect: Rect; align: IconAlignmentType;
                       transform: IconTransformType;
                       theIconSuite: Handle): OSErr;
```

Performing Operations on Icons in an Icon Suite

```
FUNCTION ForEachIconDo (theSuite: Handle; selector: IconSelectorValue;
                       action: IconAction; yourDataPtr: Ptr): OSErr;
```

Getting and Setting the Label for an Icon Suite

```
FUNCTION GetSuiteLabel (theSuite: Handle): Integer;
FUNCTION SetSuiteLabel (theSuite: Handle; theLabel: Integer): OSErr;
```

Getting Label Information

```
FUNCTION GetLabel (labelNumber: Integer; VAR labelColor: RGBColor;
                  VAR labelString: Str255): OSErr;
```

Disposing of Icon Suites

```
FUNCTION DisposeIconSuite (theIconSuite: Handle;
                          disposeData: Boolean): OSErr;
```

Converting an Icon Mask to a Region

```
FUNCTION IconSuiteToRgn (theRgn: RgnHandle; iconRect: Rect;
                        align: IconAlignmentType;
                        theIconSuite: Handle): OSErr;
```

```
FUNCTION IconIDToRgn (theRgn: RgnHandle; iconRect: Rect;
                     align: IconAlignmentType;
                     iconID: Integer): OSErr;
```

```
FUNCTION IconMethodToRgn (theRgn: RgnHandle; iconRect: Rect;
                          align: IconAlignmentType;
                          theMethod: IconGetter;
                          yourDataPtr: Ptr): OSErr;
```

Determining Whether a Point or Rectangle Is Within an Icon

```
FUNCTION PtInIconSuite (testPt: Point; iconRect: Rect;
                       align: IconAlignmentType;
                       theIconSuite: Handle): Boolean;
```

```
FUNCTION PtInIconID (testPt: Point; iconRect: Rect;
                    align: IconAlignmentType;
                    iconID: Integer): Boolean;
```

```
FUNCTION PtInIconMethod (testPt: Point; iconRect: Rect;
                         align: IconAlignmentType;
                         theMethod: IconGetter;
                         yourDataPtr: Ptr): Boolean;
```

```
FUNCTION RectInIconSuite (testRect: Rect; iconRect: Rect;
                          align: IconAlignmentType;
                          theIconSuite: Handle): Boolean;
```

```
FUNCTION RectInIconID (testRect: Rect; iconRect: Rect;
                       align: IconAlignmentType;
                       iconID: Integer): Boolean;
```

```
FUNCTION RectInIconMethod (testRect: Rect; iconRect: Rect;
                           align: IconAlignmentType;
                           theMethod: IconGetter;
                           yourDataPtr: Ptr): Boolean;
```

Working With Icon Caches

```
FUNCTION MakeIconCache (VAR theHandle: Handle;
                        makeIcon: IconGetter;
                        yourDataPtr: UNIV Ptr): OSErr;
```

Icon Utilities

```

FUNCTION LoadIconCache      (theRect: Rect; align: IconAlignmentType;
                           transform: IconTransformType;
                           theIconCache: Handle): OSErr;

FUNCTION GetIconCacheData  (theCache: Handle; VAR theData: Ptr): OSErr;
FUNCTION SetIconCacheData  (theCache: Handle; theData: Ptr): OSErr;
FUNCTION GetIconCacheProc  (theCache: Handle;
                           VAR theProc: IconGetter): OSErr;
FUNCTION SetIconCacheProc  (theCache: Handle; theProc: IconGetter): OSErr;

```

Application-Defined Routines
Icon Action Functions

```

FUNCTION MyIconAction      (theType: ResType; VAR theIcon: Handle;
                           yourDataPtr: Ptr): OSErr;

```

Icon Getter Functions

```

FUNCTION MyIconGetter      (theType: ResType; yourDataPtr: Ptr): Handle;

```

C Summary

Constants

```

enum {
    #define gestaltIconUtilitiesAttr 'icon' /*Icon Utilities attributes*/
    gestaltIconUtilitiesPresent      = 0    /*check this bit in the */
                                        /* response parameter*/
};
/*types for icon families*/
#define largelBitMask                'ICN#' /*icon list resource for large icons*/
#define large4BitData                'icl4' /*large 4-bit color icon resource*/
#define large8BitData                'icl8' /*large 8-bit color icon resource*/
#define small1BitMask                'ics#' /*icon list resource for small icons*/
#define small4BitData                'ics4' /*small 4-bit color icon resource*/
#define small8BitData                'ics8' /*small 8-bit color icon resource*/
#define mini1BitMask                 'icm#' /*icon list resource for mini icons*/
#define mini4BitData                 'icm4' /*mini 4-bit color icon resource*/
#define mini8BitData                 'icm8' /*mini 4-bit color icon resource*/

```

Icon Utilities

```
enum { /*IconAlignmentType values*/
    atNone           = 0x0,           /*no alignment*/
    atVerticalCenter = 0x1,           /*centered vertically*/
    atTop            = 0x2,           /*top aligned*/
    atBottom         = 0x3,           /*bottom aligned*/
    atHorizontalCenter = 0x4,         /*centered horizontally*/
    atAbsoluteCenter = (atVerticalCenter | atHorizontalCenter),
    atCenterTop      = (atTop | atHorizontalCenter),
    atCenterBottom   = (atBottom | atHorizontalCenter),
    atLeft           = 0x8,           /*left aligned*/
    atCenterLeft     = (atVerticalCenter | atLeft),
    atTopLeft        = (atTop | atLeft),
    atBottomLeft     = (atBottom | atLeft),
    atRight          = 0xC,           /*right aligned*/
    atCenterRight    = (atVerticalCenter | atRight),
    atTopRight       = (atTop | atRight),
    atBottomRight    = (atBottom | atRight),
};
```

```
enum { /*IconTransformType values*/
    ttNone           = 0x0,
    ttDisabled       = 0x1,
    ttOffline        = 0x2,
    ttOpen           = 0x3,
    ttLabel1         = 0x0100,
    ttLabel2         = 0x0200,
    ttLabel3         = 0x0300,
    ttLabel4         = 0x0400,
    ttLabel5         = 0x0500,
    ttLabel6         = 0x0600,
    ttLabel7         = 0x0700,
    ttSelected       = 0x4000,
    ttSelectedDisabled = (ttSelected | ttDisabled),
    ttSelectedOffline = (ttSelected | ttOffline),
    ttSelectedOpen   = (ttSelected | ttOpen),
};
```

```
enum { /*IconSelectorValue masks*/
    svLarge1Bit      = 0x00000001, /*'ICN#' resource*/
    svLarge4Bit      = 0x00000002, /*'icl4' resource*/
    svLarge8Bit      = 0x00000004, /*'icl8' resource*/
    svSmall11Bit     = 0x00000100, /*'ics#' resource*/
    svSmall4Bit      = 0x00000200, /*'ics4' resource*/
};
```

Icon Utilities

```

svSmall8Bit          = 0x00000400, /*'ics8' resource*/
svMini1Bit           = 0x00010000, /*'icm#' resource*/
svMini4Bit           = 0x00020000, /*'icm4' resource*/
svMini8Bit           = 0x00040000, /*'icm8' resource*/
svAllLargeData       = 0x000000FF, /*'ICN#', 'icl4', and 'icl8' */
                      /* resources*/
svAllSmallData       = 0x0000FF00, /*'ics#', 'ics4', and 'ics8' */
                      /* resources*/
svAllMiniData        = 0x00FF0000, /*'icm#', 'icm4', and 'icm8' */
                      /* resources*/
svAll11BitData       = (svLarge1Bit | svSmall11Bit | svMini1Bit),
svAll4BitData        = (svLarge4Bit | svSmall4Bit | svMini4Bit),
svAll8BitData        = (svLarge8Bit | svSmall8Bit | svMini8Bit),
svAllAvailableData   = (long)0xFFFFFFFF /*all resources of given ID*/
};

```

Data Types

```

struct CIcon {
    PixMap iconPMap;          /*the icon's pixel map*/
    BitMap iconMask;         /*the icon's mask*/
    BitMap iconBMap;        /*the icon's bitmap*/
    Handle iconData;         /*handle to the icon's data*/
    short iconMaskData;     /*the data for the icon's mask*/
};

typedef struct CIcon CIcon;
typedef Cicon *CIconPtr, **CIconHandle; /*ptr, handle to color icon record*/

typedef unsigned long IconSelectorValue; /*icon selector type*/
typedef short IconAlignmentType;        /*icon alignment type*/
typedef short IconTransformType;        /*icon transform type*/

/*pointer to action function*/
typedef pascal OSErr (*IconActionProcPtr)(ResType theType, Handle *theIcon,
                                          void *yourDataPtr);

typedef IconActionProcPtr IconAction;

/*pointer to icon getter function*/
typedef pascal Handle (*IconGetterProcPtr)(ResType theType,
                                          void *yourDataPtr);

typedef IconGetterProcPtr IconGetter;

```

Icon Utilities Routines

Drawing Icons From Resources

```

pascal OSErr PlotIconID      (const Rect *theRect, IconAlignmentType align,
                             IconTransformType transform, short theResID);

pascal OSErr PlotIconMethod (const Rect *theRect, IconAlignmentType align,
                             IconTransformType transform,
                             IconGetterProcPtr theMethod,
                             void *yourDataPtr);

pascal void PlotIcon        (const Rect *theRect, Handle theIcon);

pascal OSErr PlotIconHandle (const Rect *theRect, IconAlignmentType align,
                             IconTransformType transform, Handle theIcon);

pascal OSErr PlotCIcon      (const Rect *theRect, CIconHandle theIcon);

pascal OSErr PlotCIconHandle (const Rect *theRect, IconAlignmentType align,
                              IconTransformType transform,
                              CIconHandle theCIcon);

pascal OSErr PlotSICNHandle (const Rect *theRect, IconAlignmentType align,
                              IconTransformType transform, Handle theSICN);

```

Getting Icons From Resources That Don't Belong to an Icon Family

```

pascal Handle GetIcon      (short iconID);
pascal CIconHandle GetCIcon (short iconID);

```

Disposing of Icons

```

pascal OSErr DisposeCIcon (CIconHandle theIcon);

```

Creating an Icon Suite

```

pascal OSErr GetIconSuite (Handle *theIconSuite, short theResID,
                           IconSelectorValue selector);

pascal OSErr NewIconSuite (Handle *theIconSuite);

```

Icon Utilities

```
pascal OSErr AddIconToSuite
                                (Handle theIconData, Handle theSuite,
                                 ResType theType);
```

Getting Icons From an Icon Suite

```
pascal OSErr GetIconFromSuite
                                (Handle *theIconData, Handle theSuite,
                                 ResType theType);
```

Drawing Icons From an Icon Suite

```
pascal OSErr PlotIconSuite (const Rect *theRect, IconAlignmentType align,
                             IconTransformType transform,
                             Handle theIconSuite);
```

Performing Operations on Icons in an Icon Suite

```
pascal OSErr ForEachIconDo (Handle theSuite, IconSelectorValue selector,
                             IconActionProcPtr action, void *yourDataPtr);
```

Getting and Setting the Label for an Icon Suite

```
pascal short GetSuiteLabel (Handle theSuite);
pascal OSErr SetSuiteLabel (Handle theSuite, short theLabel);
```

Getting Label Information

```
pascal OSErr GetLabel (short labelNumber, RGBColor *labelColor,
                       Str255 labelString);
```

Disposing of Icon Suites

```
pascal OSErr DisposeIconSuite
                                (Handle theIconSuite, Boolean disposeData);
```

Converting an Icon Mask to a Region

```
pascal OSErr IconSuiteToRgn
                                (RgnHandle theRgn, const Rect *iconRect,
                                 IconAlignmentType align, Handle theIconSuite);
pascal OSErr IconIDToRgn (RgnHandle theRgn, const Rect *iconRect,
                          IconAlignmentType align, short iconID);
```

Icon Utilities

```
pascal OSErr IconMethodToRgn
    (RgnHandle theRgn, const Rect *iconRect,
     IconAlignmentType align,
     IconGetterProcPtr theMethod,
     void *yourDataPtr);
```

Determining Whether a Point or Rectangle Is Within an Icon

```
pascal Boolean PtInIconSuite
    (Point testPt, const Rect *iconRect,
     IconAlignmentType align, Handle theIconSuite);
pascal Boolean PtInIconID   (Point testPt, const Rect *iconRect,
                             IconAlignmentType align, short iconID);
pascal Boolean PtInIconMethod
    (Point testPt, const Rect *iconRect,
     IconAlignmentType align,
     IconGetterProcPtr theMethod,
     void *yourDataPtr);
pascal Boolean RectInIconSuite
    (const Rect *testRect, const Rect *iconRect,
     IconAlignmentType align, Handle theIconSuite);
pascal Boolean RectInIconID   (const Rect *testRect, const Rect *iconRect,
                             IconAlignmentType align, short iconID);
pascal Boolean RectInIconMethod
    (const Rect *testRect, const Rect *iconRect,
     IconAlignmentType align,
     IconGetterProcPtr theMethod,
     void *yourDataPtr);
```

Working With Icon Caches

```
pascal OSErr MakeIconCache (Handle *theHandle, IconGetterProcPtr makeIcon,
                             void *yourDataPtr);
pascal OSErr LoadIconCache (const Rect *theRect, IconAlignmentType align,
                             IconTransformType transform,
                             Handle theIconCache);
```

Icon Utilities

```

pascal OSErr GetIconCacheData
                                (Handle theCache, void **theData);
pascal OSErr SetIconCacheData
                                (Handle theCache, void *theData);
pascal OSErr GetIconCacheProc
                                (Handle theCache, IconGetter *theProc);
pascal OSErr SetIconCacheProc
                                (Handle theCache, IconGetter theProc);

```

Application-Defined Routines

Icon Action Functions

```

pascal OSErr MyIconAction (ResType theType, Handle *theIcon,
                           void *yourDataPtr);

```

Icon Getter Functions

```

pascal Handle MyIconGetter (ResType theType, void *yourDataPtr);

```

Assembly-Language Summary

Data Structure

Color Icon Data Structure

0	iconPMap	60 bytes	icon's pixel map
50	iconMask	14 bytes	icon's mask
64	iconBMap	14 bytes	icon's bitmap
78	iconData	4 bytes	handle to icon's data
82	iconMaskData	variable	data for icon's mask

Trap Macros

Trap Macros Requiring Routine Selectors

_IconDispatch

Selector	Routine
\$0702	NewIconSuite
\$1702	GetSuiteLabel
\$0203	DisposeIconSuite
\$1603	SetSuiteLabel
\$1904	GetIconCacheData
\$1A04	SetIconCacheData
\$1B04	GetIconCacheProc
\$1C04	SetIconCacheProc
\$0005	PlotIconID
\$0105	GetIconSuite
\$0B05	GetLabel
\$0306	PlotIconSuite
\$0406	MakeIconCache
\$0606	LoadIconCache
\$0806	AddIconToSuite
\$0906	GetIconFromSuite
\$0D06	PtInIconID
\$1006	RectInIconID
\$1306	IconIDToRgn
\$1D06	PlotIconHandle
\$1E06	PlotSICNHandle
\$1F06	PlotCIconHandle
\$0E07	PtInIconSuite
\$1107	RectInIconSuite
\$1407	IconSuiteToRgn
\$0A08	ForEachIconDo
\$0508	PlotIconMethod
\$0F09	PtInIconMethod
\$1209	RectInIconMethod
\$1509	IconMethodToRgn

Result Codes

noErr	0	No error
paramErr	-50	Error in parameter list
memFullErr	-108	Not enough memory in heap zone
memWZErr	-111	Attempt to operate on a free block
resNotFound	-192	Resource not found
noMaskFoundErr	-1000	Cannot find or create mask for the icon family

Component Manager

Contents

Introduction to Components	6-3
About the Component Manager	6-4
Using the Component Manager	6-6
Opening Connections to Components	6-7
Opening a Connection to a Default Component	6-7
Finding a Specific Component	6-8
Opening a Connection to a Specific Component	6-9
Getting Information About a Component	6-10
Using a Component	6-11
Closing a Connection to a Component	6-12
Creating Components	6-13
The Structure of a Component	6-13
Handling Requests for Service	6-18
Responding to the Open Request	6-19
Responding to the Close Request	6-21
Responding to the Can Do Request	6-22
Responding to the Version Request	6-22
Responding to the Register Request	6-23
Responding to the Unregister Request	6-24
Responding to the Target Request	6-25
Responding to Component-Specific Requests	6-26
Reporting an Error Code	6-28
Defining a Component's Interfaces	6-28
Managing Components	6-30
Registering a Component	6-30
Creating a Component Resource	6-32
Establishing and Managing Connections	6-34
Component Manager Reference	6-37
Data Structures for Applications	6-37
The Component Description Record	6-37

Component Identifiers and Component Instances	6-40
Routines for Applications	6-41
Finding Components	6-42
Opening and Closing Components	6-44
Getting Information About Components	6-47
Retrieving Component Errors	6-51
Data Structures for Components	6-52
The Component Description Record	6-52
The Component Parameters Record	6-54
Routines for Components	6-56
Registering Components	6-57
Dispatching to Component Routines	6-63
Managing Component Connections	6-65
Setting Component Errors	6-69
Working With Component Reference Constants	6-70
Accessing a Component's Resource File	6-71
Calling Other Components	6-73
Capturing Components	6-75
Targeting a Component Instance	6-77
Changing the Default Search Order	6-78
Application-Defined Routine	6-79
Resources	6-80
The Component Resource	6-80
Summary of the Component Manager	6-86
Pascal Summary	6-86
Constants	6-86
Data Types	6-87
Routines for Applications	6-89
Routines for Components	6-90
Application-Defined Routine	6-92
C Summary	6-92
Constants	6-92
Data Structures	6-93
Routines for Applications	6-95
Routines for Components	6-96
Application-Defined Routine	6-97
Assembly-Language Summary	6-98
Trap Macros	6-98
Result Codes	6-99

Component Manager

This chapter describes how you can use the Component Manager to allow your application to find and utilize various software objects (components) at run time. It also discusses how you can create your own components and how you can use the Component Manager to help manage your components. You should read this chapter if you are developing an application that uses components or if you plan to develop your own components.

The rest of this chapter

- n contains a general introduction to components and the features provided by the Component Manager
- n discusses how to use the facilities of the Component Manager to call components
- n describes how to create a component

Several of the sections in this chapter are divided into two main topics: one describes how applications can use components, and one describes how to create your own components. If you are developing an application that uses components, you should focus on the material that describes how to use existing components—you do not need to read the material that describes how to create a component. If you are developing a component, however, you should be familiar with all the information in this chapter.

For information on a specific component, see the documentation supplied with that component. For example, for information on the components that Apple supplies with QuickTime, see *Inside Macintosh: QuickTime Components*.

Introduction to Components

A **component** is a piece of code that provides a defined set of services to one or more clients. Applications, system extensions, as well as other components can use the services of a component. A component typically provides a specific type of service to its clients. For example, a component might provide image compression or image decompression capabilities; an application could call such a component, providing the image to compress, and the component could perform the desired operation and return the compressed image to the application.

Multiple components can provide the same type of service. For example, separate components might exist that can compress an image by 20 percent, 40 percent, or 50 percent, with varying degrees of fidelity. All components of the same type must support the same basic interface. This allows your application to use the same interface for any given type of component and get the same type of service, yet allows your application to obtain different levels of service.

The Component Manager provides access to components and manages them by, for example, keeping track of the currently available components and routing requests to the appropriate component.

Component Manager

The Component Manager classifies components by three main criteria: the type of service provided, the level of service provided, and the component manufacturer. The Component Manager uses a **component type** to identify the type of service provided by a component. Like resource types, a component type is a sequence of four characters. All components of the same component type provide the same type of services and support a common application interface. For example, all image compressor components have a component type of 'imco'. Other types of components include video digitizers, timing sources, movie controllers, and sequence capturers.

Note

Component types consisting of only lowercase characters are reserved for definition by Apple. You can define component types using other combinations of characters, but you must register any new component types with Apple's Component Registry Group (AppleLink REGISTRY). ^u

The Component Manager allows components to identify variations on the basic interface they must support by specifying a four-character **component subtype**. The value of the component subtype is meaningful only in the context of a given component type. For example, image compressor components use the component subtype to specify the compression algorithm supported by the component.

All components of a given type-subtype combination must support a common application interface. However, components that share a type-subtype specification may support routines that are not part of the basic interface defined for their type. In this manner, components can provide enhanced services to client applications while still supporting the basic application interface.

Finally, the Component Manager allows components to have a four-character manufacturer code that identifies the manufacturer of the component. You must register your component with Apple's Component Registry Group to receive a manufacturer code for your component. The manufacturer code allows applications to further distinguish between components of the same type-subtype.

About the Component Manager

The Component Manager provides services that allow applications to obtain run-time location of and access to functional objects (in much the same way that the Resource Manager allows applications that are running to access data objects dynamically).

The Component Manager creates an interface between components and clients, which can be applications, other components, system extensions, and so on. Instead of implementing support for a particular data format, protocol, or model of a device, you can use a standard interface through which your application communicates with all components of a given type. You can then use the Component Manager to locate and communicate with components of that type. Those components, in turn, provide the appropriate services to your client application.

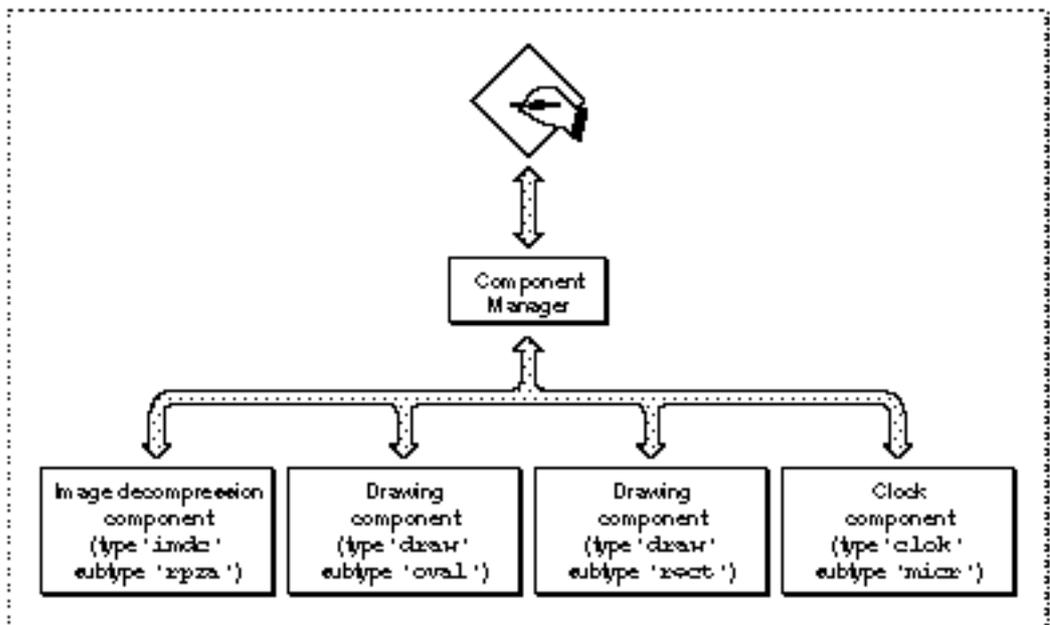
Component Manager

Given a particular component type, the Component Manager can locate and query all components of that type. You can find out how many components of a specific type are available and you can get further details about a component's capabilities without having to open it first. For each component, the Component Manager keeps track of many characteristics, including its name, icon, and information string.

For example, components of type 'imdc' provide image decompression services. All components of type 'imdc' share a common application interface, but each image decompressor component may support a unique compression technique or take advantage of a special hardware implementation. Individual components may support additions to the defined application interface, as long as they support the common routines. Any algorithm-dependent or implementation-dependent variations of the general decompression interface can be implemented by each 'imdc' component as extensions to the basic interface.

Figure 6-1 shows the relationship between an application, the Component Manager, and several components. Applications and other clients use the Component Manager to access components. In this figure, four components are available to the application: an image decompression component (of type 'imdc'), two drawing components (of type 'draw'), and a clock component (of type 'clock'). Note that the two drawing components have different subtypes: 'oval' and 'rect'. The drawing component with subtype 'oval' draws ovals, and the drawing component with subtype 'rect' draws rectangles.

Figure 6-1 The relationship between an application, the Component Manager, and components



Component Manager

The Component Manager allows a single component to serve multiple client applications at the same time. Each client application has a unique access path to the component. These access paths are called **component connections**. You identify a component connection by specifying a **component instance**. The Component Manager provides this component instance to your application when you open a connection to a component. The component maintains separate status information for each open connection.

For example, multiple applications might each open a connection to an image decompression component. The Component Manager routes each application request to the component instance for that connection. Because a component can maintain separate storage for each connection, application requests do not interfere with each other and each application has full access to the services provided by the component. (See Figure 6-2 on page 6-34 for an illustration of multiple applications using the services of the same component.)

Using the Component Manager

This section describes how you can use the Component Manager to

- n gain access to components
- n locate components and take advantage of their services
- n get information about a component
- n close a connection to a component

The Component Manager is available in System 7.1 or later and may be present in System 7. To determine whether the Component Manager is available, call the `Gestalt` function with the `gestaltComponentMgr` selector and check the value of the `response` parameter.

```
CONST
    gestaltComponentMgr    = 'cpnt';
```

The `Gestalt` function returns in the `response` parameter a 32-bit value indicating the version of the Component Manager that is installed. Version 3 and above supports automatic version control, the `unregister` request, and icon families. You should test the version number before using any of these features.

This section presents several examples demonstrating how to use components and the Component Manager. All of these examples use the services of a drawing component—a simple component that draws an object of a particular shape on the screen. Drawing components have a component type of `'draw'`. The component subtype value indicates the type of object the component draws. For example, a drawing component that draws

an oval has a component subtype of 'oval'. For information on creating your own components and for listings that show the code for a drawing component, see “Creating Components” beginning on page 6-13.

Opening Connections to Components

When your application requires the services of a component, you typically perform these steps:

- n open a connection to the desired component
- n use the services of the component
- n close the connection to the component

The following sections describe each of these steps in more detail.

Opening a Connection to a Default Component

Your application must use the Component Manager to gain access to a component. The first step is to locate an appropriate component. You can locate the component yourself, or you can allow the Component Manager to locate a suitable component for you. Your application then opens a connection to that component. Once you have opened a connection to a component, you can use the services provided by that component. When you have finished using the component, you should close the connection.

If you are interested only in using a component of a particular type-subtype and you do not need to specify any other characteristics of the component, use the `OpenDefaultComponent` function and specify only the component type and subtype—the Component Manager then selects a component for you and opens a connection to that component. This is the easiest technique for opening a component connection. The `OpenDefaultComponent` function searches its list of available components and attempts to open a connection to a component with the specified type and subtype. If more than one component of the specified type and subtype is available, `OpenDefaultComponent` selects the first one in the list. If successful, the `OpenDefaultComponent` function returns a component instance that identifies your connection to the component. You can then use that connection to employ the services of the selected component.

This code demonstrates the use of the `OpenDefaultComponent` function. The code opens a connection to a component of type 'draw' and subtype 'oval'—a drawing component that draws an oval.

```
VAR
    aDrawOvalComp: ComponentInstance;

    aDrawOvalComp := OpenDefaultComponent('draw', 'oval');
```

Component Manager

If it cannot find or open a component of the specified type-subtype, the `OpenDefaultComponent` function returns a function result of `NIL`.

To open a connection to a component with a specific type-subtype-manufacturer code or with other specified characteristics, first use the `FindNextComponent` function to find the desired component, then open the component using the `OpenComponent` function. These operations are described in the next two sections.

Finding a Specific Component

If you are interested in asserting greater control over the selection of a component, you can use the Component Manager to find a component that provides a specified service. For example, you can use the `FindNextComponent` function in a loop to retrieve information about all the components that are registered on a given computer. Each time you call this function, the Component Manager returns information about a single component. You can obtain a count of all the components on a given computer by calling the `CountComponents` function. Both of these functions allow you to specify search criteria, for example, by component type and subtype, or by manufacturer. By using these criteria to narrow your search, you can quickly and easily find a component that meets your needs.

You specify the search criteria for the component using a component description record. A component description record is defined by the `ComponentDescription` data type. For more information on the fields of this record, see “The Component Description Record” beginning on page 6-37.

TYPE

```
ComponentDescription =
  RECORD
    componentType:      OSType;      {type}
    componentSubType:   OSType;      {subtype}
    componentManufacturer: OSType;    {manufacturer}
    componentFlags:     LongInt;     {control flags}
    componentFlagsMask: LongInt;     {mask for flags}
  END;
```

By default, the Component Manager considers all fields of the component description record when performing a search. Your application can override the default behavior of which fields the Component Manager considers for a search. Specify 0 in any field of the component description record to prevent the Component Manager from considering the information in that field when performing the search.

Listing 6-1 shows an application-defined procedure, `MyFindVideoComponent`, that fills out a component description record to specify the search criteria for the desired component. The `MyFindVideoComponent` procedure then uses the `FindNextComponent` function to return the first component with the specified characteristics—in this example, any component with the type `VideoDigitizerComponentType`.

Listing 6-1 Finding a component

```
PROCEDURE MyFindVideoComponent(VAR videoCompID: Component);
VAR
    videoDesc: ComponentDescription;
BEGIN
    {find a video digitizer component}
    videoDesc.componentType := VideoDigitizerComponentType;
    videoDesc.componentSubType := OSType(0);      {any subtype}
    videoDesc.componentManufacturer:= OSType(0); {any manufacturer}
    videoDesc.componentFlags := 0;
    videoDesc.componentFlagsMask := 0;
    videoCompID := FindNextComponent(Component(0), videoDesc);
END;
```

The `FindNextComponent` function requires two parameters: a value that indicates which component to begin the search with and a component description record. You can specify 0 in the first parameter to start the search at the beginning of the component list. Alternatively, you can specify a component identifier obtained from a previous call to `FindNextComponent`.

The `FindNextComponent` function returns a component identifier to your application. The returned component identifier identifies a given component to the Component Manager. You can use this identifier to retrieve more information about the component or to open a connection to the component. The next two sections describe these tasks.

Opening a Connection to a Specific Component

You can open a connection to a specific component by calling the `OpenComponent` function (alternatively, you can use the `OpenDefaultComponent` function, as discussed in “Opening a Connection to a Default Component” on page 6-7). Your application must provide a component identifier to the `OpenComponent` function. You get a component identifier from the `FindNextComponent` function, as described in the previous section.

Component Manager

The `OpenComponent` function returns a component instance that identifies your connection to the component. Listing 6-2 shows how to use the `OpenComponent` function to gain access to a specific component. The application-defined procedure `MyGetComponent` uses the `MyFindVideoComponent` procedure (defined in Listing 6-1) to find a video digitizer component and then opens the component.

Listing 6-2 Opening a specific component

```
PROCEDURE MyGetComponent
    (VAR videoCompInstance: ComponentInstance);
VAR
    videoCompID:      Component;
BEGIN
    {first find a video digitizer component}
    MyFindVideoComponent(videoCompID);
    {now open it}
    IF videoCompID <> NIL THEN
        videoCompInstance := OpenComponent(videoCompID);
    END;
```

Getting Information About a Component

You can use the `GetComponentInfo` function to retrieve information about a component, including the component name, icon, and other information. Listing 6-3 shows an application-defined procedure that retrieves information about a video digitizer component.

Listing 6-3 Getting information about a component

```
PROCEDURE MyGetCompInfo (compName, compInfo, compIcon: Handle;
    VAR videoDesc: ComponentDescription);
VAR
    videoCompID:      Component;
    myErr:            OSErr;
BEGIN
    {first find a video digitizer component}
    MyFindVideoComponent(videoCompID);
    {now get information about it}
    IF videoCompID <> NIL THEN
        myErr := GetComponentInfo(videoCompID, videoDesc, compName,
            compInfo, compIcon);
    END;
```

Component Manager

You specify the component in the first parameter to `GetComponentInfo`. You specify the component using either a component identifier (obtained from `FindNextComponent` or `RegisterComponent`) or a component instance (obtained from `OpenDefaultComponent` or `OpenComponent`).

The `GetComponentInfo` function returns information about the component in the second through fifth parameters of the function. The `GetComponentInfo` function returns information about the component (such as its type, subtype, and manufacturer) in a component description record. The function also returns the component name, icon, and other information through handles. You must allocate these handles before calling `GetComponentInfo`. (Alternatively, you can specify `NIL` in the `compName`, `compInfo`, and `compIcon` parameters if you do not want the information returned.) The icon returned in the `compIcon` parameter is a handle to a black-and-white icon. If a component has an icon family, you can retrieve a handle to its icon suite using `GetComponentIconSuite`.

Using a Component

Once you have established a connection to a component, you can use its services.

Each time you call a component routine, you must specify the component instance that identifies your connection and provide any other parameters as required by the routine.

For example, Listing 6-4 illustrates the use of a drawing component. The application-defined procedure establishes a connection to a drawing component, calls the component's `DrawerSetup` function to establish the rectangle in which to draw the desired object, and then draws the object using the `DrawerDraw` function.

Listing 6-4 Using a drawing component

```
PROCEDURE MyDrawAnOval (VAR aDrawOvalComp: ComponentInstance);
VAR
    r:          Rect;
    result:     ComponentResult;
BEGIN
    {open a connection to a drawing component}
    aDrawOvalComp := OpenDefaultComponent('draw', 'oval');
    IF aDrawOvalComp <> NIL THEN
        BEGIN
            SetRect(r, 40, 40, 80, 80);
            {set up rectangle for oval}
            result := DrawerSetup(aDrawOvalComp, r);
            IF result = noErr THEN
                result := DrawerDraw(aDrawOvalComp); {draw oval}
        END;
    END;
END;
```

Component Manager

If you specify an invalid connection as a parameter to a component routine, the Component Manager sets the function result of the component routine to `badComponentInstance`.

Each component type supports a defined set of functions. You must refer to the appropriate documentation for a description of the functions supported by a component. You also need to refer to the component's documentation for information on the appropriate interface files that you must include to use the component (the interface files for the drawing component are shown beginning on page 6-28). The components that Apple provides with QuickTime are described in *Inside Macintosh: QuickTime Components*. As an example, drawing components support the following functions:

```
FUNCTION DrawerSetup(myInstance: ComponentInstance;
                    VAR r: Rect): ComponentResult;
FUNCTION DrawerClick(myInstance: ComponentInstance;
                    p: Point): ComponentResult;
FUNCTION DrawerMove (myInstance: ComponentInstance; x: Integer;
                    y: Integer): ComponentResult;
FUNCTION DrawerDraw (myInstance: ComponentInstance)
                    : ComponentResult;
FUNCTION DrawerErase(myInstance: ComponentInstance)
                    : ComponentResult;
```

Closing a Connection to a Component

When you finish using a component, you must close your connection to that component. Use the `CloseComponent` function to close the connection. For example, this code calls the application-defined procedure `MyDrawAnOval` (see Listing 6-4), which opens a connection to a drawing component and uses that component to draw an oval. This code closes the oval drawer component after it is finished using it.

```
VAR
    aDrawOvalComp: ComponentInstance;
    result: OSErr;

MyDrawAnOval(aDrawOvalComp);    {open component and draw an oval}
result := DrawerErase(aDrawOvalComp); {erase the oval}
result := CloseComponent(aDrawOvalComp); {close the component}
```

Creating Components

This section describes how to create a component and how your component interacts with the Component Manager. This section also describes many of the routines that the Component Manager provides to help you manage your component. If you are developing a component, you should read the material in this section.

If you are developing an application that uses components, you may find this material interesting, but you do not need to be familiar with it. You should read the preceding section, “Using the Component Manager,” and then use the “Component Manager Reference” section as needed.

This section discusses how you can

- n structure your component
- n respond to requests from the Component Manager
- n define the functions that applications may call to request service from your component
- n manage your component with the help of the Component Manager
- n make your component available for use by applications

This section presents several examples demonstrating how to create components and register them with the Component Manager. All of these examples are based on a “drawing component”—a simple component that draws an object of a particular shape on the screen. This section includes the code for a drawing component.

The Structure of a Component

Every component must have a single entry point that returns a value of type `ComponentResult` (a long integer). Whenever the Component Manager receives a request for your component, it calls your component’s entry point and passes any parameters, along with information about the current connection, in a component parameters record. The Component Manager also passes a handle to the global storage (if any) associated with that instance of your component.

When your component receives a request, it should examine the parameters to determine the nature of the request, perform the appropriate processing, set an error code if necessary, and return an appropriate function result to the Component Manager.

Component Manager

The component parameters record is defined by a data structure of type `ComponentParameters`.

```

TYPE ComponentParameters =
    PACKED RECORD
        flags:      Char;                {reserved}
        paramSize:  Char;                {size of parameters}
        what:       Integer;             {request code}
        params:     ARRAY[0..0] OF LongInt; {actual parameters}
    END;

```

The `what` field contains a value that specifies the type of request. Negative values are reserved for definition by Apple. You can use values greater than or equal to 0 to define other requests that are supported by your component. Follow these guidelines when defining your request codes: request codes between 0 and 256 are reserved for definition by components of a given type and a given type-subtype. Use request codes greater than 256 for requests that are unique to your component. For example, a certain component of a certain type-subtype might support values 0 through 5 as requests that are supported by all components of that type, values 128 through 140 as requests that are supported by all components of that given type-subtype, and values 257 through 260 as requests supported only by that component.

Table 6-1 shows the request codes defined by Apple and the actions your component should take upon receiving them. Note that four of the request codes—`open`, `close`, `can do`, and `version`—are required. Your component must respond to these four request codes. These request codes are described in greater detail in “Handling Requests for Service” beginning on page 6-18.

Table 6-1 Request codes

Request code	Action your component should perform	Required
<code>kComponentOpenSelect</code>	Open a connection	Yes
<code>kComponentCloseSelect</code>	Close an open connection	Yes
<code>kComponentCanDoSelect</code>	Determine whether your component supports a particular request	Yes
<code>kComponentVersionSelect</code>	Return your component’s version number	Yes
<code>kComponentRegisterSelect</code>	Determine whether your component can operate in the current environment	No

Table 6-1 Request codes (continued)

Request code	Action your component should perform	Required
kComponentTargetSelect	Call another component whenever it would call itself (as a result of your component being used by another component)	No
kComponentUnregisterSelect	Perform any operations that are necessary as a result of your component being unregistered	No

The example drawing component (shown in Listing 6-5 on page 6-16) supports the four required request codes, and in addition supports the request codes that are required for all components of the type 'draw'. All drawing components must support these request codes:

```

CONST
    kDrawerSetUpSelect      = 0;  {set up drawing region}
    kDrawerDrawSelect      = 1;  {draw the object}
    kDrawerEraseSelect     = 2;  {erase the object}
    kDrawerClickSelect     = 3;  {determine if cursor is }
                               { inside of the object}
    kDrawerMoveSelect     = 4;  {move the object}

```

The `params` field of the component parameters record is an array that contains the parameters specified by the application that called your component. You can directly extract the parameters from this array, or you can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` function to extract the parameters from this array and pass these parameters to a subroutine of your component (see page 6-63 and page 6-64 for more information about these functions).

Listing 6-5 shows the structure of a drawing component—a simple component that draws an object on the screen. The component subtype of a drawing component indicates the type of object the component draws. This particular drawing component is of the subtype 'oval'; it draws oval objects.

Whenever an application calls your component, the Component Manager calls your component's main entry point (for example, the `OvalDrawer` function). This entry point must be the first function in the component's code segment.

As previously described, the Component Manager passes two parameters to your component: a component parameters record and a parameter of type `Handle`. The parameters specified by the calling application are contained in the component parameters record. Your component can access the parameters directly from this record. Alternatively, as shown in Listing 6-5, you can use Component Manager routines to extract the parameters from this array and invoke a subroutine of your component. By taking advantage of these routines, you can simplify the structure of your component code.

Component Manager

The `OvalDrawer` function first examines the value of the `what` field of the component parameters record. The `what` field contains the request code. The `OvalDrawer` function performs the action specified by the request code. The `OvalDrawer` function uses a number of subroutines to carry out the desired action. It uses the Component Manager routines `CallComponentFunction` and `CallComponentFunctionWithStorage` to extract the parameters from the component parameters record and to call the specified component's subroutine with these parameters.

For example, when the drawing component receives the request code `kComponentOpenSelect`, it calls the function `CallComponentFunction`. It passes the component parameters record and a pointer to the component's `OvalOpen` subroutine as parameters to `CallComponentFunction`. This function extracts the parameters and calls the `OvalOpen` function. The `OvalOpen` function allocates memory for this instance of the component. Your component can allocate memory to hold global data when it receives an open request. To do this, allocate the memory and then call the `SetComponentInstanceStorage` function. This function associates the allocated memory with the current instance of your component. The next time this instance of your component is called, the Component Manager passes a handle to your previously allocated memory in the `storage` parameter. For additional information on handling the open request, see "Responding to the Open Request" on page 6-19.

When the drawing component receives the drawing setup request (indicated by the `kDrawerSetupSelect` constant), it calls the Component Manager function `CallComponentFunctionWithStorage`. Like `CallComponentFunction`, this function extracts the parameters and calls the specified subroutine (`OvalSetup`). The `CallComponentFunctionWithStorage` function also passes as a parameter to the subroutine a handle to the memory associated with this instance of the component. The `OvalSetup` subroutine can use this memory as needed. For additional information on handling the drawing setup request, see "Responding to Component-Specific Requests" on page 6-26.

Listing 6-5 A drawing component for ovals

```

UNIT Ovals;
INTERFACE
{include a USES statement if required}

FUNCTION OvalDrawer (params: ComponentParameters;
                    storage: Handle): ComponentResult;

IMPLEMENTATION

CONST
    kOvalDrawerVersion      = 0;  {version number of this component}

    kDrawerSetUpSelect      = 0;  {set up drawing region}

```

CHAPTER 6

Component Manager

```
kDrawerDrawSelect      = 1;  {draw the object}
kDrawerEraseSelect     = 2;  {erase the object}
kDrawerClickSelect     = 3;  {determine if cursor is }
                          { inside of the object}
kDrawerMoveSelect      = 4;  {move the object}

TYPE
  GlobalsRecord =
    RECORD
      bounds:          Rect;
      boundsRgn:       RgnHandle;
      self:             ComponentInstance;
    END;
  GlobalsPtr      = ^GlobalsRecord;
  GlobalsHandle   = ^GlobalsPtr;

{any subroutines used by the component go here}

FUNCTION OvalDrawer (params: ComponentParameters;
                    storage: Handle): ComponentResult;

BEGIN
  {perform action corresponding to request code}
  IF params.what < 0 THEN    {handle the required request codes}
    CASE (params.what) OF
      kComponentOpenSelect:
        OvalDrawer := CallComponentFunction(params,
                                             ComponentRoutine(@OvalOpen));

      kComponentCloseSelect:
        OvalDrawer := CallComponentFunctionWithStorage(storage, params,
                                                       ComponentRoutine(@OvalClose));

      kComponentCanDoSelect:
        OvalDrawer := CallComponentFunction(params,
                                             ComponentRoutine(@OvalCanDo));

      kComponentVersionSelect:
        OvalDrawer := kOvalDrawerVersion;
    OTHERWISE
      OvalDrawer := badComponentSelector;
    END {of CASE}
  ELSE
    {handle component-specific request codes}
    CASE (params.what) OF
      kDrawerSetupSelect:
        OvalDrawer := CallComponentFunctionWithStorage
                      (storage, params,
                       ComponentRoutine(@OvalSetup));
```

Component Manager

```

kDrawerDrawSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalDraw));
kDrawerEraseSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalErase));
kDrawerClickSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalClick));
kDrawerMoveSelect:
    OvalDrawer := CallComponentFunctionWithStorage
                    (storage, params,
                    ComponentRoutine(@OvalMoveTo));
OTHERWISE
    OvalDrawer := badComponentSelector;
END; {of CASE}
END; {of OvalDrawer}

END.

```

The next section describes how your component should respond to the required request codes. Following sections provide more information on

- n defining your component's interfaces
- n registering your component
- n how to store your component in a component resource file

Handling Requests for Service

Whenever an application requests services from your component, the Component Manager calls your component and passes two parameters: the application's parameters in a component parameters record and a handle to the memory associated with the current connection. The component parameters record also contains information identifying the nature of the request.

There are two classes of requests: requests that are defined by the Component Manager and requests that are defined by your component. The Component Manager defines seven request codes: open, close, can do, version, register, unregister, and target. All components must support open, close, can do, and version requests. The register, unregister, and target requests are optional. Apple reserves all negative request codes for definition by the Component Manager. You are free to assign request codes greater than or equal to 0 to the functions supported by a component whose interface you have

Component Manager

defined. (However, request codes between 0 and 256 are reserved for definition by components of a given type and a given type-subtype. Request codes greater than 256 are available for requests that are unique to your component.)

You can refer to the standard request codes with these constants.

```

CONST kComponentOpenSelect      = -1; {open request}
      kComponentCloseSelect     = -2; {close request}
      kComponentCanDoSelect     = -3; {can do request}
      kComponentVersionSelect   = -4; {version request}
      kComponentRegisterSelect  = -5; {register request}
      kComponentTargetSelect    = -6; {target request}
      kComponentUnregisterSelect = -7; {unregister request}

```

The following sections discuss what your component must do when it receives these Component Manager requests.

Responding to the Open Request

The Component Manager issues an open request to your component whenever an application or any other client tries to open a connection to your component by calling the `OpenComponent` (or `OpenDefaultComponent`) function. The open request allows your component to establish the environment to support the new connection. Your component must support this request.

Your component should perform the necessary processing to establish the new connection. At a minimum, you must allocate the memory for any global data for the connection. Be sure to allocate this memory in the current heap zone, not in the system heap. You should call the `SetComponentInstanceStorage` procedure to inform the Component Manager that you have allocated memory. The Component Manager stores a handle to the memory and provides that handle to your component as a parameter in subsequent requests.

You may also want to open and read data from your component's resource file—if you do so, use the `OpenComponentResFile` function to open the file and be sure to close the resource file before returning.

If your component uses the services of other components, open connections to them when you receive the open request.

Once you have successfully set up the connection, set your component's function result to 0 and return to the Component Manager.

You can also refuse the connection. If you cannot successfully establish the environment for a connection (for example, there is insufficient memory to support the connection, or required hardware is unavailable), you can refuse the connection by setting the component's function result to a nonzero value. You can also use the open request as an opportunity to restrict the number of connections your component can support.

If your application is registered globally, you should also set the A5 world for your component in response to the open request. You can do this using the

Component Manager

SetComponentInstanceA5 procedure. (See page 6-68 for information on this procedure.)

The Component Manager sets these fields in the component parameters record that it provides to your component on an open request:

Field descriptions

what	The Component Manager sets this field to kComponentOpenSelect.
params	The first entry in this array contains the component instance that identifies the new connection.

Listing 6-6 shows the subroutine that handles the open request for the drawing component. Note that your component can directly access the parameters from the component parameters record, or use subroutines and the CallComponentFunction and CallComponentFunctionWithStorage functions to extract the parameters for you (see Listing 6-5 on page 6-16). The code in this chapter takes the second approach.

The OvalOpen function allocates memory to hold global data for this instance of the component. It calls the SetComponentInstanceStorage function so that the Component Manager can associate the allocated memory with this instance of the component. The Component Manager passes a handle to this memory in subsequent calls to this instance of the component.

Listing 6-6 Responding to an open request

```

FUNCTION OvalOpen (self: ComponentInstance): ComponentResult;
VAR
    myGlobals: GlobalsHandle;
BEGIN
    {allocate storage}
    myGlobals :=
        GlobalsHandle(NewHandleClear(sizeof(GlobalsRecord)));
    IF myGlobals = NIL THEN
        OvalOpen := MemError
    ELSE
        BEGIN
            myGlobals^.self := self;
            myGlobals^.boundsRgn := NewRgn;
            SetComponentInstanceStorage(myGlobals^.self,
                Handle(myGlobals));
            {if your component is registered globally, set }
            { its A5 world before returning}
            OvalOpen := noErr;
        END;
    END;
END;

```

Responding to the Close Request

The Component Manager issues a close request to your component when a client application closes its connection to your component by calling the `CloseComponent` function. Your component should dispose of the memory associated with the connection. Your component must support this request. Your component should also close any files or connections to other components that it no longer needs.

The Component Manager sets these fields in the component parameters record that it provides to your component on a close request:

Field descriptions

<code>what</code>	The Component Manager sets this field to <code>kComponentCloseSelect</code> .
<code>params</code>	The first entry in this array contains the component instance that identifies the open connection.

Listing 6-7 shows the subroutine that handles the close request for the drawing component (as defined in Listing 6-5 on page 6-16). The `OvalClose` function closes the open connection. The drawing component uses the `CallComponentFunctionWithStorage` function to call the `OvalClose` function (see Listing 6-5). Because of this, in addition to the parameters specified in the component parameters record, the Component Manager also passes to the `OvalClose` function a handle to the memory associated with the component instance.

Listing 6-7 Responding to a close request

```
FUNCTION OvalClose (globals: GlobalsHandle;
                  self: ComponentInstance): ComponentResult;
BEGIN
  IF globals <> NIL THEN
    BEGIN
      DisposeRgn(globals^^.boundsRgn);
      DisposeHandle(Handle(globals));
    END;
    OvalClose := noErr;
  END;
```

IMPORTANT

When responding to a close request, you should always test the handle passed to your component against `NIL` because it is possible for your close request to be called with a `NIL` handle in the `storage` parameter. For example, you can receive a `NIL` handle if your component returns a nonzero function result in response to an open request. s

Responding to the Can Do Request

The Component Manager issues a can do request to your component when an application calls the `ComponentFunctionImplemented` function to determine whether your component supports a given request code. Your component must support this request.

Set your component's function result to 1 if you support the request code; otherwise, set your function result to 0.

The Component Manager sets these fields in the component parameters record that it provides to your component on a can do request:

Field descriptions

what	The Component Manager sets this field to <code>kComponentCanDoSelect</code> .
params	The first entry in this array contains the request code as an integer value.

Listing 6-8 shows the subroutine that handles the can do request for the drawing component (as defined in Listing 6-5 on page 6-16). The `OvalCanDo` function examines the specified request code and compares it with the request codes that it supports. It returns a function result of 1 if it supports the request code; otherwise, it returns 0.

Listing 6-8 Responding to the can do request

```
FUNCTION OvalCanDo (selector: Integer): ComponentResult;
BEGIN
    IF ((selector >= kComponentVersionSelect) AND
        (selector <= kDrawerMoveSelect)) THEN
        OvalCanDo := 1           {valid request}
    ELSE
        OvalCanDo := 0;        {invalid request}
END;
```

Responding to the Version Request

The Component Manager issues a version request to your component when an application calls the `GetComponentVersion` function to retrieve your component's version number. Your component must support this request.

In response to a version request, your component should return its version number as its function result. Use the high-order 16 bits to represent the major version and the low-order 16 bits to represent the minor version. The major version should represent the component specification level; the minor version should represent your implementation's version number.

Component Manager

If the Component Manager supports automatic version control (a feature available in version 3 and above of the manager), it automatically resolves conflicts between different versions of the same component. For more information on this feature, see the next section, “Responding to the Register Request.”

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on a version request:

Field description

<code>what</code>	The Component Manager sets this field to <code>kComponentVersionSelect</code> .
-------------------	---

Listing 6-5 on page 6-16 shows how the drawing component handles the version request. It simply returns its version number as its function result.

Responding to the Register Request

The Component Manager may issue a register request when your component is registered. This request gives your component an opportunity to determine whether it can operate in the current environment. For example, your component might use the register request to verify that a specific piece of hardware is available on the computer. This is an optional request—your component is not required to support it.

The Component Manager issues this request only if you have set the `cmpWantsRegisterMessage` flag to 1 in the `componentFlags` field of your component’s component description record (see “Data Structures for Components” beginning on page 6-52 for more information about the component description record).

Your component should not normally allocate memory in response to the register request. The register request is provided so that your application can determine whether it should be registered and made available to any clients. Once a client attempts to connect to your component, your component receives an open request, at which time it can allocate any required memory. Because your component might not be opened during a particular session, following this guideline allows other applications to make use of memory that isn’t currently needed by your component.

If you want the Component Manager to provide automatic version control (a feature available in version 3 and above of the manager), your component can specify the `componentDoAutoVersion` flag in the optional extension to the component resource. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Set your function result to `TRUE` to indicate that you do not want your component to be registered; otherwise, set the function result to `FALSE`.

Component Manager

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on a register request:

Field description

<code>what</code>	The Component Manager sets this field to <code>kComponentRegisterSelect</code> .
-------------------	--

If you request that your component receive a register request, the Component Manager actually sends your component a series of three requests: an open request, then the register request, followed by a close request.

For more information about the process the Component Manager uses to register components, see “Registering a Component” on page 6-30.

Responding to the Unregister Request

The unregister request is supported only in version 3 and above of the Component Manager. If your component specifies the `componentWantsUnregister` flag in the `componentRegisterFlags` field of the optional extension to the component resource, the Component Manager may issue an unregister request when your component is unregistered. This request gives your component an opportunity to perform any clean-up operations, such as resetting the hardware. This is an optional request—your component is not required to support it.

Return any error information as your component’s function result.

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on an unregister request:

Field description

<code>what</code>	The Component Manager sets this field to <code>kComponentUnregisterSelect</code> .
-------------------	--

If you have specified that your component should not receive a register request, then your component does not receive an unregister request if it has not been opened. However, if a client opens and closes your component, and then later the Component Manager unregisters your component, the Component Manager does send your component an unregister request (in a series of three requests: open, unregister, close).

If you have specified that your component should receive a register request, when your component is registered the Component Manager sends your component a series of three requests: an open request, then the register request, followed by a close request. In this situation, even if your component is not opened by a client, the Component Manager sends your component an unregister request when it unregisters your component.

For more information about the `componentWantsUnregister` flag, see “Resources” beginning on page 6-80.

Responding to the Target Request

The Component Manager issues a target request to inform an instance of your component that it has been targeted by another component. The component that targets another component instance may also choose to first capture the component, but it is not necessary to do so. Thus, a component can choose to

- n capture a component and target an instance of it
- n capture a component without targeting any instance of it
- n target a component instance without capturing the component

To first capture another component, the capturing component calls the `CaptureComponent` function. When a component is captured, the Component Manager removes it from the list of available components. This makes the captured component available only to the capturing component and to any clients currently connected to it. Typically, a component captures another component when it wants to override one or more functions of the other component.

After calling the `CaptureComponent` function, the capturing component can choose to target a particular instance of the component. However, a component can capture another component without targeting it.

A component uses the `ComponentSetTarget` function to send a target request to a specific component instance. After receiving a target request, whenever the targeted component instance would call itself (that is, call any of its defined functions), instead it should always call the component that targeted it.

For example, a component called `NewMath` might first capture a component called `OldMath`. `NewMath` does this by using `FindNextComponent` to get a component identifier for `OldMath`. `NewMath` then calls `CaptureComponent` to remove `OldMath` from the list of available components. At this point, no other clients can access `OldMath`, except for those clients previously connected to it.

`NewMath` might then call `ComponentSetTarget` to target a particular component instance of `OldMath`. The `ComponentSetTarget` function sends a target request to the specified component instance. When `OldMath` receives a target request, it saves the component instance of the component that targeted it. When `OldMath` receives a request, it processes it as usual. However, whenever `OldMath` calls one of its defined functions: in its defined API, it calls `NewMath` instead. (Suppose `OldMath` provides request codes for these functions: `DoMultiply`, `DoAdd`, `DoDivide`, and `DoSubtract`. If `OldMath`'s `DoMultiply` function calls its own `DoAdd` function, then `OldMath` calls `NewMath` to perform the addition.)

The target request is an optional request—your component is not required to support it.

Component Manager

The Component Manager sets these fields in the component parameters record that it provides to your component on a target request:

Field descriptions

what	The Component Manager sets this field to <code>kComponentTargetSelect</code> .
params	The first entry in this array contains the component instance that identifies the component issuing the target request.

Responding to Component-Specific Requests

When your component receives a component-specific request, it should handle the request as appropriate. For example, the drawing component responds to five component-specific requests: setup, draw, erase, click, and move to. See Listing 6-5 on page 6-16 for the code that defines the drawing component's entry point. The drawing component uses `CallComponentFunctionWithStorage` to extract the parameters and call the appropriate subroutine.

Listing 6-9 shows the drawing component's `OvalSetup` function. This function sets up the data structures that must be in place before drawing the oval.

Listing 6-9 Responding to the setup request

```

FUNCTION OvalSetup (globals: GlobalsHandle;
                   boundsRect: Rect): ComponentResult;
VAR
    ignoreErr: ComponentResult;
BEGIN
    globals^^.bounds := boundsRect;
    OpenRgn;
    ignoreErr := OvalDraw(globals);
    CloseRgn(globals^^.boundsRgn);
    OvalSetup := noErr;
END;

```

Listing 6-10 shows the drawing component's `OvalDraw` function. This function draws an oval in the previously allocated region.

Listing 6-10 Responding to the draw request

```

FUNCTION OvalDraw (globals: GlobalsHandle): ComponentResult;
BEGIN
    FrameOval(globals^^.bounds);
    OvalDraw := noErr;
END;

```

Listing 6-11 shows the drawing component's `OvalErase` function. This function erases an oval.

Listing 6-11 Responding to the erase request

```

FUNCTION OvalErase (globals: GlobalsHandle): ComponentResult;
BEGIN
    EraseOval(globals^^.bounds);
    OvalErase := noErr;
END;

```

Listing 6-12 shows the drawing component's `OvalClick` function. This function determines whether the given point is within the oval. If so, the function returns 1; otherwise, it returns 0. Because the `OvalClick` function returns information other than error information as its function result, `OvalClick` sets any error information using `SetComponentInstanceError`.

Listing 6-12 Responding to the click request

```

FUNCTION OvalClick (globals: GlobalsHandle; p: Point)
                    : ComponentResult;
BEGIN
    IF PtInRgn(p, globals^^.boundsRgn) THEN
        OvalClick := 1
    ELSE
        OvalClick := 0;
    SetComponentInstanceError(globals^^.self, noErr);
END;

```

Component Manager

Listing 6-13 shows the drawing component's `OvalMoveTo` function. This function moves the oval's coordinates to the specified location. Note that this function does not erase or draw the oval; the calling application is responsible for issuing the appropriate requests. For example, the calling application can issue requests to draw, erase, move to, and draw—to draw the oval in one location, then erase the oval, move it to a new location, and finally draw the oval in its new location.

Listing 6-13 Responding to the move to request

```
FUNCTION OvalMoveTo (globals: GlobalsHandle; x, y: Integer)
    : ComponentResult;
VAR
    r: Rect;
BEGIN
    r := globals^^.bounds;
    x := x - (r.right + r.left) DIV 2;
    y := y - (r.bottom + r.top) DIV 2;
    OffsetRect(globals^^.bounds, x, y);
    OffsetRgn(globals^^.boundsRgn, x, y);
    OvalMoveTo := noErr;
END;
```

Reporting an Error Code

The Component Manager maintains error state information for all currently active connections. In general, your component returns error information in its function result; a nonzero function result indicates an error occurred, and a function result of 0 indicates the request was successful. However, some requests require that your component return other information as its function result. In these cases, your component can use the `SetComponentInstanceError` procedure to report its latest error state to the Component Manager. You can also use this procedure at any time during your component's execution to report an error.

Defining a Component's Interfaces

You define the interfaces supported by your component by declaring a set of functions for use by applications. These function declarations specify the parameters that must be provided for each request. The following code illustrates the general form of these function declarations, using the setup request defined for the sample drawing component as an example:

```
FUNCTION DrawerSetup (myInstance: ComponentInstance;
    VAR r: Rect): ComponentResult;
```

Component Manager

This example declares a function that supports the setup request. The first parameter to any component function must be a parameter that accepts a component instance. The Component Manager uses this value to correctly route the request. The calling application must supply a valid component instance when it calls your component. The second and following parameters are those required by your component function. For example, the `DrawerSetup` function takes one additional parameter, a rectangle. Finally, all component functions must return a function result of type `ComponentResult` (a long integer).

These function declarations must also include inline code. This code identifies the request code assigned to the function, specifies the number of bytes of parameter data accepted by the function, and executes a trap to the Component Manager. To continue with the Pascal example used earlier, the inline code for the `DrawerSetup` function is

```
INLINE $2F3C, $0004, $0000, $7000, $A82A;
```

The first element of this code, `$2F3C`, is the opcode for a move instruction that loads the contents of the next two elements onto the stack. The Component Manager uses these values when it invokes your component.

The second element, `$0004`, defines an integer value that specifies the number of bytes of parameter data required by the function, not including the component instance parameter. In this case, the size of a pointer to the rectangle is specified: 4 bytes.

Note

Note that Pascal calling conventions require that Boolean and 1-byte parameters are passed as 16-bit integer values. \cup

The third element, `$0000`, specifies the request code for this function as an integer value. Each function supported by your component must have a unique request code. Your component uses this request code to identify the application's request. You may define only request code values greater than or equal to 0; negative values are reserved for definition by the Component Manager. Recall from the oval drawing component that the request code for the setup request, `kDrawerSetUpSelect`, has a value of 0.

The fourth element, `$7000`, is the opcode for an instruction that sets the D0 register to 0, which tells the Component Manager to call your component rather than to field the request itself.

The fifth element, `$A82A`, is the opcode for an instruction that executes a trap to the Component Manager.

If you are declaring functions for use by Pascal-language applications, your declarations should take the following form:

```
FUNCTION DrawerSetup (myInstance: ComponentInstance;
                     VAR r: Rect): ComponentResult;
INLINE $2F3C, $0004, $0000, $7000, $A82A;
```

Component Manager

If you are declaring functions for use by C-language applications, your declarations can take the following form:

```
pascal ComponentResult DrawerSetup
    (ComponentInstance myInstance, Rect *r) =
        {0x2F3C, 0x4, 0x0, 0x7000, 0xA82A};
```

Alternatively, you can define the following statement to replace the inline code:

```
#define ComponentCall (callNum, paramSize)
    {0x2F3C, paramSize, callNum, 0x7000, 0xA82A}
```

Using this statement results in the following declaration format:

```
pascal ComponentResult DrawerSetup
    (ComponentInstance myInstance, Rect *r) =
        ComponentCall (kDrawerSetUpSelect, 4);
```

When a client application calls your function, the system executes the inline code, which invokes the Component Manager. The Component Manager then formats a component parameters record, locates the storage for the current connection, and invokes your component. The Component Manager provides the component parameters record and a handle to the storage of the current connection to your component as function parameters.

Managing Components

This section discusses the Component Manager routines that help you manage your component. It describes how to register your component and how to allow applications to connect to your component.

Registering a Component

Applications must use the services of the Component Manager to find components that meet their needs. Before an application can find a component, however, that component must be registered with the Component Manager. When you register your component, the Component Manager adds the component to its list of available components.

There are two mechanisms for registering a component with the Component Manager. First, during startup processing, the Component Manager searches the Extensions folder (and all of the folders within the Extensions folder) for files of type 'thng'. If the file contains all the information needed for registration (see "Creating a Component Resource" on page 6-32 for more information on creating a component file), the Component Manager automatically registers the component stored in the file. Components registered in this manner are registered globally; that is, the component is made available to all applications and other clients.

Second, your application (or another application) can register your component. When you register your component in this manner, you can specify whether the component should be made available to all applications (global registration) or only to your application (local registration). Your application can register a component that is in memory or that is stored in a resource. You use the `RegisterComponent` function to register a component that is in memory. You use the `RegisterComponentResource` function to register a component that is stored in a component resource. See “The Component Resource” on page 6-80 for a description of the format and content of component resources. The code in Listing 6-14 demonstrates how an application can use the `RegisterComponent` function to register a component that is in memory.

Listing 6-14 Registering a component

```
VAR
    cd:      ComponentDescription;
    draw:    Component;

    WITH cd DO
    BEGIN
        {initialize the component description record}
        componentType := 'draw';
        componentSubtype := 'oval';
        componentManufacturer := 'appl';
        componentFlags := 0;
        componentFlagsMask := 0;
    END;
    {register the component}
    draw := RegisterComponent(cd, ComponentRoutine(@OvalDrawer),
                             0, NIL, NIL, NIL);
```

The code in Listing 6-14 specifies six parameters to the `RegisterComponent` function. The first three are a component description record, a pointer to the component’s entry point, and a value of 0 to indicate that this component should be made available only to this application. A component that is registered locally is visible only within the A5 world of the registering program. The last three parameters are specified as `NIL` to indicate that the component doesn’t have a name, an information string, or an icon. See page 6-57 for more information on the `RegisterComponent` function.

When a component is registered and the `cmpWantsRegisterMessage` bit is not set in the `componentFlags` field of the component description record, the Component Manager adds the component to its list of registered components. Whenever a client requests access to or information about a component (for example, by using `OpenDefaultComponent`, `FindNextComponent`, or `GetComponentInfo`), the Component Manager searches its list of registered components.

Component Manager

If a component's `cmpWantsRegisterMessage` bit is set, the Component Manager does not automatically add your component to its list of registered components. Instead, it sends your component a series of three requests: open, register, and close. If your component returns a nonzero value as its function result in response to the register request, your component is not added to the Component Manager's list of registered components. Thus, clients are not able to connect to or get information about your component. You might choose to set the `cmpWantsRegisterMessage` bit if, for example, your application requires specific hardware.

Alternatively, you can let your component be automatically registered. Your component can then check for any specific hardware requirements upon receiving an open request. This lets clients attempt to connect to your component and also lets them get information about your component. However, in most cases, if your component requires specific hardware to operate, you should set the `cmpWantsRegisterMessage` bit and respond to the register request appropriately.

If your component controls a hardware resource, you should register your component once for each hardware resource that is available (rather than registering once and allowing multiple instances of your component). This allows clients to easily determine how many hardware resources are available by using the `FindNextComponent` function. If you register a component multiple times, be sure that you specify a unique name for each registration.

If the feature is available, you can request that the Component Manager provide automatic version control for your component (this feature is available only in version 3 and above of the manager). To request automatic version control, specify the `componentDoAutoVersion` flag in the optional extension to the component resource. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Creating a Component Resource

You can create a component resource (a resource of type `'thng'`) in a component file. A component file is a file whose resource fork contains a component resource and other required resources for the component. If you store your component in a component file, either you can allow applications to use the `RegisterComponentResource` function to register your component as needed, or you can automatically register your component at startup by storing your component file in the Extensions folder.

Component Manager

A component file consists of

- n a component description record that specifies the characteristics of your component (its type, subtype, manufacturer, and control flags)
- n the resource type and resource ID of your component's code resource
- n the resource type and resource ID of your component's name string
- n the resource type and resource ID of your component's information string
- n the resource type and resource ID of your component's icon
- n optional information about the component (its version number, additional flags, and resource ID of the component's icon family)
- n the actual resources for your component's code, name, information string, and icon

Listing 6-15 shows, in Rez format, a component resource that defines an oval drawing component. This drawing component does not specify optional information (see Figure 6-5 on page 6-85 for the contents of the optional extension to the component resource). For compatibility with early versions of the Component Manager, component resources should be locked.

Listing 6-15 Rez input for a component resource

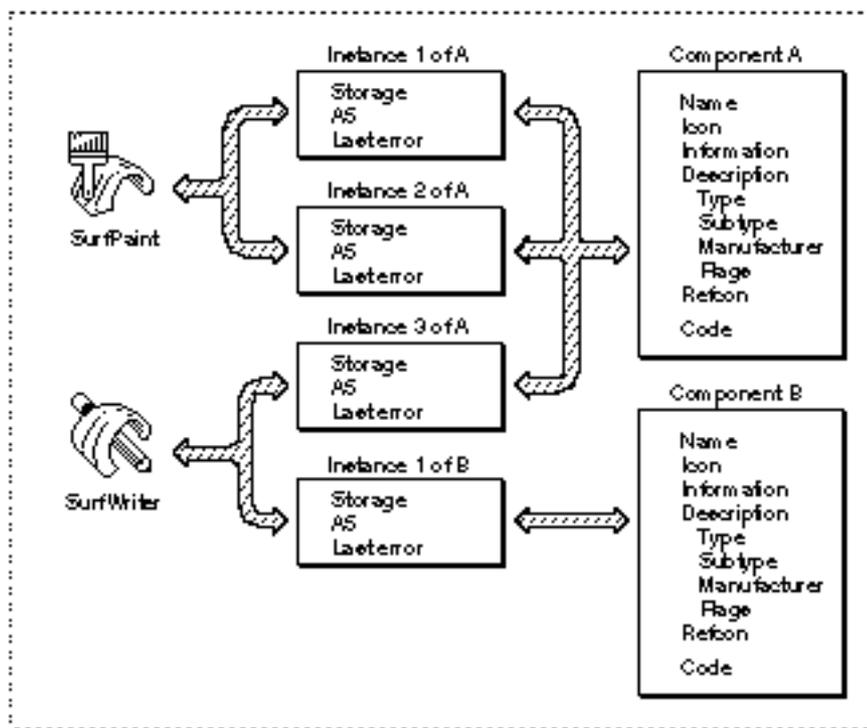
```
resource 'thng' (128, locked) {
    'draw',          /*component type*/
    'oval',          /*component subtype*/
    'appl',          /*component manufacturer*/
    $00000000,       /*component flags: 0*/
    $00000000,       /*reserved (component flags mask): 0*/
    'CODE',          /*component code resource type*/
    128,             /*component code resource ID*/
    'STR ',          /*component name resource type*/
    128,             /*component name resource ID*/
    'STR ',          /*component info resource type*/
    129,             /*component info resource ID*/
    'ICON',          /*component icon resource type*/
    128              /*component icon resource ID*/
    /*optional information (if any) goes here*/
};
```

The component resource, and the resources that define the component's code, name, information string, and icon, must be in the same file. A component file must have the file type 'thng' and reside in the Extensions folder in order to be automatically registered by the Component Manager at startup.

Establishing and Managing Connections

Your component may support one or more connections at a time. In addition, a single application may have open connections with two or more different components at the same time. In fact, a single application can use more than one connection to a single component. Figure 6-2 shows two applications and two components: the first application, SurfPaint, uses two connections to component A; the second application, SurfWriter, uses one connection to component A and one to component B.

Figure 6-2 Supporting multiple component connections



A component can allocate separate storage for each open connection. A component can also set the A5 world for a specific component instance and can maintain separate error information for each instance. A component can also use a reference constant value to maintain global data for the component.

When an application requests that the Component Manager open a connection to your component, the Component Manager issues an open request to your component. At this time, your component should allocate any memory it needs in order to maintain a connection for the requesting application. Be sure to allocate this memory in the current heap zone rather than in the system heap. As described in “Responding to the Open Request” on page 6-19, you can use the `SetComponentInstanceStorage` procedure to associate the allocated memory with the component instance. Whenever the application requests services from your component, the Component Manager supplies

you with the handle to this memory. You can also use the open request as an opportunity to restrict the number of connections your component can support.

To allocate global data for your component, you can maintain a reference constant for use by your component. The Component Manager provides two routines, `SetComponentRefcon` and `GetComponentRefcon`, that allow you to work with your component's reference constant. Note that your component has one reference constant, regardless of the number of connections maintained by your component.

If your component uses its reference constant and is registered globally, be aware that in certain situations the Component Manager may clone your component. This situation occurs only when the Component Manager opens a component that is registered globally and there's no available space in the system heap. In this case, the Component Manager clones your component, creating a new registration of the component in the caller's heap, and returns to the caller the component identifier of the cloned component, not the component identifier of the original registration. The reference constant of the original component is not preserved in the cloned component. Thus you need to take extra steps to set the reference constant of the cloned component to the same value as that of the original component.

To determine whether your component has been cloned, you can examine your component's A5 world using the `GetComponentInstanceA5` function. If the returned value of the A5 world is nonzero, your component is cloned (only components registered globally can be cloned; if your component is registered locally it has a valid, nonzero A5 world and you don't need to check whether it's been cloned). If you determine that your component is cloned, you can retrieve the original reference constant by using the `FindNextComponent` function to iterate through all registrations of your component. You should compare the component identifier of the cloned component with the component identifier returned by `FindNextComponent`. Once you find a component with the same component description but a different component identifier, you've found the original component. You can then use `GetComponentRefcon` to get the reference constant of the original component and then use `SetComponentRefcon` to set the reference constant of the cloned component appropriately. This technique works if a component registers itself only once or registers itself multiple times but with a unique name for each registration. This technique does not work if a component registers itself multiple times using the same name.

When responding to a request from an application, your component can invoke the services of other components. The Component Manager provides two techniques for calling other components. First, your component can call another component using the standard mechanisms also used by applications. The Component Manager then passes the requests to the appropriate component, and your component receives the results of those requests.

Second, your component can redirect a request to another component. For example, you might want to create two similar components that provide different levels of service to applications. Rather than completely implementing both components, you could design one to rely on the capabilities of the other. Use the `DelegateComponentCall` function to pass a request on to another component.

Listing 6-16 shows an example of delegating a request to another component. The component in this example is a drawing component that draws rectangles. The `RectangleDrawer` component handles open, close, and setup requests. It delegates all other requests to another component. When the `RectangleDrawer` component receives an open request, it opens the component to which it will later delegate requests, and stores in its allocated storage the delegated component's component instance. It then specifies this value when it calls the `DelegateComponentCall` function.

Listing 6-16 Delegating a request to another component

```

FUNCTION RectangleDrawer(params: ComponentParameters;
                        storage: Handle): ComponentResult;
VAR
    theRtn: ComponentRoutine;
    safe: Boolean;
BEGIN
    safe := FALSE;
    CASE (params.what) OF
        kComponentOpenSelect:
            theRtn := ComponentRoutine(@RectangleOpen);
        kComponentCloseSelect:
            theRtn := ComponentRoutine(@RectangleClose);
        kDrawerSetupSelect:
            theRtn := ComponentRoutine(@RectangleSetup);
        OTHERWISE
            BEGIN
                safe := TRUE;
                IF (storage <> NIL) THEN
                    RectangleDrawer :=
                        DelegateComponentCall
                            (params,
                             ComponentInstance(StorageHdl(storage)^^.delegateInstance))
                ELSE
                    RectangleDrawer := badComponentSelector;
            END;
    END; {of CASE}
    IF NOT safe THEN
        RectangleDrawer :=
            CallComponentFunctionWithStorage(storage, params, theRtn);
    END;

```

Component Manager Reference

This section provides information about the data structures, routines, and resources defined by the Component Manager. This section is divided into the following topics:

- n “Data Structures for Applications” describes the data structures used by applications.
- n “Routines for Applications” discusses the Component Manager routines that are available to applications that use components.
- n “Data Structures for Components” describes the data structures used by components.
- n “Routines for Components” describes the Component Manager routines that are used by components.
- n “Application-Defined Routine” describes how to define a component function and supply the appropriate registration information.
- n “Resources” describes the format and content of component resources.

Assembly-Language Note

You can invoke Component Manager routines by using the trap `_ComponentDispatch` with the appropriate routine selector. The routine selectors are listed in “Assembly-Language Summary” beginning on page 6-98. u

Data Structures for Applications

This section describes the format and content of the data structures used by applications that use components.

Your application can use the component description record to find components that provide specific services or meet other selection criteria.

The Component Description Record

The component description record identifies the characteristics of a component, including the type of services offered by the component and its manufacturer.

Applications and components use component description records in different ways. An application that uses components specifies the selection criteria for finding a component in a component description record. A component uses the component description record to specify its registration information and capabilities. If you are developing a component, see page 6-52 for information on how a component uses the component description record.

The `ComponentDescription` data type defines the component description record.

Component Manager

```

TYPE ComponentDescription =
  RECORD
    componentType:      OSType;      {type}
    componentSubType:   OSType;      {subtype}
    componentManufacturer:
                                OSType;
    componentFlags:     LongInt;     {control flags}
    componentFlagsMask: LongInt;     {mask for control }
                                { flags}
  END;

```

Field descriptions`componentType`

A four-character code that identifies the type of component. All components of a particular type must support a common set of interface routines. For example, drawing components all have a component type of 'draw'.

Your application can use this field to search for components of a given type. You specify the component type in the `componentType` field of the component description record you supply to the `FindNextComponent` or `CountComponents` routine. A value of 0 operates as a wildcard.

`componentSubType`

A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of drawing components indicates the type of object the component draws. Drawing components that draw ovals have a subtype of 'oval'.

Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type ('draw') that also have a specific entry in the `componentSubType` field ('oval'). By specifying particular values for both fields in the component description record that you supply to the `FindNextComponent` or `CountComponents` routine, your application retrieves information about only those components that meet both of these search criteria. A value of 0 operates as a wildcard.

Component Manager

`componentManufacturer`

A four-character code that identifies the manufacturer of the component. This field allows for further differentiation between individual components. For example, components made by a specific manufacturer may support an extended feature set. Components provided by Apple use a manufacturer value of 'appl'.

Your application can use this field to find components from a certain manufacturer. Specify the appropriate manufacturer code in the `componentManufacturer` field of the component description record you supply to the `FindNextComponent` or `CountComponents` routine. A value of 0 operates as a wildcard.

`componentFlags`

A 32-bit field that provides additional information about a particular component.

The high-order 8 bits are defined by the Component Manager. You should usually set these bits to 0.

The low-order 24 bits are specific to each component type. These flags can be used to indicate the presence of features or capabilities in a given component.

Your application can use these flags to further narrow the search criteria applied by the `FindNextComponent` or `CountComponents` routine. If you use the `componentFlags` field in a component search, you use the `componentFlagsMask` field to indicate which flags are to be considered in the search.

`componentFlagsMask`

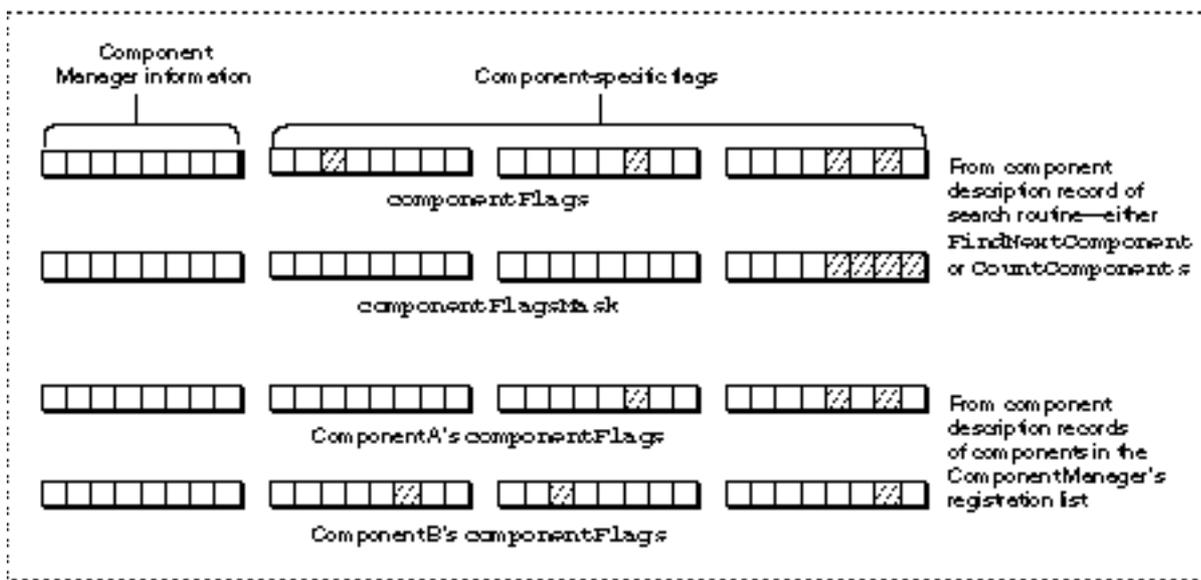
A 32-bit field that indicates which flags in the `componentFlags` field are relevant to a particular component search operation.

For each flag in the `componentFlags` field that is to be considered as a search criterion by the `FindNextComponent` or `CountComponents` routine, your application should set the corresponding bit in the `componentFlagsMask` field to 1. The Component Manager considers only these flags during the search. You specify the desired flag value (either 0 or 1) in the `componentFlags` field.

For example, to look for a component with a specific control flag that is set to 0, set the appropriate bit in the `ComponentFlags` field to 0 and the same bit in the `ComponentFlagsMask` field to 1. To look for a component with a specific control flag that is set to 1, set the bit in the `ComponentFlags` field to 1 and the same bit in the `ComponentFlagsMask` field to 1. To ignore a flag, set the bit in the `ComponentFlagsMask` field to 0.

Figure 6-3 shows how the various fields interact during a search. In the case depicted in the figure, the `componentFlagsMask` field of a component description record supplied to a search routine specifies that only the low-order four flags of the `componentFlags` field are to be examined during the search. The `componentFlags` fields in the component description records of components A and B have a number of flags set. However, in this example the mask specifies that the Component Manager examine only the low-order 4 bits, and therefore only component A meets the search criteria.

Figure 6-3 Interaction between the `componentFlags` and `componentFlagsMask` fields



Component Identifiers and Component Instances

In general, when using Component Manager routines, your application must specify the particular component using either a component identifier or component instance. The Component Manager identifies *each component* by a component identifier. The Component Manager identifies *each instance* of a component by a component instance. Thus, when your application searches for a component with a particular type and subtype using the `FindNextComponent` function, `FindNextComponent` returns a component identifier that identifies the component. Similarly, your application specifies a component identifier to the `GetComponentInfo` function to obtain information about a component.

When you open a connection to a component, the `OpenDefaultComponent` and `OpenComponent` functions return a component instance. The returned component instance identifies that specific instance of the component. If you open the same component again, the Component Manager returns a different component instance. So a

Component Manager

component has a single component identifier and can have multiple component instances. To use a component function, your application specifies a component instance.

Although conceptually component identifiers and component instances serve different purposes, Component Manager routines (with the exception of `DelegateComponentCall`) allow you to use component identifiers and component instances interchangeably. If you do this, you must always coerce the data type appropriately.

A component identifier is defined by the data type `Component`:

```
TYPE
    {component identifier}
    Component          = ^ComponentRecord;
    ComponentRecord   =
    RECORD
        data: ARRAY[0..0] OF LongInt;
    END;
```

A component instance is defined by the data type `ComponentInstance`:

```
TYPE
    {component instance}
    ComponentInstance = ^ComponentInstanceRecord;
    ComponentInstanceRecord =
    RECORD
        data: ARRAY[0..0] OF LongInt;
    END;
```

Routines for Applications

This section discusses the Component Manager routines that are used by applications. If you are developing an application that uses components, you should read this section. If you are developing an application that registers components, you should also read “Registering Components” beginning on page 6-57.

If you are developing a component, you should read this section and “Routines for Components” beginning on page 6-56.

This section describes the routines that allow your application to

- n search for components
- n gain access to and release components
- n get detailed information about specific components
- n get component error information

Note

Any of the routines discussed in this section that require a component identifier also accept a component instance. Similarly, you can supply a component identifier to any routine that requires a component instance (except for the `DelegateComponentCall` function). If you do this, you must always coerce the data type appropriately. \cup

Finding Components

The Component Manager provides routines that allow your application to search for components. Your application specifies the search criteria in a component description record. (See “Data Structures for Applications” beginning on page 6-37 for information about the component description record.) Based on the values you specify in fields of the component description record, the Component Manager attempts to find components that meet the needs of your application.

You can use the `CountComponents` function to determine the number of components that match a component description. Use the `FindNextComponent` function to find an individual component that matches a description.

You can use the `GetComponentListModSeed` function to determine whether the list of registered components has changed.

FindNextComponent

The `FindNextComponent` function returns the component identifier for the next registered component that meets the selection criteria specified by your application. You specify the selection criteria in a component description record.

Your application can use the component identifier returned by this function to get more information about the component or to open the component.

```
FUNCTION FindNextComponent (aComponent: Component;
                           looking: ComponentDescription)
                           : Component;
```

`aComponent`

The starting point for the search. Set this field to 0 to start the search at the beginning of the component list. If you are continuing a search, you can specify a component identifier previously returned by the `FindNextComponent` function. The function then searches the remaining components.

`looking`

A component description record. Your application specifies the criteria for the component search in the fields of this record.

The Component Manager ignores fields in the component description record that are set to 0. For example, if you set all the fields to 0, all components meet the search criteria. In this case, your application can

retrieve information about all of the components that are registered in the system by repeatedly calling `FindNextComponent` and `GetComponentInfo` until the search is complete. Similarly, if you set all fields to 0 except for the `componentManufacturer` field, the Component Manager searches all registered components for a component supplied by the manufacturer you specify. Note that the `FindNextComponent` function does not modify the contents of the component description record you supply. To retrieve detailed information about a component, you need to use the `GetComponentInfo` function to get the component description record for each returned component.

DESCRIPTION

The `FindNextComponent` function returns the component identifier of a component that meets the search criteria. `FindNextComponent` returns a function result of 0 when there are no more matching components.

SEE ALSO

Use the `GetComponentInfo` function, described on page 6-48, to retrieve more information about a component. To open a component, use the `OpenDefaultComponent` or `OpenComponent` function, described on page 6-45 and page 6-46, respectively. See page 6-37 for information on the component description record.

See Listing 6-1 on page 6-9 for an example of searching for a specific component.

CountComponents

Your application can use the `CountComponents` function to determine the number of registered components that meet your selection criteria. You specify the selection criteria in a component description record. The `CountComponents` function returns the number of components that meet those search criteria.

```
FUNCTION CountComponents (looking: ComponentDescription): LongInt;
```

`looking` A component description record. Your application specifies the criteria for the component search in the fields of this record.

The Component Manager ignores fields in the component description record that are set to 0. For example, if you set all the fields to 0, the Component Manager returns the number of components registered in the system. Similarly, if you set all fields to 0 except for the `componentManufacturer` field, the Component Manager returns the number of registered components supplied by the manufacturer you specify.

DESCRIPTION

The `CountComponents` function returns a long integer containing the number of components that meet the specified search criteria.

SEE ALSO

See page 6-37 for information on the component description record.

GetComponentListModSeed

The `GetComponentListModSeed` function allows you to determine if the list of registered components has changed. This function returns the value of the component registration seed number. By comparing this value to values previously returned by the this function, you can determine whether the list has changed. Your application may use this information to rebuild its internal component lists or to trigger other activity that is necessary whenever new components are available.

```
FUNCTION GetComponentListModSeed: LongInt;
```

DESCRIPTION

The `GetComponentListModSeed` function returns a long integer containing the component registration seed number. Each time the Component Manager registers or unregisters a component it generates a new, unique seed number.

Opening and Closing Components

The `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions allow your application to gain access to and release components. Your application must open a component before it can use the services provided by that component. Similarly, your application must close the component when it is finished using the component.

You can use the `OpenDefaultComponent` function to open a component of a specified component type and subtype. You do not have to supply a component description record or call the `FindNextComponent` function to use this function.

You use the `OpenComponent` function to gain access to a specified component. To use this function, your application must have previously obtained a component identifier for the desired component by using the `FindNextComponent` function. (If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.)

Once you are finished using a component, use the `CloseComponent` function to release the component.

OpenDefaultComponent

The `OpenDefaultComponent` function allows your application to gain access to the services provided by a component. Your application must open a component before it can call any component functions. You specify the component type and subtype values of the component to open. The Component Manager searches for a component that meets those criteria. If you want to exert more control over the selection process, you can use the `FindNextComponent` and `OpenComponent` functions.

```
FUNCTION OpenDefaultComponent (componentType: OSType;
                               componentSubType: OSType)
                               : ComponentInstance;
```

`componentType`

A four-character code that identifies the type of component. All components of a particular type support a common set of interface routines. Your application uses this field to search for components of a given type.

`componentSubType`

A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of an image compressor component indicates the compression algorithm employed by the compressor.

Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type ('draw') that also have a specific subtype ('oval'). Set this parameter to 0 to select a component with any subtype value.

DESCRIPTION

The `OpenDefaultComponent` function searches its list of registered components for a component that meets the search criteria. If it finds a component that matches the search criteria, `OpenDefaultComponent` opens a connection to the component and returns a component instance. The returned component instance identifies your application's connection to the component. You must supply this component instance whenever you call the functions provided by the component. When you close the component, you must also supply this component instance to the `CloseComponent` function.

If more than one component in the list of registered components meets the search criteria, `OpenDefaultComponent` opens the first one that it finds in its list.

If it cannot open the specified component, the `OpenDefaultComponent` function returns a function result of `NIL`.

SEE ALSO

For an example that opens a component using the `OpenDefaultComponent` function, see “Opening a Connection to a Default Component” beginning on page 6-7.

OpenComponent

The `OpenComponent` function allows your application to gain access to the services provided by a component. Your application must open a component before it can call any component functions. You specify the component with a component identifier that your application previously obtained from the `FindNextComponent` function.

Alternatively, you can use the `OpenDefaultComponent` function, as previously described, to open a component without calling the `FindNextComponent` function.

Note that your application may maintain several connections to a single component, or it may have connections to several components at the same time.

```
FUNCTION OpenComponent (aComponent: Component): ComponentInstance;
```

`aComponent`

A component identifier that specifies the component to open. Your application obtains this identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

DESCRIPTION

The `OpenComponent` function returns a component instance. The returned component instance identifies your application’s connection to the component. You must supply this component instance whenever you call the functions provided by the component. When you close the component, you must also supply this component instance to the `CloseComponent` function.

If it cannot open the specified component, the `OpenComponent` function returns a function result of `NIL`.

SEE ALSO

For examples of opening a specific component by using the `FindNextComponent` and `OpenComponent` functions, see Listing 6-1 on page 6-9 and Listing 6-2 on page 6-10, respectively. For a description of the `FindNextComponent` function, see page 6-42.

CloseComponent

The `CloseComponent` function terminates your application's access to the services provided by a component. Your application specifies the connection to be closed with the component instance returned by the `OpenComponent` or `OpenDefaultComponent` function.

```
FUNCTION CloseComponent
    (aComponentInstance: ComponentInstance): OSErr;
```

`aComponentInstance`

A component instance that specifies the connection to close. Your application obtains the component instance from the `OpenComponent` function or the `OpenDefaultComponent` function.

DESCRIPTION

The `CloseComponent` function closes only a single connection. If your application has several connections to a single component, you must call the `CloseComponent` function once for each connection.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier

SEE ALSO

For a description of the `OpenDefaultComponent` and `OpenComponent` functions, see page 6-45 and page 6-46, respectively.

Getting Information About Components

Your application can get the registration information for any component using the `GetComponentInfo` function. You can use the `GetComponentIconSuite` function to get a handle to the component's icon suite, if any.

In addition, for components to which your application already has a connection, your application can obtain the component's version number and also determine whether the component supports a particular request by using the `GetComponentVersion` and `ComponentFunctionImplemented` functions.

GetComponentInfo

The `GetComponentInfo` function returns all of the registration information for a component. Your application specifies the component with a component identifier returned by the `FindNextComponent` function. The `GetComponentInfo` function returns information about the component in a component description record. The `GetComponentInfo` function also returns the component's name, information string, and icon. (To get a handle to the component's icon suite, if it provides one, use the `GetComponentIconSuite` function.)

A component provides this registration information when it is registered with the Component Manager.

```
FUNCTION GetComponentInfo (aComponent: Component;
                          VAR cd: ComponentDescription;
                          componentName: Handle;
                          componentInfo: Handle;
                          componentIcon: Handle): OSErr;
```

`aComponent`

A component identifier that specifies the component for the operation. Your application obtains a component identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

You may supply a component instance rather than a component identifier to this function. (If you do so, you must coerce the data type appropriately.) Your application can obtain a component instance from the `OpenComponent` function or the `OpenDefaultComponent` function.

`cd`

A component description record. The `GetComponentInfo` function returns information about the specified component in a component description record.

`componentName`

An existing handle that is to receive the component's name. If the component does not have a name, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's name.

`componentInfo`

An existing handle that is to receive the component's information string. If the component does not have an information string, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's information string.

`componentIcon`

An existing handle that is to receive the component's icon. If the component does not have an icon, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's icon.

DESCRIPTION

The `GetComponentInfo` function returns information about the specified component in the `cd`, `componentName`, `componentInfo`, and `componentIcon` parameters.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier

SEE ALSO

For information on the component description record, see page 6-37. For information on the `FindNextComponent` function, see page 6-42. For information on registering components, see “Registering Components” beginning on page 6-57.

For an example of the use of the `GetComponentInfo` function, see Listing 6-3 on page 6-10.

GetComponentIconSuite

The `GetComponentIconSuite` function returns a handle to the component’s icon suite (if it provides one).

```
FUNCTION GetComponentIconSuite (aComponent: Component;
                                VAR iconSuite: Handle): OSErr;
```

`aComponent`

A component identifier that specifies the component for the operation. Your application obtains a component identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

`iconSuite` `GetComponentIconSuite` returns, in this parameter, a handle to the component’s icon suite, if any. If the component has not provided an icon suite, `GetComponentIconSuite` returns `NIL` in this parameter.

DESCRIPTION

The `GetComponentIconSuite` function returns a handle to the component’s icon suite. A component provides to the Component Manager the resource ID of its icon family in the optional extensions to the component resource. Your application is responsible for disposing of the returned icon suite handle.

SPECIAL CONSIDERATIONS

The `GetComponentIconSuite` function is available only in version 3 of the Component Manager.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier

SEE ALSO

For information about icon suites and icon families, see the chapter “Icon Utilities” in this book.

GetComponentVersion

The `GetComponentVersion` function returns a component’s version number.

```
FUNCTION GetComponentVersion (ci: ComponentInstance): LongInt;
```

`ci` The component instance from which you want to retrieve version information. Your application obtains the component instance from the `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

The `GetComponentVersion` function returns a long integer containing the version number of the component you specify. The high-order 16 bits represent the major version, and the low-order 16 bits represent the minor version. The major version specifies the component specification level; the minor version specifies a particular implementation’s version number.

ComponentFunctionImplemented

The `ComponentFunctionImplemented` function allows you to determine whether a component supports a specified request. Your application can use this function to determine a component’s capabilities.

```
FUNCTION ComponentFunctionImplemented (ci: ComponentInstance;
                                       ftnNumber: Integer)
                                       : LongInt;
```

Component Manager

<code>ci</code>	A component instance that specifies the connection for this operation. Your application obtains the component instance from the <code>OpenDefaultComponent</code> or <code>OpenComponent</code> function.
<code>ftnNumber</code>	A request code value. See <i>Inside Macintosh: QuickTime Components</i> for information about the request codes supported by the components supplied by Apple with QuickTime. For other components, see the documentation supplied with the component for request code values.

DESCRIPTION

The `ComponentFunctionImplemented` function returns a long integer indicating whether the component supports the specified request. You can interpret this long integer as if it were a Boolean value. If the returned value is `TRUE`, the component supports the specified request. If the returned value is `FALSE`, the component does not support the request.

Retrieving Component Errors

The Component Manager provides a routine that allows your application to retrieve the last error code that was generated by a component instance. Some component routines return error information as their function result. Other component routines set an error code that your application can retrieve using the `GetComponentInstanceError` function. Refer to the documentation supplied with the component for information on how that particular component handles errors.

GetComponentInstanceError

The `GetComponentInstanceError` function returns the last error generated by a specific connection to a component.

```
FUNCTION GetComponentInstanceError
    (aComponentInstance: ComponentInstance): OSErr;
```

`aComponentInstance`

A component instance that specifies the connection from which you want error information. Your application obtains the component instance from the `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

Once you have retrieved an error code, the Component Manager clears the error code for the connection. If you want to retain that error value, you should save it in your application's local storage.

RESULT CODES

noErr	0	No error
invalidComponentID	-3000	No component with this component identifier

Data Structures for Components

This section describes the format and content of the data structures used by components.

Components, and applications that register components, use the component description record to identify a component. A component resource incorporates the information in a component description record and also includes other information. If you are developing a component or an application that registers components, you must be familiar with both the component description record and component resource; see “Resources” beginning on page 6-80 for a description of the component resource.

The Component Manager passes information about a request to your component in a component parameters record.

The Component Description Record

The component description record identifies the characteristics of a component, including the type of services offered by the component and the manufacturer of the component.

Components use component description records to identify themselves to the Component Manager. If your component is stored in a component resource, the information in the component description record must be part of that resource (see the description of the component resource, on page 6-80). If you have developed an application that registers your component, that application must supply a component description record to the `RegisterComponent` function (see “Registering Components” on page 6-57 for information about registering components).

The `ComponentDescription` data type defines the component description record. Note that the valid values of fields in the component description record are determined by the component type specification. For example, all image compressor components must use the `componentSubType` field to specify the compression algorithm used by the compressor.

```

TYPE ComponentDescription =
    RECORD
        componentType:      OSType;      {type}
        componentSubType:   OSType;      {subtype}
        componentManufacturer:
                                OSType;
                                {manufacturer}
        componentFlags:     LongInt;     {control flags}
        componentFlagsMask: LongInt;     {reserved}
    END;

```

Component Manager

Field descriptions`componentType`

A four-character code that identifies the type of component. All components of a particular type must support a common set of interface routines. For example, drawing components all have a component type of 'draw'.

Your component must support all of the standard routines for the component type specified by this field. Type codes with all lowercase characters are reserved for definition by Apple. See *Inside Macintosh: QuickTime Components* for information about the QuickTime components supplied by Apple. You can define your own component type code as long as you register it with Apple's Component Registry Group.

`componentSubType`

A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component. For example, the subtype of a drawing component indicates the type of object the component draws. Drawing components that draw ovals have a subtype of 'oval'.

Your component may use this field to indicate more specific information about the capabilities of the component. There are no restrictions on the content you assign to this field. If no additional information is appropriate for your component type, you may set the `componentSubType` field to 0.

`componentManufacturer`

A four-character code that identifies the manufacturer of the component. This field allows for further differentiation between individual components. For example, components made by a specific manufacturer may support an extended feature set. Components provided by Apple use a manufacturer value of 'appl'.

Your component uses this field to indicate the manufacturer of the component. You obtain your manufacturer code, which can be the same as your application signature, from Apple's Component Registry Group.

`componentFlags`

A 32-bit field that provides additional information about a particular component.

The high-order 8 bits are reserved for definition by the Component Manager and provide information about the component. The following bits are currently defined:

```
CONST
    cmpWantsRegisterMessage = $80000000;
    cmpFastDispatch         = $40000000;
```

The setting of the `cmpWantsRegisterMessage` bit determines whether the Component Manager calls this component during registration. Set this bit to 1 if your component should be called when it is registered; otherwise, set this bit to 0. If you want to automatically dispatch requests to your component to the appropriate routine that handles the request (rather than your component calling `CallComponentFunction` or `CallComponentFunctionWithStorage`), set the `cmpFastDispatch` bit. If you set this bit, you must write your component's entry point in assembly language. If you set this bit, the Component Manager calls your component's entry point with the call's parameters, the handle to that instance's storage, and the caller's return address already on the stack. The Component Manager passes the request code in register D0 and passes the stack location of the instance's storage in register A0. Your component can then use the request code in register D0 to directly dispatch the request itself (for example, by using this value as an index into a table of function addresses). Be sure to note that the standard request codes have negative values. Also note that the function parameter that the caller uses to specify the component instance instead contains a handle to the instance's storage. When the component function completes, control returns to the calling application.

For more information about component registration and initialization, see "Responding to the Register Request" on page 6-23.

The low-order 24 bits are specific to each component type. You can use these flags to indicate any special capabilities or features of your component. Your component may use all 24 bits, as appropriate to its component type. You must set all unused bits to 0.

`componentFlagsMask`

Reserved. (However, note that applications can use this field when performing search operations, as described on page 6-39.)

Your component must set the `componentFlagsMask` field in its component description record to 0.

The Component Parameters Record

The Component Manager uses the component parameters record to pass information to your component about a request from an application. The information in this record completely defines the request. Your component services the request as appropriate.

Component Manager

The `ComponentParameters` data type defines the component parameters record.

```
ComponentParameters =
    PACKED RECORD
        flags:      Char;           {reserved}
        paramSize:  Char;           {size of parameters}
        what:       Integer;        {request code}
        params:     ARRAY[0..0] OF LongInt; {actual parameters}
    END;
```

Field descriptions

`flags` Reserved for use by Apple.

`paramSize` Specifies the number of bytes of parameter data for this request. The actual parameters are stored in the `params` field.

`what` Specifies the type of request. Component designers define the meaning of positive values and assign them to requests that are supported by components of a given type. Negative values are reserved for definition by Apple. Apple has defined these request codes:

```
CONST
    kComponentOpenSelect      = -1; {required}
    kComponentCloseSelect    = -2; {required}
    kComponentCanDoSelect    = -3; {required}
    kComponentVersionSelect  = -4; {required}
    kComponentRegisterSelect = -5; {optional}
    kComponentTargetSelect   = -6; {optional}
    kComponentUnregisterSelect = -7; {optional}
```

`params` An array that contains the parameters specified by the application that called your component. You can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` routine to convert this array into a Pascal-style invocation of a subroutine in your component.

For information on how your component responds to requests, see “Handling Requests for Service” beginning on page 6-18.

Routines for Components

This section describes the Component Manager routines that are used by components. It also discusses routines a component or application can use to register a component. This section first describes the routines for registering components then describes the routines that allow your component to

- n extract the parameters from a component parameters record and invoke a subroutine of your component with these parameters
- n manage open connections
- n associate storage with a specific connection
- n pass error information to the Component Manager for later use by the calling application
- n store and retrieve your component's reference constant
- n open and close its resource file
- n call other components
- n capture other components
- n target a component instance

Note that version 3 and above of the Component Manager supports automatic version control, the unregister request, and icon families. You should test the version number before using any of these features. You can use the `Gestalt` function with the `gestaltComponentMgr` selector to do this. When you specify this selector, `Gestalt` returns in the `response` parameter a 32-bit value indicating the version of the Component Manager that is installed.

If you are developing an application that uses components but does not register them, you do not have to read this material, though it may be interesting to you. For a discussion of the Component Manager routines that support applications that use components, see "Routines for Applications" beginning on page 6-41.

If you are developing an application that registers components, you should read the next section, "Registering Components." You may also find the other topics in this section interesting.

If you are developing a component, you should read this entire section. For more information about creating components, see "Creating Components" beginning on page 6-13.

Several of the routines discussed in this section use the component parameters record. For a complete description of that structure, see "Data Structures for Components" beginning on page 6-52. For information on the distinction between component identifiers and component instances, see page 6-40.

Note

Any of the routines discussed in this section that require a component identifier also accept a component instance. Similarly, you can supply a component identifier to any routine that requires a component instance (except for the `DelegateComponentCall` function). If you do this, you must always coerce the data type appropriately. For more information, see “Component Identifiers and Component Instances” on page 6-40. \cup

Registering Components

Before a component can be used by an application, the component must be registered with the Component Manager. The Component Manager automatically registers component resources stored in files with file types of 'thng' that are stored in the Extensions folder (for information about the content of component resources, see “Resources” beginning on page 6-80).

Alternatively, you can use either the `RegisterComponent` function or the `RegisterComponentResource` function to register components. Both applications and components can use these routines to register components.

Furthermore, you can use the `RegisterComponentResourceFile` function to register all components specified in a given resource file.

Once you have registered your component, applications can find the component and retrieve information about it using the Component Manager routines described earlier in this chapter in “Routines for Applications” beginning on page 6-41.

Finally, you can use the `UnregisterComponent` function to remove a component from the registration list.

Note

When an application quits, the Component Manager automatically closes any component connections to that application. In addition, if the application has registered components that reside in its heap space, the Component Manager automatically unregisters those components. \cup

RegisterComponent

The `RegisterComponent` function makes a component available for use by applications (or other clients). Once the Component Manager has registered a component, applications can find and open the component using the standard Component Manager routines. To register a component, you provide information identifying the component and its capabilities. The Component Manager returns a component identifier that uniquely identifies the component to the system.

Component Manager

Components you register with the `RegisterComponent` function must be in memory when you call this function. If you want to register a component that is stored in the resource fork of a file, use the `RegisterComponentResource` function. Use the `RegisterComponentResourceFile` function to register all components in the resource fork of a file.

Note that a component residing in your application heap remains registered until your application unregisters it or quits. A component residing in the system heap and registered by your application remains registered until your application unregisters it or until the computer is shut down.

```
FUNCTION RegisterComponent (cd: ComponentDescription;
                           componentEntryPoint: ComponentRoutine;
                           global: Integer;
                           componentName: Handle;
                           componentInfo: Handle;
                           componentIcon: Handle): Component;
```

`cd` A component description record that describes the component to be registered. You must correctly fill in the fields of this record before calling the `RegisterComponent` function. When applications search for components using the `FindNextComponent` function, the Component Manager compares the attributes you specify here with those specified by the application. If the attributes match, the Component Manager returns the component identifier to the application.

`componentEntryPoint` The address of the main entry point of the component you are registering. The routine referred to by this parameter receives all requests for the component.

`global` A set of flags that control the scope of component registration. You can use these flags to specify a value for the `global` parameter:

```
registerCmpGlobal = 1;
```

Specify this flag to indicate that this component should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, the component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

```
registerCmpNoDuplicates = 2;
```

Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponent` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

Component Manager

```
registerCompAfter = 4;
```

Specify this flag to indicate that this component should be registered after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

```
componentName
```

A handle to the component's name. Set this parameter to `NIL` if you do not want to assign a name to the component.

```
componentInfo
```

A handle to the component's information string. Set this parameter to `NIL` if you do not want to assign an information string to the component.

```
componentIcon
```

A handle to the component's icon (a 32-by-32 pixel black-and-white icon). Set this parameter to `NIL` if you do not want to supply an icon for this component. Note that this icon is not used by the Finder; you supply an icon only so that other components or applications can display your component's icon if needed.

DESCRIPTION

The `RegisterComponent` function registers the specified component, recording the information specified in the `cd`, `componentName`, `componentInfo`, and `componentIcon` parameters. The function returns the component identifier assigned to the component by the Component Manager. If it cannot register the component, the `RegisterComponent` function returns a function result of `NIL`.

SEE ALSO

For a complete description of the component description record, see “Data Structures for Components” beginning on page 6-52.

RegisterComponentResource

The `RegisterComponentResource` function makes a component available for use by applications (or other clients). Once the Component Manager has registered a component, applications can find and open the component using the standard Component Manager routines. You provide information identifying the component and specifying its capabilities. The Component Manager returns a component identifier that uniquely identifies the component to the system.

Components you register with the `RegisterComponentResource` function must be stored in a resource file as a component resource (see “The Component Resource” beginning on page 6-80 for a description of the format and content of component resources). If you want to register a component that is in memory, use the `RegisterComponent` function.

Component Manager

The `RegisterComponentResource` function does not actually load the code specified by the component resource into memory. Rather, the Component Manager loads the component code the first time an application opens the component. If the code is not in the same file as the component resource or if the Component Manager cannot find the file, the open request fails.

Note that a component registered locally by your application remains registered until your application unregisters it or quits. A component registered globally by your application remains registered until your application unregisters it or until the computer is shut down.

```
FUNCTION RegisterComponentResource (cr: ComponentResourceHandle;
                                   global: Integer): Component;
```

`cr` A handle to a component resource that describes the component to be registered. The component resource contains all the information required to register the component.

`global` A set of flags that controls the scope of component registration. You can use these flags to specify a value for the `global` parameter:

```
registerCmpGlobal = 1;
```

Specify this flag to indicate that this component should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, the component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

```
registerCmpNoDuplicates = 2;
```

Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponentResource` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

```
registerCompAfter = 4;
```

Specify this flag to indicate that this component should be registered after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

DESCRIPTION

The `RegisterComponentResource` function returns the component identifier assigned to the component by the Component Manager. If the `RegisterComponentResource` function could not register the component, it returns a function result of `NIL`.

SEE ALSO

For a description of the format and content of component resources, see “Resources” beginning on page 6-80.

RegisterComponentResourceFile

The `RegisterComponentResourceFile` function registers all component resources in the given resource file according to the flags specified in the `global` parameter.

```
FUNCTION RegisterComponentResourceFile (resRefNum: integer;
                                       global: integer): LongInt;
```

`resRefNum` The reference number of the resource file containing the components to register.

`global` A set of flags that control the scope of the registration of the components in the resource file specified in the `resRefNum` parameter. You can use these flags to specify a value for the `global` parameter:

```
registerCmpGlobal = 1;
```

Specify this flag to indicate that each component in the resource file should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, each component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

```
registerCmpNoDuplicates = 2;
```

Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponentResourceFile` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

```
registerCompAfter = 4;
```

Specify this flag to indicate that as `RegisterComponentResourceFile` registers a component, it should register the component after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

DESCRIPTION

The `RegisterComponentResourceFile` function registers components in a resource file. If the `RegisterComponentResourceFile` function successfully registers all components in the specified resource file, `RegisterComponentResourceFile` returns

Component Manager

a function result that indicates the number of components registered. If the `RegisterComponentResourceFile` function could not register one or more of the components in the resource file or if the specified file reference number is invalid, it returns a negative function result.

SEE ALSO

For a description of the format and content of component resources, see “Resources” beginning on page 6-80.

UnregisterComponent

The `UnregisterComponent` function removes a component from the Component Manager’s registration list. Most components are registered at startup and remain registered until the computer is shut down. However, you may want to provide some services temporarily. In that case you dispose of the component that provides the temporary service by using this function.

```
FUNCTION UnregisterComponent (aComponent: Component): OSErr;
```

`aComponent`

A component identifier that specifies the component to be removed. Applications that register components may obtain this identifier from the `RegisterComponent` or `RegisterComponentResource` functions.

DESCRIPTION

The `UnregisterComponent` function removes the component with the specified component identifier from the list of available components. The component to be removed from the registration list must not be in use by any applications or components. If there are open connections to the component, the `UnregisterComponent` function returns a negative result code.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component with this component identifier
<code>validInstancesExist</code>	-3001	This component has open connections

SEE ALSO

If you provide a component that supports the unregister request, see “Responding to the Register Request” on page 6-23 for more information.

Dispatching to Component Routines

This section discusses routines that simplify the process of calling subroutines within your component.

When an application requests service from your component, your component receives a component parameters record containing the information for that request. That component parameters record contains the parameters that the application provided when it called your component. Your component can use this record to access the parameters directly. Alternatively, you can use the routines described in this section to extract those parameters and pass them to a subroutine of your component. By taking advantage of these routines, you can simplify the structure of your component code. For more information about the interface between the Component Manager and your component, see “Creating Components” beginning on page 6-13.

Use the `CallComponentFunction` function to call a component subroutine without providing it access to global data for that connection. Use the `CallComponentFunctionWithStorage` function to call a component subroutine and to pass it a handle to the memory that stores the global data for that connection.

CallComponentFunction

The `CallComponentFunction` function invokes a specified function of your component with the parameters originally provided by the application that called your component. You pass these parameters by specifying the same component parameters record passed to your component’s main entry point.

```
FUNCTION CallComponentFunction (params: ComponentParameters;
                               func: ComponentFunction): LongInt;
```

<code>params</code>	The component parameters record that your component received from the Component Manager.
<code>func</code>	The address of the function that is to handle the request. The Component Manager calls the routine referred to by the <code>func</code> parameter as a Pascal function with the parameters that were originally provided by the application. The routine referred to by this parameter must return a function result of type <code>ComponentResult</code> (a long integer) indicating the success or failure of the operation.

DESCRIPTION

`CallComponentFunction` returns the value that is returned by the routine referred to by the `func` parameter. Your component should use this value to set the current error for this connection.

SPECIAL CONSIDERATIONS

If your component subroutine does not need global data, your component should use `CallComponentFunction`. If your component subroutine requires memory in which to store global data for the component, your component must use `CallComponentFunctionWithStorage`, which is described next.

SEE ALSO

For an example that uses `CallComponentFunction`, see Listing 6-5 on page 6-16. You can use the `SetComponentInstanceError` procedure, described on page 6-69, to set the current error.

CallComponentFunctionWithStorage

The `CallComponentFunctionWithStorage` function invokes a specified function of your component with the parameters originally provided by the application that called your component. You pass these parameters by specifying the same component parameters record that was received by your component's main entry point. The `CallComponentFunctionWithStorage` function also provides a handle to the memory associated with the current connection.

```
FUNCTION CallComponentFunctionWithStorage
    (storage: Handle; params: ComponentParameters;
     func: ComponentFunction): LongInt;
```

storage	A handle to the memory associated with the current connection. The Component Manager provides this handle to your component along with the request.
params	The component parameters record that your component received from the Component Manager.
func	The address of the function that is to handle the request. The Component Manager calls the routine referred to by the <code>func</code> parameter as a Pascal function with the parameters that were originally provided by the application. These parameters are preceded by a handle to the memory associated with the current connection. The routine referred to by the <code>func</code> parameter must return a function result of type <code>ComponentResult</code> (a long integer) indicating the success or failure of the operation.

DESCRIPTION

The `CallComponentFunctionWithStorage` function returns the value that is returned by the function referred to by the `func` parameter. Your component should use this value to set the current error for this connection.

SPECIAL CONSIDERATIONS

`CallComponentFunctionWithStorage` takes as a parameter a handle to the memory associated with the connection, so subroutines of a component that don't need global data should use the `CallComponentFunction` routine described in the previous section instead.

If your component subroutine requires a handle to the memory associated with the connection, you must use `CallComponentFunctionWithStorage`. You allocate the memory for a given connection each time your component is opened. You inform the Component Manager that a connection has memory associated with it by calling the `SetComponentInstanceStorage` procedure.

SEE ALSO

For an example that uses `CallComponentFunctionWithStorage`, see Listing 6-5 on page 6-16. Use the `SetComponentInstanceError` procedure, described on page 6-69, to set the current error for a connection. A description of the `SetComponentInstanceStorage` procedure is given next.

Managing Component Connections

The Component Manager provides a number of routines that help your component manage the connections it maintains with its client applications and components.

Use the `SetComponentInstanceStorage` procedure to inform the Component Manager of the memory your component is using to maintain global data for a connection. Whenever the client application issues a request to the connection, the Component Manager provides to your component the handle to the allocated memory for that connection along with the parameters for the request. You can also use the `GetComponentInstanceStorage` function to retrieve a handle to the storage for a connection.

Use the `CountComponentInstances` function to count all the connections that are currently maintained by your component. This routine is similar to the `CountComponents` routine that the Component Manager provides to client applications and components.

Use the `SetComponentInstanceA5` procedure to set the A5 world for a connection. Once you set the A5 world for a connection, the Component Manager automatically switches the contents of the A5 register when your component receives a request for that connection. When your component returns to the Component Manager, the Component Manager restores the A5 register. Your component can use the `GetComponentInstanceA5` function to retrieve the A5 world for a connection.

SetComponentInstanceStorage

When an application or component opens a connection to your component, the Component Manager sends your component an open request. In response to this open request, your component should set up an environment to service the connection. Typically, your component should allocate some memory for the connection. Your component can then use that memory to maintain state information appropriate to the connection.

The `SetComponentInstanceStorage` procedure allows your component to pass a handle to this memory to the Component Manager. The Component Manager then provides this handle to your component each time the client application requests service from this connection.

PROCEDURE `SetComponentInstanceStorage`

```
(aComponentInstance: ComponentInstance; theStorage: Handle);
```

`aComponentInstance`

The connection to associate with the allocated memory. The Component Manager provides a component instance to your component when the connection is opened.

`theStorage`

A handle to the memory that your component has allocated for the connection. Your component must allocate this memory in the current heap. The Component Manager saves this handle and provides it to your component, along with other parameters, in subsequent requests to this connection.

DESCRIPTION

The `SetComponentInstanceStorage` procedure associates the handle passed in the parameter `theStorage` with the connection specified by the `aComponentInstance` parameter. Your component should dispose of any allocated memory for the connection only in response to the close request.

SPECIAL CONSIDERATIONS

Note that whenever an open request fails, the Component Manager always issues the close request. Furthermore, the value stored with `SetComponentInstanceStorage` is always passed to the close request, so it must be valid or `NIL`. If the open request tries to dispose of its allocated memory before returning, it should call `SetComponentInstanceStorage` again with a `NIL` handle to keep the Component Manager from passing an invalid handle to the close request.

SEE ALSO

For an example that allocates memory in response to an open request, see Listing 6-6 on page 6-20.

GetComponentInstanceStorage

The `GetComponentInstanceStorage` function allows your component to retrieve a handle to the memory associated with a connection. Your component tells the Component Manager about this memory by calling the `SetComponentInstanceStorage` procedure. Typically, your component does not need to use this function, because the Component Manager provides this handle to your component each time the client application requests service from this connection.

```
FUNCTION GetComponentInstanceStorage
    (aComponentInstance: ComponentInstance): Handle;
```

`aComponentInstance`

The connection for which to retrieve the associated memory. The Component Manager provides a component instance to your component when the connection is opened.

DESCRIPTION

The `GetComponentInstanceStorage` function returns a handle to the memory associated with the specified connection.

CountComponentInstances

The `CountComponentInstances` function allows you to determine the number of open connections being managed by a specified component. This function can be useful if you want to restrict the number of connections for your component or if your component needs to perform special processing based on the number of open connections.

```
FUNCTION CountComponentInstances (aComponent: Component): LongInt;
```

`aComponent`

The component for which you want a count of open connections. You can use the component instance that your component received in its open request to identify your component.

DESCRIPTION

The `CountComponentInstances` function returns the number of open connections for the specified component.

SetComponentInstanceA5

The `SetComponentInstanceA5` procedure allows your component to set the A5 world for a connection.

```
PROCEDURE SetComponentInstanceA5
    (aComponentInstance: ComponentInstance; theA5: LongInt);
```

`aComponentInstance`

The connection for which to set the A5 world. The Component Manager provides a component instance to your component when the connection is opened.

`theA5`

The value of the A5 register for the connection. The Component Manager sets the A5 register to this value automatically, and it restores the previous A5 value when your component returns to the Component Manager.

DESCRIPTION

The `SetComponentInstanceA5` procedure sets the A5 world for the specified component instance. Once you set the A5 world for a connection, the Component Manager automatically switches the contents of the A5 register when your component receives a request over that connection. When your component returns to the Component Manager, the Component Manager restores your client's A5 value.

If your component has been registered globally and you have not set an A5 value, the A5 register is set to 0. In this case you should set the A5 world of your component instance to your client's A5 value by using `SetComponentInstanceA5`.

In general, your component uses this procedure only if it is registered globally; in this case, it typically calls `SetComponentInstanceA5` when processing the open request for a new connection.

GetComponentInstanceA5

You can use the `GetComponentInstanceA5` function to retrieve the value of the A5 register for a specified connection. Your component sets the A5 register by calling the `SetComponentInstanceA5` function, as previously described. The Component Manager then sets the A5 register for your component each time the client requests

Component Manager

service on this connection. If your component has been registered globally and you have not set an A5 value, the A5 register is set to 0. In this case you should use your client's A5 value.

```
FUNCTION GetComponentInstanceA5
    (aComponentInstance: ComponentInstance): LongInt;
```

aComponentInstance

The connection for which to retrieve the A5 value. The Component Manager provides a component instance to your component when the connection is opened.

DESCRIPTION

The `GetComponentInstanceA5` function returns the value of the A5 register for the connection.

Setting Component Errors

The Component Manager maintains error state information for all currently active components. In general, your component returns error information in its function result; a nonzero function result indicates an error occurred, and a function result of 0 indicates the request was successful. However, some requests require that your component return other information as its function result. In these cases, your component can use the `SetComponentInstanceError` procedure to report its latest error state to the Component Manager. You can also use this procedure at any time during your component's execution to report an error

SetComponentInstanceError

Although your component usually returns error information as its function result, your component can choose to use the `SetComponentInstanceError` procedure to pass error information to the Component Manager. The Component Manager uses this error information to set the current error value for the appropriate connection. Applications can then retrieve this error information by calling the `GetComponentInstanceError` function. The documentation for your component should specify how the component indicates errors.

```
PROCEDURE SetComponentInstanceError
    (aComponentInstance: ComponentInstance; theError: OSErr);
```

Component Manager

`aComponentInstance`

A component instance that specifies the connection for which to set the error. The Component Manager provides a component instance to your component when the connection is opened. The Component Manager also provides a component instance to your component as the first parameter in the `params` field of the parameters record.

`theError`

The new value for the current error. The Component Manager uses this value to set the current error for the connection specified by the `aComponentInstance` parameter.

DESCRIPTION

The `SetComponentInstanceError` procedure sets the error associated with the specified component instance to the value specified by the parameter `theError`.

SEE ALSO

For a description of the `GetComponentInstanceError` function, see page 6-51.

Working With Component Reference Constants

The Component Manager provides routines that manage access to the reference constants that are associated with components. There is one reference constant for each component, regardless of the number of connections to that component. When your component is registered, the Component Manager sets this reference constant to 0.

The reference constant is a 4-byte value that your component can use in any way you decide. For example, you might use the reference constant to store the address of a data structure that is shared by all connections maintained by your component. You should allocate shared structures in the system heap. Your component should deallocate the structure when its last connection is closed or when it is unregistered.

Use the `SetComponentRefcon` procedure to set the value of the reference constant for your component. Use the `GetComponentRefcon` function to retrieve the value of the reference constant.

SetComponentRefcon

You can use the `SetComponentRefcon` procedure to set the reference constant for your component.

```
PROCEDURE SetComponentRefcon (aComponent: Component;
                             theRefcon: LongInt);
```

Component Manager

aComponent

A component identifier that specifies the component whose reference constant you wish to set.

theRefCon

The reference constant value that you want to set for your component.

DESCRIPTION

The `SetComponentRefcon` procedure sets the value of the reference constant for your component. Your component can later retrieve the reference constant using the `GetComponentRefcon` function, described next.

GetComponentRefcon

The `GetComponentRefcon` function retrieves the value of the reference constant for your component.

```
FUNCTION GetComponentRefcon (aComponent: Component): LongInt;
```

aComponent

A component identifier that specifies the component whose reference constant you wish to get.

DESCRIPTION

The `GetComponentRefcon` function returns a long integer containing the reference constant for the specified component.

Accessing a Component's Resource File

If you store your component in a component resource and register your component using the `RegisterComponentResource` function or `RegisterComponentResourceFile` function, or if the Component Manager automatically registers your component, the Component Manager allows your component to gain access to its resource file. You can store read-only data for your component in its resource file. For example, the resource file may contain the color icon for the component, static data needed to initialize private storage, or any other data that may be useful to the component. Note that there is only one resource file associated with a component.

If you store your component in a component resource but register the component with the `RegisterComponent` function, rather than with the `RegisterComponentResource` or `RegisterComponentResourceFile` function, your component cannot access its resource file with the routines described in this section.

Component Manager

The routines described in this section allow your component to gain access to its resource file. These routines provide read-only access to the data in the resource file. If your component opens its resource file, it must close the file before returning to the calling application.

Use the `OpenComponentResFile` function to open your component's resource file. Use the `CloseComponentResFile` function to close the resource file before returning to the calling application.

OpenComponentResFile

The `OpenComponentResFile` function allows your component to gain access to its resource file. This function opens the resource file with read permission and returns a reference number that your component can use to read data from the file. The Component Manager adds the resource file to the current resource chain. Your component must close the resource file with the `CloseComponentResFile` function before returning to the calling application.

Your component can use `FSpOpenResFile` or equivalent Resource Manager routines to open other resource files, but you must use `OpenComponentResFile` to open your component's resource file.

```
FUNCTION OpenComponentResFile (aComponent: Component): Integer;
```

`aComponent`

A component identifier that specifies the component whose resource file you wish to open. Applications that register components may obtain this identifier from the `RegisterComponentResource` function.

DESCRIPTION

The `OpenComponentResFile` function returns a reference number for the appropriate resource file. This function returns 0 or a negative number if the specified component does not have an associated resource file or if the Component Manager cannot open the resource file.

Note that when working with resources, your component should always first save the current resource file, perform any resource operations, then restore the current resource file to its previous value before returning.

CloseComponentResFile

This function closes the resource file that your component opened previously with the `OpenComponentResFile` function.

```
FUNCTION CloseComponentResFile (refnum: Integer): OSErr;
```

`refnum` **The reference number that identifies the resource file to be closed. Your component obtains this value from the `OpenComponentResFile` function.**

DESCRIPTION

The `CloseComponentResFile` function closes the specified resource file. Your component must close any open resource files before returning to the calling application.

RESULT CODES

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

Calling Other Components

The Component Manager provides two techniques that allow a component to call other components. First, your component may invoke the services of another component using the standard mechanisms also used by applications. The Component Manager then passes the requests to the appropriate component, and your component receives the results of those requests.

Second, your component may supplement its capabilities by using the services of another component to directly satisfy application requests. The Component Manager provides the `DelegateComponentCall` function, which allows your component to pass a request to a specified component. For example, you might want to create two similar components that provide different levels of service to applications. Rather than completely implementing both components, you could design one to rely on the capabilities of the other. In this manner, you have to implement only that portion of the more capable component that provides additional services.

DelegateComponentCall

The `DelegateComponentCall` function provides an efficient mechanism for passing on requests to a specified component. Your component must open a connection to the component to which the requests are to be passed. Your component must close that connection when it has finished using the services of the other component.

Note

The `DelegateComponentCall` function does not accept a component identifier in place of a component instance. In addition, your component should never use the `DelegateComponentCall` function with open or close requests from the Component Manager—always use the `OpenComponent` and `CloseComponent` functions to manage connections with other components. [u](#)

```
FUNCTION DelegateComponentCall
    (originalParams: ComponentParameters;
     ci: ComponentInstance): LongInt;
```

`originalParams`

The component parameters record provided to your component by the Component Manager.

`ci`

The component instance that is to process the request. The Component Manager provides a component instance to your component when it opens a connection to another component with the `OpenComponent` or `OpenDefaultComponent` function. You must specify a component instance; this function does not accept a component identifier.

DESCRIPTION

The `DelegateComponentCall` function calls the component instance specified by the `ci` parameter, and passes it the specified component parameters record. `DelegateComponentCall` returns a long integer containing the component result returned by the specified component.

SEE ALSO

See “The Component Parameters Record” on page 6-54 for a description of the component parameters record. See page 6-45, page 6-46, and page 6-47, respectively, for information on the `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions.

See Listing 6-16 on page 6-36 for an example of the use of the `DelegateComponentCall` function.

Capturing Components

The Component Manager allows your component to capture another component. When a component is captured, the Component Manager removes the captured component from its list of available components. The `FindNextComponent` function does not return information about captured components. Also, other applications or clients cannot open or access captured components unless they have previously received a component identifier or component instance for the captured component. The routines described in this section allow your component to capture and uncapture other components.

Typically, your component captures another component when you want to override all or some of the features provided by a component or to provide new features. For example, a component called `NewMath` might capture a component called `OldMath`. Suppose the `NewMath` component provides a new function, `DoExponent`. Whenever `NewMath` gets an exponent request, it can handle the request itself. For all other requests, `NewMath` might call the `OldMath` component to perform the request.

After capturing a component, your component might choose to target a particular instance of the captured component. For information on targeting a component instance, see “Responding to the Target Request” beginning on page 6-25 and “Targeting a Component Instance” on page 6-77.

Use the `CaptureComponent` function to capture a component. Use the `UncaptureComponent` function to restore a previously captured component to the search list.

CaptureComponent

The `CaptureComponent` function allows your component to capture another component. In response to this function, the Component Manager removes the specified component from the search list of components. As a result, applications cannot retrieve information about the captured component or gain access to it. Current clients of the captured component are not affected by this function.

```
FUNCTION CaptureComponent (capturedComponent: Component;
                           capturingComponent: Component)
                           : Component;
```

`capturedComponent`

The component identifier of the component to be captured. Your component can obtain this identifier from the `FindNextComponent` function or from the component registration routines.

`capturingComponent`

The component identifier of your component. Note that you can use the component instance (appropriately coerced) that your component received in its open request in this parameter.

Component Manager

DESCRIPTION

The `CaptureComponent` function removes the specified component from the search list of components and returns a new component identifier. Your component can use this new identifier to refer to the captured component. For example, your component can open the captured component by providing this identifier to the `OpenComponent` function. Your component must provide this identifier to the `UncaptureComponent` function to specify the component to be restored to the search list.

If the component specified by the `capturedComponent` parameter is already captured, the `CaptureComponent` function returns a component identifier set to `NIL`.

SEE ALSO

See “Responding to the Target Request” on page 6-25 and “Targeting a Component Instance” on page 6-77 for information about target requests. For information related to the Component Manager’s use of its list of available components, see page 6-42 for details on the `FindNextComponent` function and page 6-45 for details on the `OpenDefaultComponent` function. See “Registering Components” beginning on page 6-57 for details of the component registration routines.

UncaptureComponent

The `UncaptureComponent` function allows your component to uncapture a previously captured component.

```
FUNCTION UncaptureComponent (aComponent: Component): OSErr;
```

`aComponent`

The component identifier of the component to be uncaptured. Your component obtains this identifier from the `CaptureComponent` function.

DESCRIPTION

The `UncaptureComponent` function restores the specified component to the search list of components. Applications can then access the component and retrieve information about the component using Component Manager routines.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component has this component identifier
<code>componentNotCaptured</code>	-3002	This component has not been captured

Targeting a Component Instance

Your component can target a component instance without capturing the component or your component can first capture the component and then target a specific instance of the component. For information on capturing components, see “Capturing Components” beginning on page 6-75. To target a component instance, use the `ComponentSetTarget` function.

ComponentSetTarget

You can use the `ComponentSetTarget` function to call a component’s target request routine (that is, the routine that handles the `kComponentTargetSelect` request code). The target request informs a component that it has been targeted by another component.

You should not target a component instance if the component does not support the target request. Before calling this function, you should issue a can do request to the component instance you want to target to verify that the component supports the target request. If the component supports it, use the `ComponentSetTarget` function to send a target request to the component instance you wish to target. After receiving a target request, the targeted component instance should call the component instance that targeted it whenever the targeted component instance would normally call one of its defined functions.

```
FUNCTION ComponentSetTarget (ci: ComponentInstance;
                             target: ComponentInstance): LongInt;
```

`ci` The component instance to which to send a target request (the component that has been targeted).

`target` The component instance of the component issuing the target request.

DESCRIPTION

The `ComponentSetTarget` function returns a function result of `badComponentSelector` if the targeted component does not support the target request. Otherwise, the `ComponentSetTarget` function returns as its function result the value that the targeted component instance returned in response to the target request.

SEE ALSO

For details on how to handle the target request, see “Responding to the Target Request” on page 6-25.

Changing the Default Search Order

You can use the `SetDefaultComponent` function to change the order in which the list of registered components is searched.

SetDefaultComponent

The `SetDefaultComponent` function allows your component to change the search order for registered components. You specify a component that is to be placed at the front of the search chain, along with control information that governs the reordering operation. The order of the search chain influences which component the Component Manager selects in response to an application's use of the `OpenDefaultComponent` and `FindNextComponent` functions.

```
FUNCTION SetDefaultComponent (aComponent: Component;
                             flags: Integer): OSErr;
```

`aComponent`

A component identifier that specifies the component for this operation.

`flags`

A value specifying the control information governing the operation. The value of this parameter controls which component description fields the Component Manager examines during the reorder operation. Set the appropriate flags to 1 to define the fields that are examined during the reorder operation. The following flags are defined:

`defaultComponentIdentical`

The Component Manager places the specified component in front of all other components that have the same component description.

`defaultComponentAnyFlags`

The Component Manager ignores the value of the `componentFlags` field during the reorder operation.

`defaultComponentAnyManufacturer`

The Component Manager ignores the value of the `componentManufacturer` field during the reorder operation.

`defaultComponentAnySubType`

The Component Manager ignores the value of the `componentSubType` field during the reorder operation.

DESCRIPTION

The `SetDefaultComponent` function changes the search order of registered components by moving the specified component to the front of the search chain, according to the value specified in the `flags` parameter.

SPECIAL CONSIDERATIONS

Note that the `SetDefaultComponent` function changes the search order for all applications. As a result, you should use this function carefully.

RESULT CODES

<code>noErr</code>	0	No error
<code>invalidComponentID</code>	-3000	No component has this component identifier

Application-Defined Routine

To provide a component, you define a component function and supply the appropriate registration information. You store your component function in a code resource and typically store your component's registration information as resources in a component file. For additional information on this process, see "Creating Components" beginning on page 6-13.

MyComponent

Here's how to declare a component function named `MyComponent`:

```
FUNCTION MyComponent (params: ComponentParameters;
                    storage: Handle): ComponentResult;
```

`params` A component parameters record. The `what` field of the component parameters record indicates the action your component should perform. The parameters that the client invoked your function with are contained in the `params` field of the component parameters record. Your component can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` routine to extract the parameters from this record.

`storage` A handle to any memory that your component has associated with the connection. Typically, upon receiving an open request, your component allocates memory and uses the `SetComponentInstanceStorage` function to associate the allocated memory with the component connection.

DESCRIPTION

When your component receives a request, it should perform the action specified in the `what` field of the component parameters record. Your component should return a value of type `ComponentResult` (a long integer). If your component does not return error information as its function result, it should indicate errors using the `SetComponentInstanceError` procedure.

SEE ALSO

For information on the component parameters record, see page 6-54. For information on writing a component, see “Creating Components” beginning on page 6-13.

Resources

This section describes the resource you use to define your component. If you are developing a component, you should be familiar with the format and content of a component resource.

The Component Resource

A component resource (a resource of type `'thng'`) stores all of the information about a component in a single file. The component resource contains all the information needed to register a code resource as a component. Information in the component resource tells the Component Manager where to find the code for the component.

If you are developing an application that uses components, you do not need to know about component resources.

If you are developing a component or an application that registers components, you should be familiar with component resources. The Component Manager automatically registers any components that are stored in component files in the Extensions folder. The file type for component files must be set to `'thng'`. If you store your component in a component file in the Extensions folder, you do not need to create an application to register the component.

The Component Manager provides routines that register components. The `RegisterComponent` function registers components that are not stored in resource files. The `RegisterComponentResource` and `RegisterComponentResourceFile` functions register components that are stored as component resources in a component file. If you are developing an application that registers components, you should use the routine that is appropriate to the storage format of the component. For more information about how your application can register components, see “Registering Components” beginning on page 6-57.

Component Manager

This section describes the component resource, which must be provided by all components stored in a component file. Applications that register a component using the `RegisterComponent` function must also provide the same information as that contained in a component resource.

IMPORTANT

For compatibility with early versions of the Component Manager, a component resource must be locked. s

The `ComponentResource` data type defines the structure of a component resource. (You can also optionally append to the end of this structure the information defined by the `ComponentResourceExtension` data type, as shown in Figure 6-5 on page 6-85.)

```
ComponentResource =
    RECORD
        cd:                                {registration information}
                                ComponentDescription;
        component: ResourceSpec; {code resource}
        componentName: ResourceSpec; {name string resource}
        componentInfo: ResourceSpec; {info string resource}
        componentIcon: ResourceSpec; {icon resource}
    END;
```

Field descriptions

<code>cd</code>	A component description record that specifies the characteristics of the component. For a complete description of this record, see page 6-52.
<code>component</code>	A resource specification record that specifies the type and ID of the component code resource. The <code>resType</code> field of the resource specification record may contain any value. The component's main entry point must be at offset 0 in the resource.
<code>componentName</code>	A resource specification record that specifies the resource type and ID for the name of the component. This is a Pascal string. Typically, the component name is stored in a resource of type 'STR'.
<code>componentInfo</code>	A resource specification record that specifies the resource type and ID for the information string that describes the component. This is a Pascal string. Typically, the information string is stored in a resource of type 'STR'. You might use the information stored in this resource in a Get Info dialog box.
<code>componentIcon</code>	A resource specification record that specifies the resource type and ID for the icon for a component. Component icons are stored as 32-by-32 bit maps. Typically, the icon is stored in a resource of type 'ICON'. Note that this icon is not used by the Finder; you supply an icon only so that other components or applications can display your component's icon in a dialog box if needed.

Component Manager

A resource specification record, defined by the data type `ResourceSpec`, describes the resource type and resource ID of the component's code, name, information string, or icon. The resources specified by the resource specification records must reside in the same resource file as the component resource itself.

```
ResourceSpec =
    RECORD
        resType:      OSType;          {resource type}
        resId:        Integer;         {resource ID}
    END;
```

You can optionally include in your component resource the information defined by the `ComponentResourceExtension` data type:

```
ComponentResourceExtension =
    RECORD
        componentVersion:      LongInt; {version of component}
        componentRegisterFlags: LongInt; {additional flags}
        componentIconFamily:   Integer; {resource ID of icon }
                                { family}
    END;
```

Field descriptions`componentVersion`

The version number of the component. If you specify the `componentDoAutoVersion` flag in `componentRegisterFlags`, the Component Manager must obtain the version number of your component when your component is registered. Either you can provide a version number in your component's resource, or you can specify a value of 0 for its version number. If you specify 0, the Component Manager sends your component a version request to get the version number of your component.

`componentRegisterFlags`

A set of flags containing additional registration information. You can use these constants as flags:

```
CONST
    componentDoAutoVersion      = 1;
    componentWantsUnregister    = 2;
    componentAutoVersionIncludeFlags = 4;
```

Specify the `componentDoAutoVersion` flag if you want the Component Manager to resolve conflicts between different versions of the same component. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Specify the `componentWantsUnregister` flag if you want your component to receive an unregister request when it is unregistered.

Specify the flag `componentAutoVersionIncludeFlags` if you want the Component Manager to include the `componentFlags` field of the component description record when it searches for identical components in the process of performing automatic version control for your component. If you do not specify this flag, the Component Manager searches only the `componentType`, `componentSubType`, and `componentManufacturer` fields.

When the Component Manager performs automatic version control for your component, it searches for components with identical values in the `componentType`, `componentSubType`, and `componentManufacturer` fields (and optionally, in the `componentFlags` field). If it finds a matching component, it compares version numbers and registers the most recent version of the component. Note that the setting of the `componentAutoVersionIncludeFlags` flag affects automatic version control only and does not affect the search operations performed by `FindNextComponent` and `CountComponents`.

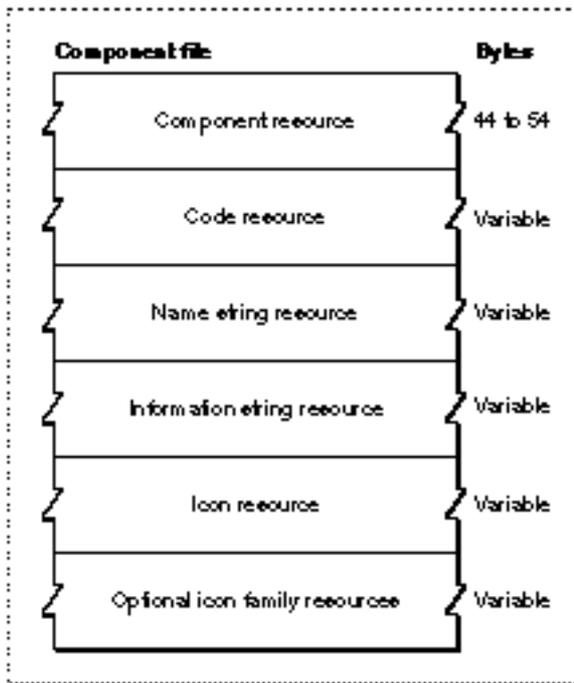
`componentIconFamily`

The resource ID of an icon family. You can provide an icon family in addition to the icon provided in the `componentIcon` field. Note that members of this icon family are not used by the Finder; you supply an icon family only so that other components or applications can display your component's icon in a dialog box if needed.

Component Manager

You store a component resource, along with other resources for the component, in the resource fork of a component file. Figure 6-4 shows the structure of a component file.

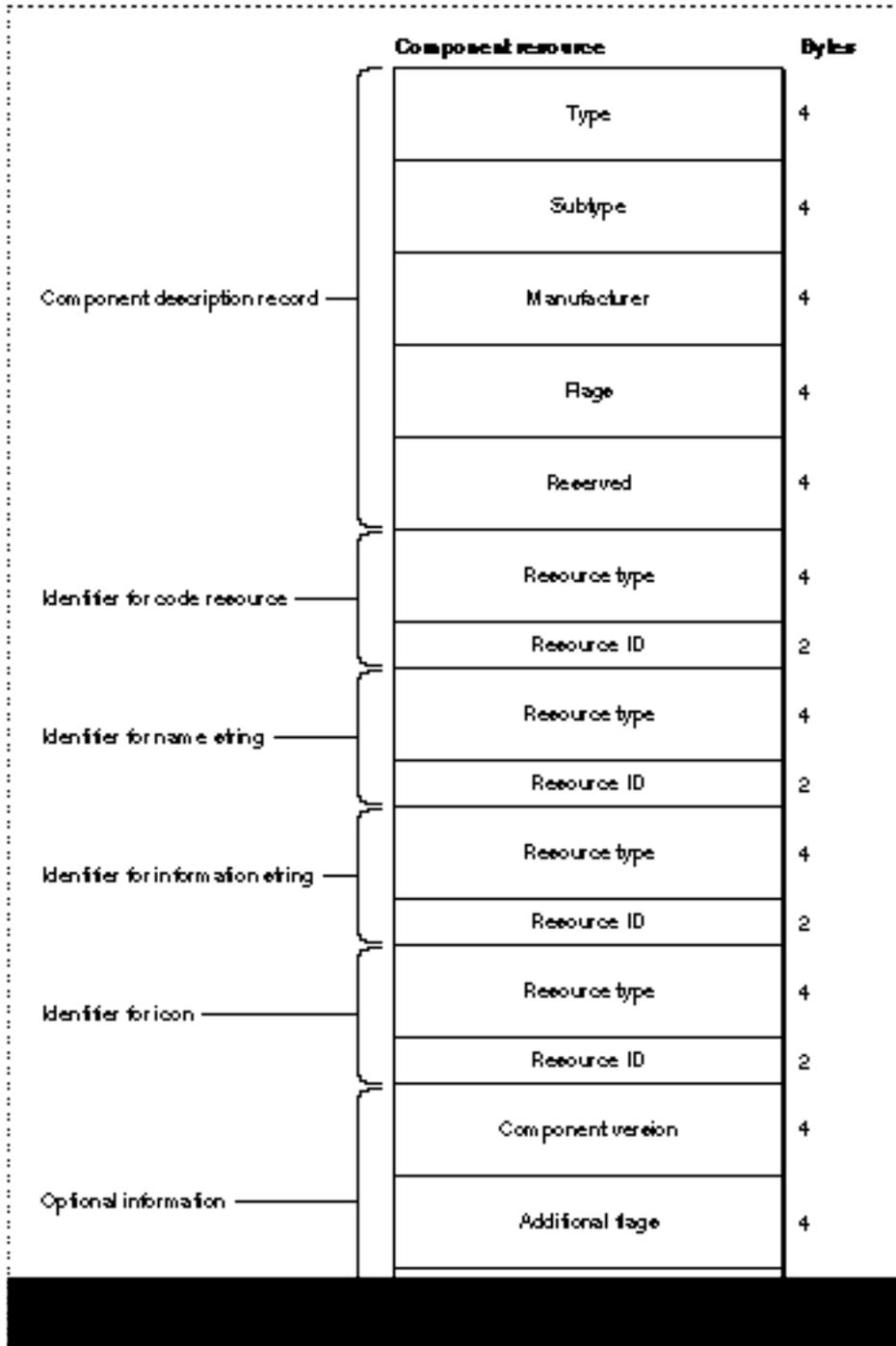
Figure 6-4 Format of a component file



You can also store other resources for your component in your component file. For example, you should include 'FREF', 'BNDL', and icon family resources so that the Finder can associate the icon identifying your component with your component file. When designing the icon for your component file, you should follow the same guidelines as those for system extension icons. See *Macintosh Human Interface Guidelines* for information on designing an icon. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the 'FREF' and 'BNDL' resources.

Figure 6-5 shows the structure of a component resource.

Figure 6-5 Structure of a compiled component ('thng') resource



Summary of the Component Manager

Pascal Summary

Constants

CONST

```

gestaltComponentMgr          = 'cpnt';

kComponentOpenSelect        = -1;    {open request}
kComponentCloseSelect       = -2;    {close request}
kComponentCanDoSelect       = -3;    {can do request}
kComponentVersionSelect     = -4;    {version request}
kComponentRegisterSelect    = -5;    {register request}
kComponentTargetSelect      = -6;    {target request}
kComponentUnregisterSelect  = -7;    {unregister request}

{wildcard values for searches}
kAnyComponentType           = 0;     {any type}
kAnyComponentSubType        = 0;     {any subtype}
kAnyComponentManufacturer   = 0;     {any manufacturer}
kAnyComponentFlagsMask     = 0;     {any flags}

{component description flag}
cmpWantsRegisterMessage     = $80000000; {send register request}
{flags for optional extension to component resource}
componentDoAutoVersion      = 1;     {provide version control}
componentWantsUnregister    = 2;     {send unregister request}
componentAutoVersionIncludeFlags = 4; {include flags in search}

{flags for SetDefaultComponent function}
defaultComponentIdentical   = 0;
defaultComponentAnyFlags    = 1;
defaultComponentAnyManufacturer = 2;
defaultComponentAnySubType  = 4;
defaultComponentAnyFlagsAnyManufacturer
                             = defaultComponentAnyFlags +
                               defaultComponentAnyManufacturer;

```

CHAPTER 6

Component Manager

```
defaultComponentAnyFlagsAnyManufacturerAnySubType
    = defaultComponentAnyFlags
      + defaultComponentAnyManufacturer
      + defaultComponentAnySubType;

{flags for the global parameter of RegisterComponentResourceFile function}
registerCmpGlobal      = 1;  {other apps can communicate with component}
registerCmpNoDuplicates = 2; {don't register if duplicate component }
                        { exists}
registerCompAfter      = 4;  {component registered after all others of }
                        { same type}
```

Data Types

TYPE

```
ComponentDescription =
RECORD
    componentType:          OSType;  {type}
    componentSubType:       OSType;  {subtype}
    componentManufacturer:  OSType;  {manufacturer}
    componentFlags:         LongInt; {control flags}
    componentFlagsMask:     LongInt; {mask for control flags }
                                { (reserved when }
                                { registering a component)}

END;

ResourceSpec =
RECORD
    resType:                OSType;  {resource type}
    resID:                  Integer; {resource ID}

END;

ComponentResourcePtr      = ^ComponentResource;
ComponentResourceHandle   = ^ComponentResourcePtr;
ComponentResource =
                                {component resource}
RECORD
    cd:                      ComponentDescription;  {registration information}
    component:               ResourceSpec;           {code resource}
    componentName:          ResourceSpec;           {name string resource}
    componentInfo:           ResourceSpec;           {info string resource}
    componentIcon:           ResourceSpec;           {icon resource}

END;
```

CHAPTER 6

Component Manager

```
ComponentResourceExtension =           {optional extension to resource}
RECORD
    componentVersion:      LongInt; {version of component}
    componentRegisterFlags: LongInt; {additional flags}
    componentIconFamily:   Integer; {resource ID of icon }
                                { family}
END;
{component parameters record}
ComponentParameters =
    PACKED RECORD
        flags:      Char;           {reserved}
        paramSize:  Char;           {size in bytes of actual }
                                { parameters passed to }
                                { this routine}
        what:       Integer;        {request code- }
                                { negative for requests }
                                { defined by Component Mgr}
        params:    ARRAY[0..0] OF LongInt; {actual parameters for }
                                { the indicated routine}
    END;

{component identifier}
Component          = ^ComponentRecord;
ComponentRecord   =
RECORD
    data:          ARRAY[0..0] OF LongInt;
END;

{component instance}
ComponentInstance = ^ComponentInstanceRecord;
ComponentInstanceRecord =
RECORD
    data:          ARRAY[0..0] OF LongInt;
END;

ComponentResult   = LongInt;
ComponentRoutine  = ProcPtr;
ComponentFunction = ProcPtr;
```

Routines for Applications

Finding Components

```

FUNCTION FindNextComponent (aComponent: Component;
                           looking: ComponentDescription): Component;
FUNCTION CountComponents (looking: ComponentDescription): LongInt;
FUNCTION GetComponentListModSeed: LongInt;

```

Opening and Closing Components

```

FUNCTION OpenDefaultComponent
    (componentType: OSType;
     componentSubType: OSType): ComponentInstance;
FUNCTION OpenComponent (aComponent: Component): ComponentInstance;
FUNCTION CloseComponent (aComponentInstance: ComponentInstance): OSErr;

```

Getting Information About Components

```

FUNCTION GetComponentInfo (aComponent: Component;
                          VAR cd: ComponentDescription;
                          componentName: Handle; componentInfo: Handle;
                          componentIcon: Handle): OSErr;
FUNCTION GetComponentIconSuite
    (aComponent: Component;
     VAR iconSuite: Handle): OSErr;
FUNCTION GetComponentVersion
    (ci: ComponentInstance): LongInt;
FUNCTION ComponentFunctionImplemented
    (ci: ComponentInstance; ftnNumber: Integer)
    : LongInt;

```

Retrieving Component Errors

```

FUNCTION GetComponentInstanceError
    (aComponentInstance: ComponentInstance): OSErr;

```

Routines for Components

Registering Components

```

FUNCTION RegisterComponent (cd: ComponentDescription;
                           componentEntryPoint: ComponentRoutine;
                           global: Integer; componentName: Handle;
                           componentInfo: Handle;
                           componentIcon: Handle): Component;

FUNCTION RegisterComponentResource
                           (cr: ComponentResourceHandle;
                           global: Integer): Component;

FUNCTION RegisterComponentResourceFile
                           (resRefNum: integer; global: integer): LongInt;

FUNCTION UnregisterComponent
                           (aComponent: Component): OSErr;

```

Dispatching to Component Routines

```

FUNCTION CallComponentFunction
                           (params: ComponentParameters;
                           func: ComponentFunction): LongInt;

FUNCTION CallComponentFunctionWithStorage
                           (storage: Handle;
                           params: ComponentParameters;
                           func: ComponentFunction): LongInt;

```

Managing Component Connections

```

PROCEDURE SetComponentInstanceStorage
                           (aComponentInstance: ComponentInstance;
                           theStorage: Handle);

FUNCTION GetComponentInstanceStorage
                           (aComponentInstance: ComponentInstance): Handle;

FUNCTION CountComponentInstances
                           (aComponent: Component): LongInt;

PROCEDURE SetComponentInstanceA5
                           (aComponentInstance: ComponentInstance;
                           theA5: LongInt);

FUNCTION GetComponentInstanceA5
                           (aComponentInstance: ComponentInstance)
                           : LongInt;

```

Setting Component Errors

```
PROCEDURE SetComponentInstanceError
    (aComponentInstance: ComponentInstance;
     theError: OSErr);
```

Working With Component Reference Constants

```
PROCEDURE SetComponentRefcon
    (aComponent: Component; theRefcon: LongInt);

FUNCTION GetComponentRefcon
    (aComponent: Component): LongInt;
```

Accessing a Component's Resource File

```
FUNCTION OpenComponentResFile
    (aComponent: Component): Integer;

FUNCTION CloseComponentResFile
    (refnum: Integer): OSErr;
```

Calling Other Components

```
FUNCTION DelegateComponentCall
    (originalParams: ComponentParameters;
     ci: ComponentInstance): LongInt;
```

Capturing Components

```
FUNCTION CaptureComponent
    (capturedComponent: Component;
     capturingComponent: Component): Component;

FUNCTION UncaptureComponent
    (aComponent: Component): OSErr;
```

Targeting a Component Instance

```
FUNCTION ComponentSetTarget (ci: ComponentInstance;
                             target: ComponentInstance): LongInt;
```

Changing the Default Search Order

```
FUNCTION SetDefaultComponent
    (aComponent: Component; flags: Integer): OSErr;
```

Application-Defined Routine

```
FUNCTION MyComponent          (params: ComponentParameters;
                             storage: Handle): ComponentResult;
```

C Summary

Constants

```
#define gestaltComponentMgr 'cpnt'          /*Gestalt selector*/

/*required component routines*/
#define kComponentOpenSelect      -1      /*open request*/
#define kComponentCloseSelect     -2      /*close request*/
#define kComponentCanDoSelect     -3      /*can do request*/
#define kComponentVersionSelect   -4      /*version request*/
#define kComponentRegisterSelect  -5      /*register request*/
#define kComponentTargetSelect    -6      /*target request*/
#define kComponentUnregisterSelect -7      /*unregister request*/

/*wildcard values for searches*/
#define kAnyComponentType         0       /*any type*/
#define kAnyComponentSubType      0       /*any subtype*/
#define kAnyComponentManufacturer 0       /*any manufacturer*/
#define kAnyComponentFlagsMask   0       /*any flags*/

/*component description flags*/
enum {
    cmpWantsRegisterMessage = 1L<<31    /*send register request*/
};

/*flags for optional extension to component resource*/
enum {
    componentDoAutoVersion        = 1,    /*provide version control*/
    componentWantsUnregister      = 2,    /*send unregister request*/
    componentAutoVersionIncludeFlags = 4   /*include flags in search*/
};

enum { /*flags for SetDefaultComponent function*/
    defaultComponentIdentical      = 0,
    defaultComponentAnyFlags       = 1,
    defaultComponentAnyManufacturer = 2,
```

CHAPTER 6

Component Manager

```
defaultComponentAnySubType      = 4,
};
#define defaultComponentAnyFlagsAnyManufacturer
    (defaultComponentAnyFlags+defaultComponentAnyManufacturer)
#define defaultComponentAnyFlagsAnyManufacturerAnySubType
    (defaultComponentAnyFlags+defaultComponentAnyManufacturer
    +defaultComponentAnySubType)

enum {
/*flags for the global parameter of RegisterComponentResourceFile function*/
    registerCmpGlobal      = 1, /*other apps can communicate with */
                            /* component*/
    registerCmpNoDuplicates = 2, /*duplicate component exists*/
    registerCmpAfter       = 4 /*component registered after all others */
                            /* of same type*/
};
```

Data Structures

```
struct ComponentDescription {
    OSType      componentType;      /*type*/
    OSType      componentSubType;   /*subtype*/
    OSType      componentManufacturer; /*manufacturer*/
    unsigned long componentFlags;    /*control flags*/
    unsigned long componentFlagsMask; /*mask for control flags */
                                        /* (reserved when registering */
                                        /* a component)*/
};
typedef struct ComponentDescription ComponentDescription;

struct ResourceSpec {
    OSType      ResType;      /*resource type*/
    short       ResID;        /*resource ID*/
};
typedef struct ResourceSpec ResourceSpec;
```

Component Manager

```

struct ComponentResource {
    ComponentDescription cd;           /*registration information*/
    ResourceSpec          component;    /*code resource*/
    ResourceSpec          componentName; /*name string resource*/
    ResourceSpec          componentInfo; /*info string resource*/
    ResourceSpec          componentIcon; /*icon resource*/
};
typedef struct ComponentResource ComponentResource;
typedef ComponentResource *ComponentResourcePtr, **ComponentResourceHandle;

/*optional extension to component resource*/
struct ComponentResourceExtension {
    long          componentVersion;    /*version number*/
    long          componentRegisterFlags; /*additional flags*/
    short        componentIconFamily;  /*resource ID of icon family*/
};
typedef struct ComponentResourceExtension ComponentResourceExtension;
/*structure received by component*/
struct ComponentParameters {
    unsigned char flags;    /*reserved*/
    unsigned char paramSize; /*size in bytes of actual parameters passed */
                          /* to this routine*/
    short        what;     /*request code, negative for requests */
                          /* defined by Component Mgr*/
    long        params[1]; /*actual parameters for the indicated */
                          /* routine*/
};
typedef struct ComponentParameters ComponentParameters;

/*component identifier*/
typedef struct privateComponentRecord *Component;
/*component instance*/
typedef struct privateComponentInstanceRecord *ComponentInstance;

typedef long ComponentResult;

typedef pascal ComponentResult (*ComponentRoutine)
    (ComponentParameters *cp, Handle componentStorage);
typedef pascal ComponentResult (*ComponentFunction)();

#define ComponentCallNow(callNumber, paramSize) \
    {0x2F3C, paramSize, callNumber, 0x7000, 0xA82A}

```

Routines for Applications

Finding Components

```

pascal Component FindNextComponent
    (Component aComponent,
     ComponentDescription *looking);

pascal long CountComponents
    (ComponentDescription *looking);

pascal long GetComponentListModSeed
    (void);

```

Opening and Closing Components

```

pascal ComponentInstance OpenDefaultComponent
    (OSType componentType,
     OSType componentSubType);

pascal ComponentInstance OpenComponent
    (Component aComponent);

pascal OSErr CloseComponent
    (ComponentInstance aComponentInstance);

```

Getting Information About Components

```

pascal OSErr GetComponentInfo
    (Component aComponent,
     ComponentDescription *cd,
     Handle componentName, Handle componentInfo,
     Handle componentIcon);

pascal OSErr GetComponentIconSuite
    (Component aComponent,
     Handle *iconSuite);

pascal long GetComponentVersion
    (ComponentInstance ci);

pascal long ComponentFunctionImplemented
    (ComponentInstance ci, short ftnNumber);

```

Retrieving Component Errors

```

pascal OSErr GetComponentInstanceError
    (ComponentInstance aComponentInstance);

```

Routines for Components

Registering Components

```
pascal Component RegisterComponent
    (ComponentDescription *cd,
     ComponentRoutine componentEntryPoint,
     short global, Handle componentName,
     Handle componentInfo, Handle componentIcon);

pascal Component RegisterComponentResource
    (ComponentResourceHandle cr, short global);

pascal long RegisterComponentResourceFile
    (short resRefNum, short global);

pascal OSErr UnregisterComponent
    (Component aComponent);
```

Dispatching to Component Routines

```
pascal long CallComponentFunction
    (ComponentParameters *params,
     ComponentFunction func);

pascal long CallComponentFunctionWithStorage
    (Handle storage, ComponentParameters *params,
     ComponentFunction func);
```

Managing Component Connections

```
pascal void SetComponentInstanceStorage
    (ComponentInstance aComponentInstance,
     Handle theStorage);

pascal Handle GetComponentInstanceStorage
    (ComponentInstance aComponentInstance);

pascal long CountComponentInstances
    (Component aComponent);

pascal void SetComponentInstanceA5
    (ComponentInstance aComponentInstance,
     long theA5);

pascal long GetComponentInstanceA5
    (ComponentInstance aComponentInstance);
```

Setting Component Errors

```
pascal void SetComponentInstanceError
    (ComponentInstance aComponentInstance,
     OSErr theError);
```

Working With Component Reference Constants

```
pascal void SetComponentRefcon
                (Component aComponent, long theRefcon);
pascal long GetComponentRefcon
                (Component aComponent);
```

Accessing a Component's Resource File

```
pascal short OpenComponentResFile
                (Component aComponent);
pascal OSErr CloseComponentResFile
                (short refnum);
```

Calling Other Components

```
pascal long DelegateComponentCall
                (ComponentParameters *originalParams,
                 ComponentInstance ci);
```

Capturing Components

```
pascal Component CaptureComponent
                (Component capturedComponent,
                 Component capturingComponent);
pascal OSErr UncaptureComponent
                (Component aComponent);
```

Targeting a Component Instance

```
pascal long ComponentSetTarget
                (ComponentInstance ci,
                 ComponentInstance target);
```

Changing the Default Search Order

```
pascal OSErr SetDefaultComponent
                (Component aComponent, short flags);
```

Application-Defined Routine

```
pascal ComponentResult MyComponent
                (ComponentParameters* params,
                 Handle storage);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

`_ComponentDispatch`

Selector	Routine
\$7001	RegisterComponent
\$7002	UnregisterComponent
\$7003	CountComponents
\$7004	FindNextComponent
\$7005	GetComponentInfo
\$7006	GetComponentListModSeed
\$7007	OpenComponent
\$7008	CloseComponent
\$700A	GetComponentInstanceError
\$700B	SetComponentInstanceError
\$700C	GetComponentInstanceStorage
\$700D	SetComponentInstanceStorage
\$700E	GetComponentInstanceA5
\$700F	SetComponentInstanceA5
\$7010	GetComponentRefcon
\$7011	SetComponentRefcon
\$7012	RegisterComponentResource
\$7013	CountComponentInstances
\$7014	RegisterComponentResourceFile
\$7015	OpenComponentResFile
\$7018	CloseComponentResFile
\$701C	CaptureComponent
\$701D	UncaptureComponent
\$701E	SetDefaultComponent
\$7021	OpenDefaultComponent
\$7024	DelegateComponentCall
\$70FF	CallComponentFunction
\$70FF	CallComponentFunctionWithStorage

Result Codes

noErr	0	No error
resFNotFound	-193	Resource file not found
invalidComponentID	-3000	No component has this component identifier
validInstancesExist	-3001	This component has open connections
componentNotCaptured	-3002	This component has not been captured
badComponentInstance	\$800008001	Invalid component passed to Component Manager
badComponentSelector	\$800008002	Component does not support the specified request code

Translation Manager

Contents

About the Translation Manager	7-4
Opening Documents From the Finder	7-5
Opening Documents Within an Application	7-8
Translating Documents on the Desktop	7-9
Sharing Data Between Applications	7-10
Using the Translation Manager	7-10
Checking for the Translation Manager	7-12
Declaring the File Types Your Application Can Open	7-13
Declaring Custom Kind Strings	7-14
Using File-Opening Dialog Boxes	7-15
Translating Files Explicitly	7-17
Writing a Translation Extension	7-18
Creating a Translation Extension	7-19
Dispatching to Translation Extension-Defined Routines	7-24
Creating a Translation List	7-27
Identifying Files	7-32
Translating Files	7-33
Writing Application Translation Extensions	7-35
Translation Manager Reference	7-36
Translation Manager Routines	7-36
Getting Translation Information	7-37
Translating Files	7-42
Resources	7-43
The Open Resource	7-44
The Kind Resource	7-45
Translation Extension Reference	7-46
Translation Extension Data Structures	7-46
File Type Specifications	7-46
File Translation Lists	7-48
Scrap Type Specifications	7-49

Scrap Translation Lists	7-49	
Translation Extension Routines	7-50	
Managing Translation Progress Dialog Boxes		7-50
Translation Extension-Defined Routines	7-54	
File Translation Extension Routines	7-54	
Scrap Translation Extension Routines	7-58	
Summary of the Translation Manager	7-63	
Pascal Summary	7-63	
Constants	7-63	
Data Types	7-63	
Translation Manager Routines	7-64	
C Summary	7-64	
Constants	7-64	
Data Types	7-65	
Translation Manager Routines	7-65	
Assembly-Language Summary	7-66	
Data Structures	7-66	
Trap Macros	7-66	
Result Codes	7-67	
Summary of Translation Extensions	7-68	
Pascal Summary	7-68	
Constants	7-68	
Data Types	7-68	
Translation Extension Routines	7-70	
Translation Extension-Defined Routines	7-70	
C Summary	7-71	
Constants	7-71	
Data Types	7-71	
Translation Extension Routines	7-73	
Translation Extension-Defined Routines	7-73	
Assembly-Language Summary	7-74	
Data Structures	7-74	
Trap Macros	7-75	
Result Codes	7-75	

Translation Manager

This chapter describes how you can use the Translation Manager to direct the translation of documents from one format to another. This chapter also gives an overview of Macintosh Easy Open. Macintosh Easy Open uses the Translation Manager to provide extensive data translation services for Macintosh computers. Macintosh Easy Open uses the Translation Manager to provide

- n automatic translation of a document from one format to some other format if the application that created it is not available when the user attempts to open the document
- n automatic translation of documents drop-launched onto an application
- n enhanced Standard File Package file-opening dialog boxes and (when necessary) automatic translation of documents the user selects in those dialog boxes
- n batch desktop translation of documents
- n automatic translation of data pasted from the Clipboard
- n automatic translation of data in editions

Most applications take advantage of these services automatically if they use the system software to open documents. You can, however, enhance your application's interaction with Macintosh Easy Open by adding several resources to your application's resource file. For example, the Finder and Macintosh Easy Open work with the Standard File Package to list in the file opening dialog box all files that your application can open, including those that it can open only after they have been translated from their current format to another format. See "Declaring the File Types Your Application Can Open" on page 7-13 for instructions on adding the required resources to your application. If, however, your application doesn't use the Standard File Package when opening files, you might also need to use the Translation Manager to direct the translation. See "Translating Files Explicitly" on page 7-17 for more information.

This chapter also describes how to write a translation extension. Macintosh Easy Open doesn't do any translating itself; instead, it uses translation extensions to translate documents (data in files) and scraps (data in memory) in the situations listed above. Translation extensions also need to be able to report the kinds of files or scraps they can handle and to identify specific files that need to be translated. You're likely to need to write a translation extension only if you are developing file or scrap translators (also known as convertors or filters).

Macintosh Easy Open and the Translation Manager are not available in all system software versions. You should use the `Gestalt` function to ensure that the services you need are available before calling them. See "Checking for the Translation Manager" on page 7-12 for details.

To use this chapter, you should already be familiar with the Standard File Package, the Scrap Manager, the Edition Manager, Finder-related resources, and the Component Manager. For information on the Standard File Package, see *Inside Macintosh: Files*. For information on the Scrap Manager and the Component Manager, see their corresponding chapters in this book. For information on the Edition Manager, see *Inside Macintosh: Interapplication Communication*. For information on Finder-related resources, see *Inside Macintosh: Macintosh Toolbox Essentials*.

About the Translation Manager

The Translation Manager provides extensive data translation services for Macintosh computers. **Macintosh Easy Open** uses the Translation Manager to provide four basic services:

- n translation of documents opened from the Finder
- n automatic translation of documents opened by applications that use the Standard File Package
- n batch translation of documents at the desktop level
- n automatic translation of data in editions or pasted from the Clipboard

These services allow your application to open documents created by other applications (possibly running on other operating systems) and to import data from other applications with better fidelity than previously possible.

Macintosh Easy Open provides the services that the Finder and the Standard File Package use to implement **implicit translation** (the conversion of a file or scrap without direct intervention from the application). The Finder needs to know which applications are capable of opening a document, either directly or after the document has been translated to another file format. The Standard File Package needs to know which other file types can be translated to some file type that the application can read. Both the Finder and the Standard File Package then call Macintosh Easy Open to translate a file to another format.

Macintosh Easy Open does not do any translating itself, and it does not have any knowledge of translation data models. Instead, it delegates these functions to translation extensions or to applications with built-in translation capability. Translation extensions and application translation capabilities operate as “black boxes” to Macintosh Easy Open.

A **translation extension** is responsible for many things, including recognizing and translating files or scraps. A translation extension might be a complete entity, able to recognize and translate all by itself. Other translation extensions might require external files, usually called translators or filters, to perform their work. In either case, the whole is called a **translation system**.

At system startup (or whenever new translation extensions become available), Macintosh Easy Open catalogs the translation capability of each translation extension and each application, and then invokes each as needed. Macintosh Easy Open can support multiple translation systems.

Translation Manager

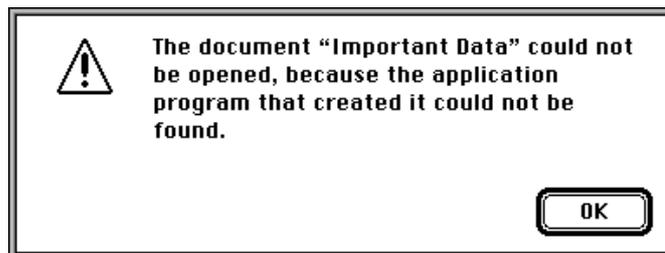
There are two types of translation systems: file translation systems and scrap translation systems. A **file translation system** can translate from one file format to another. A **scrap translation system** can translate buffers in memory. Macintosh Easy Open distinguishes between the two because a file format in memory might differ from the same file format on disk. A single translation system, however, might contain both kinds of translators.

The following four sections describe in greater detail the capabilities of Macintosh Easy Open and its interactions with other pieces of the Macintosh system software.

Opening Documents From the Finder

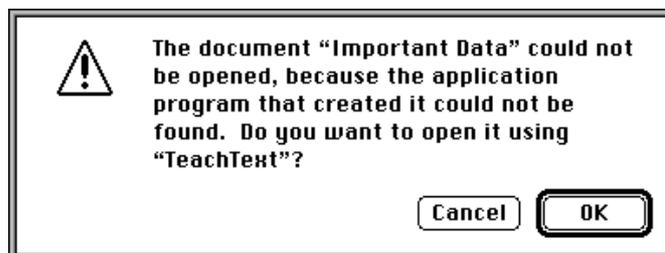
A user can ask the Finder to open a document in several ways, for example, by selecting the document's icon and choosing the Open command in the Finder's File menu or (more typically) by double-clicking the document's icon. If Macintosh Easy Open is not present in the operating environment and the user attempts to open a document created by an application that isn't available, the Finder displays the alert box shown in Figure 7-1.

Figure 7-1 The Finder's application-unavailable alert box



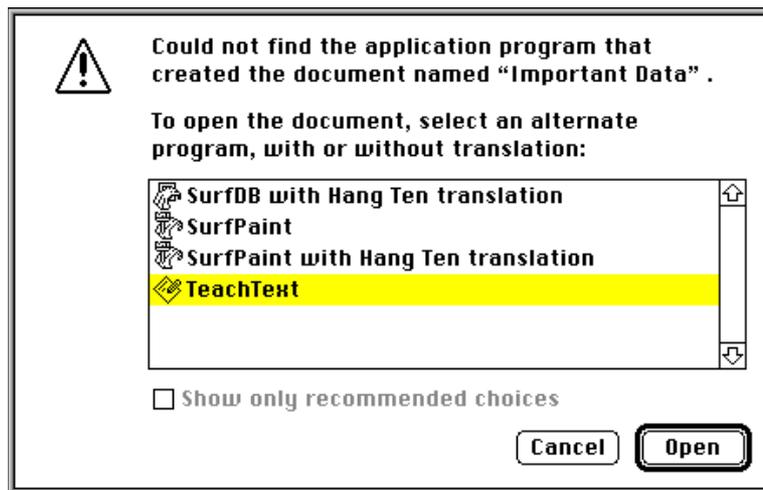
If the document the user wants to open is of type 'TEXT' or 'PICT' and the creator application cannot be found, the Finder instead displays the alert box shown in Figure 7-2, which allows the user to try to open the document using the TeachText application.

Figure 7-2 The application-unavailable alert box for 'TEXT' and 'PICT' documents



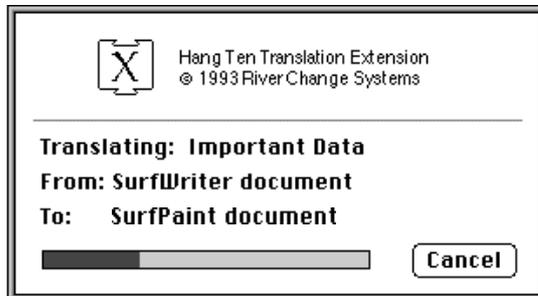
When Macintosh Easy Open is available, it intercedes in the Finder's document-opening process. For example, if the user attempts to open the document "Important Data" (of type 'SURF') created by the SurfWriter application and that application isn't available on the user's system, the Finder displays a dialog box like the one shown in Figure 7-3. This dialog box contains a list of all applications that can open a document of that type.

Figure 7-3 The translation choices dialog box



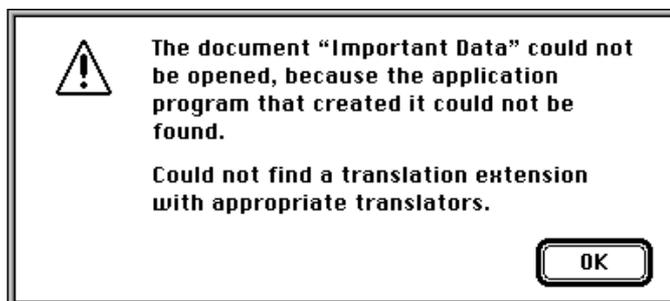
In this dialog box, the user can select a translation path from the document's current format to one that can be opened by some application that is available. In this way, the user can open documents created by missing or unavailable applications.

Macintosh Easy Open lists two kinds of applications in the dialog box shown in Figure 7-3, applications that can open the file natively (that is, in its current format) and those that can open the document only after the document has been translated into some other format. When the user selects an application requiring translation and clicks the Open button, Macintosh Easy Open calls the appropriate translation extension to translate the original document. During the translation, Macintosh Easy Open displays a translation progress dialog box, as shown in Figure 7-4.

Figure 7-4 A translation progress dialog box

The progress dialog box displays the name of the document being translated, its original format, and its target format. The top portion of the dialog box shows an advertisement provided by the particular translation extension that Macintosh Easy Open called to perform the translation. (In this case, the Hang Ten Translation Extension is being used.) It's possible that two or more translation extensions can translate the same original document; if so, they'll all be listed in the translation choices dialog box.

If none of the available translation extensions can translate a particular document, the Finder may present a modified version of the application-unavailable alert box, shown in Figure 7-5.

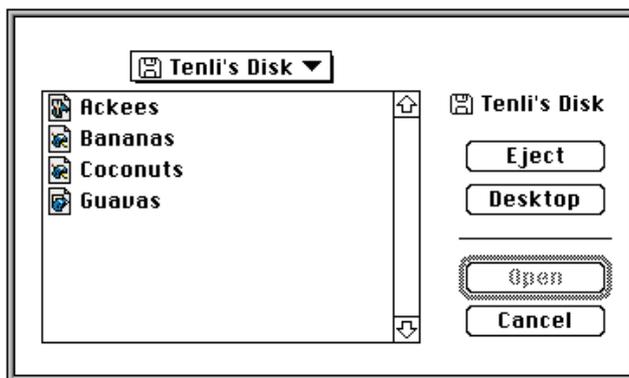
Figure 7-5 The modified application-unavailable alert box

To have another application open a document, the user can drop-launch the document. (To **drop-launch** a document is to drag the document's icon onto the application's icon.) If Macintosh Easy Open knows how to translate the document into a format that can be opened by that application, the Finder highlights the application's icon as the user drags the document icon over it. When the user drop-launches the document, Macintosh Easy Open displays a dialog box that is similar to the translation choices dialog box (see Figure 7-3).

Opening Documents Within an Application

When present in the operating environment, Macintosh Easy Open modifies the Standard File Package so that its file-opening dialog boxes display not only the file types your application can open by itself but also the file types that can be translated into those your application can open. The result is that users can open more documents using your application than they previously could. Figure 7-6 shows the enhanced file-opening dialog box.

Figure 7-6 The enhanced file-opening dialog box



In the case shown in Figure 7-6, the application can open SurfWriter documents without translating them. In addition, Macintosh Easy Open can translate SurfDB and SurfPaint documents to SurfWriter documents; as a result, any SurfDB and SurfPaint documents in the current directory are displayed in the dialog box.

If the user selects a document that your application can open only after some sort of translation, Macintosh Easy Open displays the translation progress dialog box (shown in Figure 7-4) and translates the document into a format that your application recognizes.

Notice in Figure 7-6 that the small, black-and-white generic document icons (of type 'SICN') usually displayed by the Standard File Package have been replaced by small color icons (displayed in this figure in grayscale) that are specific to each type of document. When Macintosh Easy Open is present, the Standard File Package uses small color icons (of type 'ics4' or 'ics8', according to the current bit depth of the display device) to show document types. This allows the user to distinguish more easily between documents of different file types and provides a clue to which documents belong to your application and which belong to some other application but can be opened after translation.

IMPORTANT

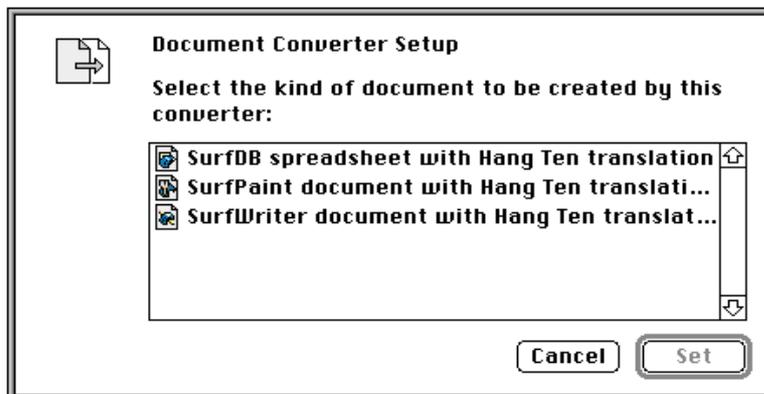
To have the Standard File Package display your application's small color icons in the file-opening dialog box, your application's resource fork should contain the appropriate small color icons (of type 'ics4' or 'ics8'). In addition, if your application uses custom Standard File Package file-opening dialog boxes, your resource fork should contain a dialog color table resource (of type 'dctb') whose resource ID is the same as the resource ID of the dialog box. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information about small color icons; see the chapter "Dialog Manager" in that same book for information about dialog color tables. s

Translating Documents on the Desktop

Macintosh Easy Open includes a tool, called Document Converter, that allows users to convert documents without opening them. This tool is useful if a user wants to convert a number of documents (batch translation) or wants to give the translated documents to other users who don't have either Macintosh Easy Open or the appropriate translation extensions installed on their machines.

To translate documents on the desktop, the user needs to configure the Document Converter tool. When the user opens the Document Converter, it displays the dialog box shown in Figure 7-7.

Figure 7-7 Document Converter configuration dialog box



This dialog box lists target document types, not applications. The user selects a target document type and clicks the Set button to complete the configuration. At that point, the Document Converter application quits and changes its own name to reflect the conversion path of documents subsequently dropped onto it.

Once the Document Converter has been configured, the user can translate documents by dropping them onto the Document Converter icon. The Document Converter creates a new document in the target format and leaves the original document unmodified. The user can also drop a group of documents—or even a folder of documents—onto the Document Converter icon; in these cases, the Document Converter translates all the documents in the group.

Sharing Data Between Applications

Macintosh Easy Open can translate not only documents (data stored in files) but also scraps (data stored in memory) and other data. For instance, when a user copies a selection in one document and pastes the data into a document created by some other application, Macintosh Easy Open steps in, if necessary, to translate the data from its original format (as contained on the Clipboard) to the format of the target document. Because the source and target formats are known, Macintosh Easy Open doesn't need to present the translation choices dialog box (shown in Figure 7-3 on page 7-6). Instead, Macintosh Easy Open proceeds directly with the translation. The only sign that Macintosh Easy Open is at work is the translation progress dialog box.

Data shared in editions might also need to be translated from one format to another. When a user subscribes to an edition (or updates an existing subscriber) and the data in the edition is not already in the format of the subscribing application, Macintosh Easy Open translates the data. Once again, it displays the translation progress dialog box to show the user that it's at work.

Using the Translation Manager

Most applications benefit from the services of Macintosh Easy Open automatically if they use the standard Macintosh system software (such as the Standard File Package, the Edition Manager, and the Scrap Manager) to open files or exchange data with other applications. If the appropriate translators are present on a particular computer, Macintosh Easy Open implicitly translates file and scrap formats into those supported by your application. To facilitate this translation, however, you should

- n Make your application stationery-aware. When Macintosh Easy Open passes your application a translated document, the document's stationery bit is set if your application is stationery-aware. The user should be prompted to save any changes to the translated document under a new name when closing the document.
- n Add a resource of type 'open' to your application. This resource indicates what file types your application can open. See "Declaring the File Types Your Application Can Open" on page 7-13 for complete details.

Translation Manager

- n Add a resource of type 'kind' to your application. This resource allows the Finder to display custom kind strings in its windows. See “Declaring Custom Kind Strings” beginning on page 7-14 for complete details.
- n Add a resource of type 'dctb' to your application if it uses custom Standard File Package file-opening dialog boxes. This resource allows the Standard File Package to display the enhanced small color icons in its dialog boxes. See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on creating resources of type 'dctb'.
- n Avoid using a file filter function as the only method of specifying file types when calling the Standard File Package routines `StandardGetFile` and `CustomGetFile` (or the original `SFGetFile` and `SFPGetFile`). Instead of a file filter function (or in addition to it), you should use the `typeList` parameter to specify file types to list in the file-opening dialog box. Alternatively, you can pass the special value `kUseOpenResourceTypes` in the `numTypes` parameter to have the file types read from your application's 'open' resource. See “Using File-Opening Dialog Boxes” on page 7-15 for more details.
- n Use the Scrap Manager properly:
 - n Put formats on the scrap in order of fidelity.
 - n Get formats from the scrap in the order that your application can best interpret.
 - n Don't call `GetScrap` unless the user has just pasted, because doing so may cause a lengthy translation.
 - n Be able to put the popular scrap formats (such as 'styl') on the scrap.
 - n Don't rely on the `offset` parameter returned by `GetScrap`. It is undefined after implicit translation.
- n Avoid using 'TEXT' as a file type of a document your application creates unless the document contains plain ASCII text intended to be viewed by the user as plain text.
- n Use file types that accurately indicate the format type of the documents your application creates. When you revise your application and make extensive changes to the file format of a document, previous versions of your application will not be able to read the document. In this case, you should assign a different file type to the new format.

If your application does not use the Standard File Package to allow the user to select files to open, you can use the Translation Manager to make your application compatible with Macintosh Easy Open. See “Translating Files Explicitly” on page 7-17 for details.

Checking for the Translation Manager

Macintosh Easy Open and the Translation Manager are not available in all system software versions. You can use the `Gestalt` function to determine whether the services you need are available before calling them. To get information about the Translation Manager, you pass `Gestalt` the selector `gestaltTranslationAttr`.

```
CONST
    gestaltTranslationAttr      = 'xlat';    {Translation Manager}
```

`Gestalt` returns in the `response` parameter a bit field that encodes information about the Translation Manager. Currently only 1 bit is used:

```
CONST
    gestaltTranslationMgrExists = 0;        {TM is present}
```

If the indicated bit (bit 0) is set, the Translation Manager is available and you can safely call the routines it provides. Otherwise, if that bit is clear, the Translation Manager is not available.

As you have seen, Macintosh Easy Open works with the Standard File Package, the Edition Manager, and the Scrap Manager to translate files and scraps implicitly. In most cases, you don't need to know that a file or scrap has been implicitly translated, but in some cases you might need this information. You can use `Gestalt` to determine whether these other system software parts are capable of supporting the capabilities of Macintosh Easy Open. Listing 7-1 lists the translation-specific `Gestalt` selectors and bit numbers of the `response` parameter for the Standard File Package, the Edition Manager, and the Scrap Manager.

Listing 7-1 Translation-specific selectors and response bit for `Gestalt`

```
CONST
    gestaltStandardFileAttr      = 'stdf';    {Standard File Package}
    gestaltStandardFileTranslationAware = 1;
    gestaltStandardFileHasColorIcons = 2;

    gestaltEditionMgrAttr        = 'edtn';    {Edition Manager}
    gestaltEditionMgrTranslationAware = 1;

    gestaltScrapMgrAttr          = 'scra';    {Scrap Manager}
    gestaltScrapMgrTranslationAware = 0;
```

For complete information about the `Gestalt` function, see the chapter “`Gestalt Manager`” in *Inside Macintosh: Operating System Utilities*.

Declaring the File Types Your Application Can Open

In system software versions 7.0 and later, the Finder determines which types of files your application can open by inspecting the resources of type 'FREF' whose resource IDs are listed in your application's bundle (that is, your application's resource of type 'BNDL'). The Finder uses this information to determine which file types can be drop-launched on your application. All file types in the 'FREF' resources listed in your application's bundle, regardless of whether they have associated icons, are considered droppable on your application.

Note

See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of resources of types 'FREF' and 'BNDL'. u

In some cases, however, your application might include 'FREF' resources for file types that you don't want the user to open. For example, your application might use non-document files such as dictionaries and help files. Even though these files should have icons and hence deserve 'FREF' resources, their contents should not be displayed to the user. Similarly, your application might read data from preferences files; this data is intended to be used internally by the application, not opened by the user as a document.

Because the list of file types your application can open may be different from the list of types that have icons, the Translation Manager defines a new resource of type 'open'. The **open resource** declares which file types your application can open as documents (and hence can be dropped onto your application). Listing 7-2 shows a sample resource of type 'open', in Rez input format.

Listing 7-2 A sample resource of type 'open'

```
/*open resource for TeachText*/
resource 'open' (128)
{
    'ttxx', { 'ttro', 'PICT', 'TEXT' }
};
```

An open resource consists of an application signature followed by a list of file types. It indicates that the specified application can open files whose types occur in the list. For example, TeachText can open documents created in its own private format, 'ttro', as well as documents of file type 'PICT' and 'TEXT'. If Macintosh Easy Open is available, the Finder allows the user to drop documents of those types onto the application. In addition, if any translation extensions are installed, all documents that can be translated to one of the specified types can also be dropped on the application. So, if a translation extension exists that can translate documents from type 'SURF' to type 'ttro', the user can drop SurfWriter documents onto TeachText.

You should list file types in your open resource in order of decreasing preference. If the Translation Manager has to choose between multiple file types as the destination file type for a translation, it chooses the file type that occurs earliest in the list.

The open resource is also used by the routine `StandardOpenDialog` to determine which documents should be listed in the file-opening dialog box. See “Using File-Opening Dialog Boxes” on page 7-15 for details on `StandardOpenDialog`.

IMPORTANT

If you use the `StandardOpenDialog` function, the open resource in your application should have resource ID 128. s

Your application might need to determine dynamically which types of files it can open (perhaps by inspecting which filters are available in a certain folder). If so, you cannot list those file types statically in an open resource. Instead, you can write a simple translation extension to generate a list of openable file types at runtime. See “Writing Application Translation Extensions” beginning on page 7-35 for details.

Declaring Custom Kind Strings

A file’s **kind string** is the string displayed in the “Kind” column in a Finder window when a folder’s contents are viewed by name, size, kind, label, or date (that is, by any method other than by icon or small icon). The Finder determines the kind string for a file by taking the name of the application that created it and, in the case of English, appending “document” to that name (for example “SurfWriter document”). If the user does not have the application that created the file, the kind string is simply “document”.

Note

Localized versions of the Finder determine the kind string in other ways. For instance, the Finder may prepend some string (for example, “document de SurfWriter”). u

If the application isn’t available on the computer (the situation in which the user is most likely to want the kind information), the kind string is not particularly helpful. In that case, the displayed string is “document” (or some localization thereof), and the user has no idea which application created it. Moreover, the documents of applications that support many kinds of documents all have the same kind string, even though those documents may be of entirely different kinds (such as word-processing documents, spreadsheet documents, graphics documents, and so forth). It would be better to have the Finder list more information about a document than its creator.

To solve these problems, Macintosh Easy Open allows you to define a custom kind string for each type of file your application creates. You do this by including a **kind resource** (a resource of type ‘kind’) in your application’s resource file. The custom kind strings defined in a kind resource override the algorithm the Finder uses to create kind strings. Listing 7-3 shows a sample kind resource, in Rez input format.

Listing 7-3 A sample resource of type 'kind'

```

/*sample kind resource for SurfSoft Works*/
resource 'kind' (1000)
{
    'WAVE',
    verUS,
    {
        ftApplicationName,      "SurfSoft Works",
        'SWTD',                  "SurfSoft Works text document",
        'SWSS',                  "SurfSoft Works spreadsheet",
        'SWDB',                  "SurfSoft Works database",
    }
}
};

```

A kind resource consists of an application signature, a localization code, and a list of file types and their corresponding kind strings. Each file type is associated with one kind string.

To reduce the number of entries in a kind resource, you can declare your application's name by including an entry having the special file type `ftApplicationName`, as illustrated in Listing 7-3. Then, whenever Macintosh Easy Open encounters a document that belongs to your application but whose file type isn't listed in your application's kind resource, the Finder uses its standard algorithm to generate a kind string in the form "<application name> document".

Note

Because a kind resource contains the application signature, an application's kind resource can be located in some file other than the application's resource file. This feature allows translation extensions to provide kind strings for applications that might not be present on a particular computer. However, the kind resource in an application's resource file overrides any kind resource located elsewhere. \cup

The Finder uses only custom kind strings that have the same localization as the current system itself.

Using File-Opening Dialog Boxes

Macintosh Easy Open works with the Standard File Package to list in the file-opening dialog box all files that your application can open, including those that it can open after they have been translated from their current format to some new format. In general, you don't need to rewrite your application (or even include any additional resources) to receive this service. Macintosh Easy Open provides it automatically when present in the operating environment.

Translation Manager

There are, however, some cases in which Macintosh Easy Open cannot provide this service and that might therefore require you to modify your application if you want to maximize compatibility with Macintosh Easy Open. In particular, if you use a file filter function when calling the Standard File Package routines as the *only* way of determining which files appear in the list of files to open, Macintosh Easy Open cannot safely add any files to that list. This is a problem only when you specify -1 as the value for the `numTypes` parameter in a call to `StandardGetFile` or `CustomGetFile`.

Note

For complete information about file filter functions, see the chapter “Standard File Package” in *Inside Macintosh: Files*. [u](#)

If you use a file filter function when calling the Standard File Package, you should make sure that the list of file types you pass in the `typeList` parameter isn't empty (that is, that the value of the `numTypes` parameter isn't 0). In that case, Macintosh Easy Open is able to expand the list of file types your application can open, regardless of whether you use a file filter function. Macintosh Easy Open inspects the file types passed in the `typeList` parameter and adds to them all file types that can be translated into those file types. In short, you can use a file filter function and benefit from the translation services of Macintosh Easy Open if you specify a non-empty list of file types in the `typeList` parameter.

IMPORTANT

If for some reason you want to prevent Macintosh Easy Open from expanding the list of file types your application can open, simply set the `numTypes` parameter to -1 when calling `StandardGetFile` or `CustomGetFile`. [s](#)

When Macintosh Easy Open is present, you can pass the special value `kUseOpenResourceTypes` in the `numTypes` parameter to have the file types read from your application's 'open' resource.

```
CONST
```

```
    kUseOpenResourceTypes    = -2;
```

When `numTypes` is set to `kUseOpenResourceTypes`, `typeList` is set to `NIL`, and `fileFilter` is set to `NIL`, the Standard File Package displays in the file-opening dialog box all files whose types are listed in your application's 'open' resource (having resource ID 128) as well as all files whose types can be translated into those types.

You can achieve this same result by calling the new Standard File Package function `StandardOpenDialog`.

```
FUNCTION StandardOpenDialog (VAR reply: StandardFileReply): OSErr;
```

The `StandardOpenDialog` function operates exactly like the `StandardGetFile` function, whose parameters `fileFilter`, `numTypes`, and `typeList` are given the values `NIL`, `kUseOpenResourceTypes`, and `NIL`, respectively.

IMPORTANT

The `StandardOpenDialog` function is implemented as glue code and is available in System 6 and later if you link your application with the appropriate object library. [s](#)

Translating Files Explicitly

It's possible that your application might open some document files without the assistance of the Finder or the Standard File Package. For example, your application might execute a script that contains the name of a file to open. Because you're bypassing the system software services that invoke implicit translation, you might need to modify your application to perform **explicit translation** (the conversion of a file or scrap with direct intervention from your application). The Translation Manager provides several routines that you can use to retrieve information about documents and about the document types that an application can open, as well as to translate documents from one format to another.

IMPORTANT

Before calling the routines described in this section, you must make sure that they are available in the current operating environment. See "Checking for the Translation Manager" on page 7-12 for details. [s](#)

You can use the `GetFileTypesThatAppCanNativelyOpen` function to get a list of file types that an application can open by itself. This function takes a volume reference number (where the application resides), an application signature, and a pointer to a buffer to be filled with up to 64 file types. It returns a pointer to the list of the file types that the application can open without translation.

You can use the `ExtendFileTypeList` function to get a list of all file types that the Translation Manager can translate into file types in a given list. This routine takes the original list, the number of file types in it, a pointer to a buffer to be filled with file types, and the maximum number of file types that can be put into the extended list. The `ExtendFileTypeList` function returns a list of all the file types that can be translated into some type in the original list.

You can use the `CanDocBeOpened` function to verify that a specified application can open the document that it is being requested to open. It takes the source document record, the volume reference number of the application that is to open the document, the creator application signature, and the list of file types that the application can open without translation. It returns a document-opening method (`howToOpen`) and document-translation method (`howToTranslate`). The choices for document-opening method are

- n `domCannot`
- n `domNative`
- n `domTranslateFirst`
- n `domWildcard`

Translation Manager

The Translation Manager uses `howToTranslate` to get information on converting the document into a format the application can read. For more information on the `CanDocBeOpened` function, see page 7-40.

You can call the function `TranslateFile` to translate a file from one format to another. It takes the source document record, the destination document record, and the `howToTranslate` parameter returned by `CanDocBeOpened`. In the destination document record, `TranslateFile` returns the name and location of the translated file.

Writing a Translation Extension

A translation extension is a component that works with Macintosh Easy Open to provide data recognition and translation capabilities. Because a translation extension is a component, it must be able to respond to the required request codes sent by the Component Manager. In addition, a translation extension can

- n communicate its translation capability to Macintosh Easy Open
- n identify the formats of specific documents and scraps
- n translate documents and scraps

Translation extensions can identify and translate files, scraps, or both. You specify whether a translation extension handles files or scraps by setting bits in the component flags field in the component resource (see “Creating a Translation Extension” beginning on page 7-19 for details).

IMPORTANT

The information in this section describes how to write translation extensions. If you simply want to make your application compatible with Macintosh Easy Open, see “Using the Translation Manager” beginning on page 7-10. If your application needs to determine dynamically which file types it can open, see “Writing Application Translation Extensions” beginning on page 7-35. s

Macintosh Easy Open and the Translation Manager specify file and scrap formats using the `FileType` and `ScrapType` data types:

TYPE

```
FileType           = OSType;   {file types}
ScrapType          = ResType;  {scrap types}
```

Translation Manager

The `ScrapType` data type describes the format of data in memory. In general, the scrap types used by Macintosh Easy Open are identical to scrap types used by the Scrap Manager. There is, however, one notable exception. Macintosh Easy Open defines a new scrap type, `'stxt'`, to describe styled text. A scrap having format `'stxt'` is formed by appending the text (as contained in a scrap of format `'TEXT'`) to the style information (as contained in a scrap of format `'styl'`). This is necessary to have a single scrap to pass to your scrap translation extension.

The `FileType` data type describes the format of a file. Often, but not always, the format of a file's data can be identified by inspecting the file's type, as maintained in the hierarchical file system catalog file (hereafter called the file's **catalog type**). For purposes of translation, however, it is sometimes necessary to use a more specific identification. For example, some developer might revise an application but retain the existing file type for documents the application creates. This could cause problems for translation extensions, which might be able to translate a specific version of the application's data format but not later ones. Similarly, some applications that create files on Macintosh computers (such as electronic mail programs or disk-mounting utilities) often use standard file types (such as `'TEXT'` or `'BINA'`) as the default new file type. Once again, your translation extension needs more information about the actual format of the data in the file before it can translate it to some other format.

To avoid problems with using a file's catalog type as the only indication of the file's data format, Macintosh Easy Open and the Translation Manager allow you to define a **translation file type**. As just indicated, the catalog file type is often sufficient as the translation file type. However, Macintosh Easy Open always gives your translation extension the opportunity to inspect a particular file to see whether its catalog file type is in fact sufficient for translation purposes. If your extension can identify a more specific format, it should return that information to Macintosh Easy Open. (Ideally, application developers should assign catalog file types that can be used as translation file types.)

The rest of this section describes how to create a file translation extension. You create a scrap translation extension in like fashion, substituting the scrap data types for the corresponding file data types.

Creating a Translation Extension

A translation extension is a component. It contains a number of resources, including icons, strings, pictures, and the standard component resource (a resource of type `'thng'`) required of any Component Manager component. In addition, a translation extension must contain code to handle required request codes passed to it by the Component Manager as well as translation-specific request codes.

Translation Manager

For complete details on components and their structure, see the chapter “Component Manager” in this book. This section provides specific information about translation extensions.

The component resource binds together all the relevant resources contained in a component; its structure is defined by the `ComponentResource` data type.

```
TYPE ComponentResource =
  RECORD
    cd:           ComponentDescription;
    component:    ResourceSpec;
    componentName: ResourceSpec;
    componentInfo: ResourceSpec;
    componentIcon: ResourceSpec;
  END;
```

The `component` field specifies the resource type and resource ID of the component’s executable code. By convention, for translation extensions this resource should be of type `'xlat'`. (You can, however, specify some other resource type if you wish.) The resource ID can be any integer greater than or equal to 128. See the following section for further information about this code resource. The `ResourceSpec` data type has this structure:

```
TYPE ResourceSpec =
  RECORD
    resourceType: ResType;
    resourceID:   Integer;
  END;
```

The `componentName` field specifies the resource type and resource ID of the resource that contains the component’s name. Usually the name is contained in a resource of type `'STR'`. Macintosh Easy Open uses the component’s name in several of the dialog boxes it displays. (For example, in Figure 7-3 on page 7-6, one of the translation extensions has the component name “Hang Ten.”) This string should be as short as possible.

The `componentInfo` field specifies the resource type and resource ID of the resource that contains a description of the component. Usually the description is contained in a resource of type `'STR'`. This information is not currently used by Macintosh Easy Open, but some development tools may use it.

The `componentIcon` field specifies the resource type and resource ID of the resource that contains an icon for the component. Usually the icon is contained in a resource of type `'ICON'`. This icon is not currently used by Macintosh Easy Open, but some development tools may use it.

Translation Manager

Note

The icon displayed in Figure 7-4 on page 7-7 is part of the translation extension's advertisement; it is not supplied by Macintosh Easy Open itself. ^u

The `cd` field of the `ComponentResource` structure is a component description record, which contains additional information about the component. A component description record is defined by the `ComponentDescription` data structure.

```
TYPE ComponentDescription =
    RECORD
        componentType:           LongInt;
        componentSubType:        LongInt;
        componentManufacturer:    LongInt;
        componentFlags:          LongInt;
        componentFlagsMask:      LongInt;
    END;
```

For translation extensions, the `componentType` field must be set to 'xlat'. In addition, the `componentSubType` field must be set to 0 (because there are currently no subtypes of translation extensions). The `componentManufacturer` field identifies the supplier of the component. You should register your component with Apple's Component Registry Group to receive a unique manufacturer code; this code typically corresponds to the signature of your translation extension.

The `componentFlags` field of the component description for a translation extension contains bit flags that encode information about the extension. Currently, you can use this field to specify whether the extension supports file translation routines or scrap translation routines, or both. (See the chapter "Component Manager" in this book for information about the standard flags that you can also specify in the `componentFlags` field.)

```
CONST
    kSupportsFileTranslation    = 1; {file translation extension}
    kSupportsScrapTranslation   = 2; {scrap translation extension}
```

You should set the `componentFlagsMask` field to 0.

IMPORTANT

For compatibility with early versions of the Component Manager, a 'thng' resource should be locked. You can set the other resource attributes in any way you wish. ^s

In addition to the component resource, a translation extension must contain the string and icon resources specified in the component resource (for example, the resource that contains the extension's name). You might also want to include several other resources in the translation extension, including the standard 'BNDL', 'FREF', and 'ICN#' resources used by the Finder and a 'PICT' resource that contains an advertisement or banner to be displayed in the translation progress dialog box. You should also include a 'kind' resource listing kind strings for all the file types your extension can translate from or to; this allows the Finder to display correct kind strings once your extension is installed. Listing 7-4 shows, in Rez input format, the component resource and associated resources of a sample translation extension.

Listing 7-4 Sample resources for a translation extension

```

/*a component resource*/
resource 'thng' (128, locked) {
    'xlat',                /*all translation extensions have this type*/
    0,                    /*subtype is unused*/
    'MYCO',               /*creator signature of extension*/
    kSupportsFileTranslation, /*only file routines are implemented*/
    0,                    /*mask is unused and should be 0*/
    'xlat',128,           /*resource type & ID of translation extension*/
    'STR ',128,           /*resource type and ID of name string*/
    'STR ',129,           /*resource type and ID of information string*/
    'ICON',128            /*resource type and ID of icon*/
};

/*strings*/
resource 'STR ' (128, purgeable) {
    "Hang Ten"
};

resource 'STR ' (129, purgeable) {
    "Hang Ten Translation Extension"
};

/*an icon*/
resource 'ICON' (128, purgeable) {
    $"7FFF FFF0 8000 0008 8000 0008 8000 0008"
    $"8000 0008 8000 0008 8000 0008 8000 0008"
    $"A000 0008 D000 000A 9000 000D 1000 0009"
    $"1000 0001 1000 0001 1000 0001 1000 0001"
    $"1000 0001 1000 0001 1000 0001 1000 0001"
    $"1000 0009 9000 000D D000 000A A000 0008"
    $"8000 0008 8000 0008 8000 0008 8000 0008"
};

```

CHAPTER 7

Translation Manager

```
    $"8000 0008 8000 0008 8000 0008 7FFF FFF0",
};

/*kind strings for document types supported by this extension*/
resource 'kind' (128, purgeable) {
    'SURF',
    verUS,
    {
        ftApplicationName,    "SurfWriter",
        'SURF',                "SurfWriter document",
    }
};

resource 'kind' (129, purgeable) {
    'SPNT',
    verUS,
    {
        ftApplicationName ,    "SurfPaint",
        'SPNT',                "SurfPaint document",
    }
};

resource 'kind' (130, purgeable) {
    'ttxt',
    verUS,
    {
        ftApplicationName ,    "TeachText",
        'ttro',                "TeachText document",
    }
};
```

Your translation extension is contained in a resource file. The creator of the file can be any type you wish, but the type of the file must be 'thng'. If the extension contains a 'BNDL' resource, then the file's bundle bit must be set.

IMPORTANT

The Finder looks for open and kind resources only in files that have their bundle bit set. s

Dispatching to Translation Extension-Defined Routines

As explained in the previous section, the code stored in the translation extension component should be contained in a resource of type 'xlat'. The Component Manager expects that the entry point in this resource is a function having this format:

```
FUNCTION TranslateEntry (VAR params: ComponentParameters;
                       storage: Handle): ComponentResult;
```

The Component Manager calls your extension by passing `TranslateEntry` a request code in the `params.what` field of the components parameter record passed in the `params` parameter; `TranslateEntry` must interpret the request code and possibly dispatch to some other routine in the resource. Your extension must be able to handle the required request codes, defined by these constants:

```
CONST
    kComponentOpenSelect           = -1;
    kComponentCloseSelect         = -2;
    kComponentCanDoSelect         = -3;
    kComponentVersionSelect       = -4;
```

For complete details on required request codes, see the chapter “Component Manager” in this book.

In addition, your extension must be able to respond to translation-specific request codes. Currently, Macintosh Easy Open defines these six request codes:

```
CONST
    kTranslateGetFileTranslationList = 0;
    kTranslateIdentifyFile          = 1;
    kTranslateTranslateFile         = 2;
    kTranslateGetScrapTranslationList = 10;
    kTranslateIdentifyScrap         = 11;
    kTranslateTranslateScrap        = 12;
```

You can respond to these request codes by calling the Component Manager routine `CallComponentFunctionWithStorage`, passing it a pointer to a translation extension-defined routine. Listing 7-5 illustrates how to define a file translation extension entry point routine.

Listing 7-5 Handling Component Manager request codes

```

FUNCTION TranslateEntry (VAR params: ComponentParameters;
                        storage: Handle): ComponentResult;
TYPE
    LongPtr      = ^LongInt;
    LongHandle   = ^LongPtr;
VAR
    mySelf:      ComponentInstance;
    myHandle:    Handle;
    selector:    Integer;
BEGIN
    CASE params.what OF
        kComponentOpenSelect:      {component is opening}
            BEGIN
                mySelf := ComponentInstance(params.params[0]);
                myHandle := NewHandle(SizeOf(ComponentInstance));
                IF myHandle <> NIL THEN
                    BEGIN
                        LongHandle(myHandle)^ := ORD4(mySelf);
                        SetComponentInstanceStorage(mySelf, myHandle);
                        TranslateEntry := noErr;
                    END
                ELSE
                    TranslateEntry := MemError;
            END;
        kComponentCloseSelect:      {component is closing; clean up}
            BEGIN
                IF storage <> NIL THEN
                    DisposeHandle(storage);
                TranslateEntry := noErr;
            END;
        kComponentCanDoSelect:      {return known selectors}
            BEGIN
                selector := Integer((Ptr(params.params)^));
                IF ((kComponentVersionSelect <= selector)
                    AND (selector <= kComponentOpenSelect))
                    OR ((kTranslateGetFileTranslationList <= selector)
                        AND (selector <= kTranslateTranslateFile)) THEN
                    TranslateEntry := 1
                ELSE
                    TranslateEntry := 0;
            END;
    END;

```

Translation Manager

```

kComponentVersionSelect:           {provide version number}
  TranslateEntry := kMyTranslateVersionNumber;
kTranslateGetFileTranslationList:   {give file translation list}
  TranslateEntry := CallComponentFunctionWithStorage
                    (Handle(storage^^), params,
                    ComponentFunction(@DoGetFileTranslationList));

kTranslateIdentifyFile:             {identify a file}
  TranslateEntry := CallComponentFunctionWithStorage
                    (Handle(storage^^), params,
                    ComponentFunction(@DoIdentifyFile));
kTranslateTranslateFile:           {translate a file}
  TranslateEntry := CallComponentFunctionWithStorage
                    (Handle(storage^^), params,
                    ComponentFunction(@DoTranslateFile));
OTHERWISE                          {unrecognized selector}
  TranslateEntry := badComponentSelector;
END; {CASE}
END;

```

As you can see, the `TranslateEntry` function defined in Listing 7-5 simply inspects the `params.what` field to determine which request code to handle. For translation-specific request codes, it dispatches to the appropriate function in the translation extension. See the following three sections for more details on handling translation-specific request codes.

Your extension can be dynamically loaded or unloaded at any time. When Macintosh Easy Open first discovers the extension, it loads it into memory and then asks it to return a list specifying which file or scrap types it can translate into which other types. Your extension is also called during a translation to identify files or scraps and, if necessary, to translate them.

Macintosh Easy Open loads your extension into a subheap of some existing heap. In all likelihood, your extension is loaded into either the system heap or temporary memory. In some cases, however, your extension might be loaded into an application's heap. Your extension is guaranteed 32 KB of available heap space. You should do all allocation in that heap using normal Memory Manager routines. Any memory leaks are reclaimed when your routine returns and the heap is destroyed.

Translation Manager

There is no support for using global variables in the dispatcher defined in Listing 7-5. In general, the routines you need to implement are separate and self-contained, and so you shouldn't need to use global variables. You can, however, have your dispatcher set up an A5 world that contains global variables.

S WARNING

If you use PC-relative global variables (that is, data addressed relative to the program counter), be warned that the Component Manager may purge and reload your code resource. Therefore, all PC-relative global variables must be preinitialized at compile time (not at load time). s

If you need to access resources that are stored in your translation extension, you should use `OpenComponentResFile` and `CloseComponentResFile`. The open routine requires the `ComponentInstance` parameter supplied to your routine. See Listing 7-7 on page 7-33 for an example. You should not call the Resource Manager routines such as `OpenResFile` or `CloseResFile`.

S WARNING

Do not leave any resource files open when your translation extension exits. Their maps will be left in the subheap when the subheap is freed, causing the Resource Manager to crash. s

The following sections show how to respond to the

`kTranslateGetFileTranslationList`, `kTranslateIdentifyFile`, and `kTranslateTranslateFile` request codes by defining the three file translation extension functions `DoGetFileTranslationList`, `DoIdentifyFile`, and `DoTranslateFile`. You would handle scrap translation in a similar manner.

Creating a Translation List

Your translation extension must be able to inform Macintosh Easy Open of its translation capabilities in response to the `kTranslateGetFileTranslationList` request code. To do this, you can define a `DoGetFileTranslationList` function in which you fill in a **file translation list**, defined by a `FileTranslationList` record. From the file translation list you return, Macintosh Easy Open learns which types of files your extension can translate into which other types. On the basis of this information, it may later call your extension to identify a particular document and possibly to translate it.

The `FileTranslationList` record has this structure:

```

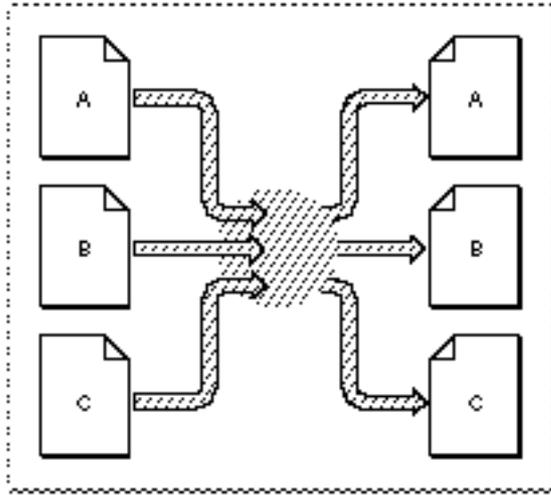
TYPE FileTranslationList =
  RECORD
    modDate:           LongInt;
    groupCount:        LongInt;
    {group1SrcCount:    LongInt;}
    {group1SrcEntrySize: LongInt;}
    {group1SrcTypes:    ARRAY[1..group1SrcCount] OF FileTypeSpec;}
    {group1DstCount:    LongInt;}
    {group1DstEntrySize: LongInt;}
    {group1DstTypes:    ARRAY[1..group1DstCount] OF FileTypeSpec;}
    {repeat above six lines for a total of groupCount times}
  END;

```

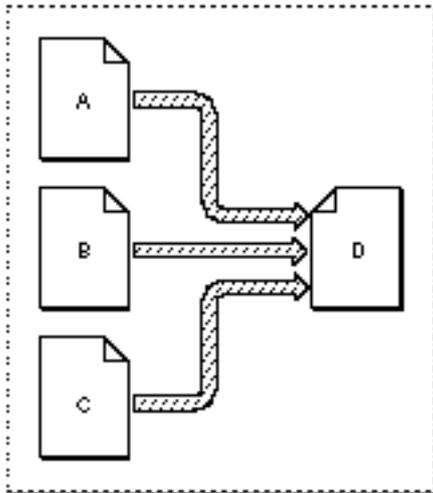
This record contains a modification date and a count of the number of translation groups that follow. Each **translation group** in the file translation list specifies a collection of file types from which the extension can translate (the `group1SrcTypes` field) and a collection of file types into which the extension can translate (the `group1DstTypes` field). Within a translation group, your extension must be able to translate any of the source types into any of the destination types.

You might have different translation groups corresponding to different categories of documents. For instance, you can place word-processing documents in one group, spreadsheet documents in another, and so on. You are, however, free to group file types in whatever manner you like.

In most cases, `group1SrcCount` and `group1DstCount` will each be greater than 1, because most translators operate by translating through a particular data model. In these cases, it's also quite likely that the source and destination file types overlap or even coincide. Figure 7-8 illustrates a typical translation group.

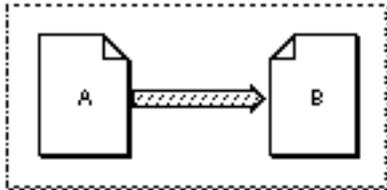
Figure 7-8 A translation group with multiple source and destination types

Similarly, you might write a translation extension that converts other file types into your own proprietary document format. In this case, you would have multiple source document types but only one destination type (`group1DstCount` equal to 1), as illustrated in Figure 7-9.

Figure 7-9 A translation group with a single destination type

It's possible, however, to have both `group1SrcCount` and `group1DstCount` equal to 1. This kind of translation is known as **point-to-point translation**. Figure 7-10 illustrates point-to-point translation.

Figure 7-10 Point-to-point translation



Note

The number of translation groups you can specify in a file translation list is limited by memory considerations only. u

Within any particular group of file types, you specify a particular document format using a **file type specification**, defined by the `FileTypeSpec` data type.

```
TYPE FileTypeSpec =
  RECORD
    format:      FileType;
    hint:        LongInt;
    flags:       TranslationAttributes;
    catInfoType: OSType;
    catInfoCreator: OSType;
  END;
```

A file type specification includes the file type, a hint reserved for use by your extension, a flags field, and the original file type and creator. See “File Type Specifications” beginning on page 7-46 for complete details on these fields.

Listing 7-6 shows a simple routine that creates a file translation list. The translation extension containing this routine can translate both SurfWriter and SurfPaint documents to a format understood by TeachText.

Listing 7-6 Creating a file translation list

```
FUNCTION DoGetFileTranslationList
  (self: ComponentInstance;
   translationList: FileTranslationListHandle)
  : ComponentResult;
```

Translation Manager

```

TYPE
  MyList =
    RECORD
      modDate:           LongInt;
      groupCount:        LongInt;
      group1SrcCount:    LongInt;
      group1SrcEntrySize: LongInt;
      group1SrcTypes:    ARRAY[1..2] OF FileTypeSpec;
      group1DstCount:    LongInt;
      group1DstEntrySize: LongInt;
      group1DstTypes:    ARRAY[1..1] OF FileTypeSpec;
    END;
  MyListPtr      = ^MyList;
  MyListHandle   = ^MyListPtr;
VAR
  myErr:         OSerr;
  myPtr:         MyListPtr;
CONST
  kStamp        = $A74520A8;      {date of original list creation}
BEGIN
  myErr := noErr;
  IF translationList^^.modDate <> kStamp THEN
  BEGIN
    {resize the handle so there's enough room}
    SetHandleSize(Handle(translationList), SizeOf(MyList));
    myErr := MemError;
    IF myErr = noErr THEN
    BEGIN
      myPtr := MyListHandle(translationList)^;
      WITH myPtr^ DO
      BEGIN
        modDate := kStamp;           {set creation date}
        groupCount := 1;             {only 1 translation group}
        group1SrcCount := 2;         {source side has two types}
        group1SrcEntrySize := SizeOf(FileTypeSpec);
        WITH group1SrcTypes[1] DO
        BEGIN
          format := 'SURF';          {SurfWriter document format}
          hint := 0;                 {no hint}
          flags := 0;                {no flags}
          catInfoType := 'SURF';     {catalog type}
          catInfoCreator := 'TONY';  {catalog creator}
        END;
      END;
    END;
  END;

```

Translation Manager

```

WITH group1SrcTypes[2] DO
BEGIN
    format := 'SPNT';           {SurfPaint document format}
    hint := 0;                  {no hint}
    flags := 0;                 {no flags}
    catInfoType := 'SPNT';      {catalog type}
    catInfoCreator := 'TONY';   {catalog creator}
END;
group1DstCount := 1;           {destination side has one type}
group1DstEntrySize := SizeOf(FileTypeSpec);
WITH group1DstTypes[1] DO
BEGIN
    format := 'ttro';           {TeachText document format}
    hint := 0;                  {no hint}
    flags := taDstDocNeedsResourceFork;
                                {TeachText documents need a }
                                { resource fork (for pictures)}
    catInfoType := 'ttro';      {catalog type}
    catInfoCreator := 'ttxt';   {catalog creator}
END;
END; {WITH myPtr^}
END; {IF}
END; {IF}
DoGetFileTranslationList := myErr;
END;

```

Because the list of file types that this extension can translate never changes, `DoGetFileTranslationList` fills out a file translation list the first time Macintosh Easy Open calls it; every other time it is called, `DoGetFileTranslationList` simply passes back the list it was passed.

In all likelihood, your translation extension will rely on external translators to perform the actual translation of files or scraps. If so, it's also likely that the user will be able to add and remove translators used by your extension—possibly by moving translators into or out of some specific folder. In that case, your `DoGetFileTranslationList` function could read the modification date of that folder and compare with a value you previously put in the `modDate` field to determine whether to regenerate the translation list.

Identifying Files

Once Macintosh Easy Open knows the types of files from and to which your extension can translate, it might call your extension to determine whether your extension can translate a particular file. This further check is necessary because some documents might have file types that are not specific enough for translation purposes. For example, a

document imported from a different operating system might have a file type of 'TEXT'. Your translation extension might be able to determine, however, that the file actually contains SurfWriterPC data and hence deserves special format conversion treatment.

When your translation extension is called with the `kTranslateIdentifyFile` request code, your extension should identify the particular document. The `TranslateEntry` extension (shown in Listing 7-5 on page 7-25) dispatches to its `DoIdentifyFile` function when it receives this request code. Listing 7-7 shows the skeleton of a `DoIdentifyFile` function.

Listing 7-7 Identifying file types

```
FUNCTION DoIdentifyFile (self: ComponentInstance; theDoc: FSSpec;
                        VAR docKind: FileType): ComponentResult;
VAR
    isKnown: Boolean; {indicates whether this extension can identify the file}
BEGIN
    {call an extension-defined routine to do the real work}
    isKnown := MyIdentifyDocument(theDoc, docKind);
    IF isKnown THEN
        DoIdentifyFile := noErr
    ELSE
        DoIdentifyFile := noTypeErr;
END;
```

Some documents can be identified simply by inspecting their file type and creator. Other documents (in particular, those of type 'TEXT') might require opening the files and examining their contents to determine whether they can be translated by your extension. If your extension cannot recognize the document type, `DoIdentifyFile` should return `noTypeErr`. Otherwise, `DoIdentifyFile` should return `noErr`, and the `docKind` parameter should be set to the recognized file type.

Note

Your `DoIdentifyFile` function should not return 'TEXT' as a file type unless it's certain that the document consists of plain, unformatted ASCII text. u

You should be aware that even if your extension identifies a particular document as one that it can translate, Macintosh Easy Open might not in fact call your extension to do the translation.

Translating Files

If your translation extension identifies a document as one that it can translate and the user chooses to use your translation extension, your extension is called with the `kTranslateTranslateFile` request code to translate the document. The

Translation Manager

TranslateEntry extension (shown in Listing 7-5 on page 7-25) dispatches to its DoTranslateFile function when it receives this request code. Listing 7-8 shows the skeleton of a DoTranslateFile function.

Listing 7-8 Translating a document

```

FUNCTION DoTranslateFile (self: ComponentInstance;
                        refNum: TranslationRefNum;
                        srcDoc: FSSpec;
                        srcType: FileType;
                        srcTypeHint: LongInt;
                        dstDoc: FSSpec;
                        dstType: FileType;
                        dstTypeHint: LongInt): ComponentResult;

VAR
    myAdvert:    Handle;
    myResFile:   Integer;
    myResult:    OSErr;

CONST
    rProgressAdvertisementResID      = 150;

BEGIN
    myResFile := OpenComponentResFile(Component(self));
    IF myResFile <> -1 THEN
        BEGIN
            {get advertisement}
            myAdvert := Get1Resource('PICT', rProgressAdvertisementResID);
            DetachResource(myAdvert);
            {display progress dialog box and show advertisement}
            myResult := SetTranslationAdvertisement(refNum, PicHandle(myAdvert));
            myResult := CloseComponentResFile(myResFile);
        END;
        {now call your routine to translate the file}
        DoTranslateFile := MyDoTranslation
                            (refNum, srcDoc, srcType, dstDoc, dstType);
        DisposeHandle(myAdvert);
    END;
END;

```

By the time the DoTranslateFile routine is called, the file specified by the dstDoc parameter already exists. The destination file has a data fork; it also has a resource fork if the flags field in the appropriate destination file type specification (in your extension's file translation list) has the taDstDocNeedsResourceFork bit set. Your extension should open the destination file and fill it with the translated data.

In Listing 7-8, the `DoTranslateFile` function calls the `SetTranslationAdvertisement` function to install an advertisement in the progress dialog box. The routine that does the actual data translation (`MyDoTranslation`) should periodically call `UpdateTranslationProgress` to update the progress bar in the dialog box.

If an error occurs during the translation, you should make sure to close any files you might have opened (for instance, the destination file's data fork and resource fork), do any other necessary cleaning up, and then return a nonzero result code through your component selector dispatcher. When Macintosh Easy Open receives a nonzero result code, it automatically deletes the destination file.

Writing Application Translation Extensions

Most applications can open only a certain number of file types and can therefore declare those openable file types by including an open resource in their resource forks. (See "Declaring the File Types Your Application Can Open" on page 7-13 for details about the open resource.) Some applications, however, need to determine dynamically which files they can open (perhaps because those applications already contain data-conversion capabilities using external filters). For these applications, the open resource alone is inadequate to specify which kinds of files they can open.

A simple way to generate dynamically a list of your application's openable file types is to provide an **application translation extension**, a translation extension that can create a list of file types and identify files, but which performs no actual translation. Essentially, the application translation extension exists solely to generate the dynamic list of file types your application can open. The source list in the file translation list that your extension returns to Macintosh Easy Open should contain a file type specification for each of those types; for the destination list of types, the file translation list should contain a single file type specification whose `format` field contains some arbitrary and otherwise unused file type. Suppose this destination file type is `'VOID'`.

The open resource in your application should then consist of a static list containing at least the value in the `format` field of the sole destination file type specification in the file translation list (that is, `'VOID'`). The net effect, as far as Macintosh Easy Open is concerned, is that your application can open documents of type `'VOID'` and that a translation extension exists that can translate some other file types into type `'VOID'`. As a result, the types in that list—which was generated dynamically—are now considered openable by your application.

Of course, in the situation imagined here, you don't want the application translation extension to do any actual data conversion. You indicate this by setting the `taDstIsAppTranslation` bit in the `flags` field of the destination file type specification. If this bit is set, Macintosh Easy Open gives the source document directly to your application without translation. No destination document is created.

Note

In the translation choices dialog box (illustrated in Figure 7-3 on page 7-6), a file type whose file type specification has the `taDstIsAppTranslation` bit set is listed by the application name only; the name of the application translation extension is not listed. u

Translation Manager Reference

This section describes the routines and resources that are specific to the Translation Manager. To take full advantage of the implicit translation capabilities of Macintosh Easy Open, you need to include appropriate resources in your application's resource fork. (See "Resources" beginning on page 7-43 for information on the open and kind resources.) In addition, you might need to call Translation Manager routines if your application doesn't use the Standard File Package or if it needs information about an application's translation capabilities.

IMPORTANT

The routines described in this section are intended for use by applications that bypass the Standard File Package or that need information about some application's ability to translate documents. Most applications don't need to use these routines. s

See "Translation Extension Reference" beginning on page 7-46 for information about data structures and routines you can use to write a translation extension.

IMPORTANT

The Translation Manager is not available in all operating environments. You should call the `Gestalt` function to ensure that it is available before calling any of its routines. See "Checking for the Translation Manager" on page 7-12 for details on calling `Gestalt`. s

Translation Manager Routines

The Translation Manager provides a number of routines that your application can call to get information about the documents and document types an application can open and to translate files and scraps. Normally, you need to use these routines only if your application doesn't use the Standard File Package to ask the user for names and locations of files to open, or if your application has other special needs.

Getting Translation Information

The Translation Manager provides several routines that you can use to get or set information about the file types that an application can open.

GetFileTypesThatAppCanNativelyOpen

You can use the `GetFileTypesThatAppCanNativelyOpen` function to obtain a list of file types that an application can open by itself.

```
FUNCTION GetFileTypesThatAppCanNativelyOpen
    (appVRefNumHint: Integer; appSignature: OSType;
     VAR nativeTypes: TypesBlock): OSErr;
```

`appVRefNumHint`

The volume reference number of volume containing the application. The search for the specified application begins on this volume; if the application isn't found there, the search continues to other mounted volumes.

`appSignature`

The signature of the application.

`nativeTypes`

On exit, a zero-terminated file types that the application can open without translation.

DESCRIPTION

The `GetFileTypesThatAppCanNativelyOpen` function returns, through the `nativeTypes` parameter, a list of all the file types that can be opened by the application having the signature `appSignature`. If `GetFileTypesThatAppCanNativelyOpen` returns successfully, the `nativeTypes` parameter contains a list of up to 64 file types. The structure of the list is defined by the `TypesBlock` data type.

TYPE

```
TypesBlock    = ARRAY[0..63] OF FileType;
TypesBlockPtr = ^TypesBlock;
```

If fewer than 64 types are returned, the end of the list is indicated by an entry whose value is 0.

SPECIAL CONSIDERATIONS

The `GetFileTypesThatAppCanNativelyOpen` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `GetFileTypesThatAppCanNativelyOpen` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `GetFileTypesThatAppCanNativelyOpen` are

Trap macro	Selector
<code>_TranslationDispatch</code>	<code>\$001C</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>wrgVolTypErr</code>	-123	Volume does not support Desktop Manager
<code>afpItemNotFound</code>	-5012	Information not found

ExtendFileTypeList

You can use the `ExtendFileTypeList` function to create a list of file types that can be translated into a type in a given list. The Standard File Package calls this function internally; your application probably won't need to use it.

```
FUNCTION ExtendFileTypeList (originalTypeList: FileTypePtr;
                             numberOriginalTypes: Integer;
                             extendedTypeList: FileTypePtr;
                             VAR numberExtendedTypes: Integer)
                             : OSErr;
```

`originalTypeList`

A pointer to a list of file types.

`numberOriginalTypes`

The number of file types in `originalTypeList`.

Translation Manager

`extendedTypeList`

On exit, a pointer to a list of file types that can be translated into the types in `originalTypeList`.

`numberExtendedTypes`

On entry, the maximum number of file types that can be put into the `extendedTypeList` parameter. On exit, the actual number of file types put into the extended type list.

DESCRIPTION

The `ExtendFileTypeList` function takes the set of types in the `originalTypeList` parameter and returns (in the `extendedTypeList` parameter) a list of types that can be translated into those types. The `extendedTypeList` parameter is of type `FileTypePtr`, which is a pointer to a file type.

TYPE

```
FileTypePtr = ^FileType;
```

Note that the number of types specified in the parameters `numberOriginalTypes` and `numberExtendedTypes` is limited only by available memory.

SPECIAL CONSIDERATIONS

The `ExtendFileTypeList` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `ExtendFileTypeList` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ExtendFileTypeList` procedure are

Trap macro	Selector
<code>_TranslationDispatch</code>	<code>\$0009</code>

RESULT CODE

<code>noErr</code>	0	No error
--------------------	---	----------

CanDocBeOpened

You can use the `CanDocBeOpened` function to determine whether a specified application can open a particular document.

```
FUNCTION CanDocBeOpened
    (targetDocument: FSSpec;
     appVRefNumHint: Integer;
     appSignature: OSType;
     nativeTypes: TypesBlockPtr;
     onlyNative: Boolean;
     VAR howToOpen: DocOpenMethod;
     VAR howToTranslate: FileTranslationSpec)
    : OSErr;
```

`targetDocument`

The document to check.

`appVRefNumHint`

The volume reference number of the volume containing the application. The search for the specified application begins on this volume; if the application isn't found there, the search continues to other mounted volumes.

`appSignature`

The signature of the application.

`nativeTypes`

A zero-terminated list of file types that the application can open without translation; if this parameter contains `NIL`, the default list of file types returned by the `GetFileTypesThatAppCanNativelyOpen` function is used.

`onlyNative`

If `TRUE`, determine only whether the application can open the document without translation; otherwise, determine whether the application can open the document after translation.

`howToOpen`

On exit, the method of opening the document. This field contains a meaningful value only if `CanDocBeOpened` returns `noErr` (indicating that the specified document can be opened).

`howToTranslate`

On exit, a buffer of information (in a private format) indicating how to translate the document.

DESCRIPTION

The `CanDocBeOpened` function determines whether a document can be opened by a particular application. If the application can open the document, `CanDocBeOpened` returns the result code `noErr` and sets the `howToOpen` parameter to a constant that indicates the method that the application would use to open the document. The `howToOpen` parameter contains a document-opening method:

```
TYPE DocOpenMethod =
    (domCannot, domNative, domTranslateFirst, domWildcard);
```

The `domCannot` constant indicates that the application cannot open the document. The `domNative` constant indicates that the application can open the document natively. The `domTranslateFirst` constant indicates that the application can open the document only after it's been translated. The `domWildcard` constant indicates that the application has the file type '****' in its list of the file types that it can open and hence can open any type of document.

If the document needs to be translated before it can be opened (as indicated by the `domTranslateFirst` method), `CanDocBeOpened` returns in the `howToTranslate` parameter a buffer of information indicating how to translate the document. The format of this information is private.

```
TYPE
    FileTranslationSpec = ARRAY[1..12] OF LongInt;
```

You pass the information returned in the `howToTranslate` parameter to the `TranslateFile` function.

SPECIAL CONSIDERATIONS

A preference must have already been set (using the Document Converter tool) on how to open the document.

The `CanDocBeOpened` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `CanDocBeOpened` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `CanDocBeOpened` procedure are

Trap macro	Selector
<code>_TranslationDispatch</code>	<code>\$001E</code>

Translation Manager

RESULT CODES

noErr	0	No error
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename
fnfErr	-43	File not found
paramErr	-50	Parameter error
extFSErr	-58	External file system
dirNFErr	-120	Directory not found or incomplete pathname
badTranslationSpecErr	-3031	Translation path is invalid
noPrefAppErr	-3032	No translation preference available
afpItemNotFound	-5012	Information not found

Translating Files

The Translation Manager provides a routine that you can use to translate files.

TranslateFile

You can use `TranslateFile` to translate a document from one format to another.

```
FUNCTION TranslateFile (sourceDocument: FSSpec;
                      destinationDocument: FSSpec;
                      howToTranslate: FileTranslationSpec)
                      : OSErr;
```

`sourceDocument`

The document to translate.

`destinationDocument`

The file to put the translated document into.

`howToTranslate`

A buffer of information indicating how to translate the document.

DESCRIPTION

The `TranslateFile` function reads the file specified by the `sourceDocument` parameter and translates it into another format. You specify in the `destinationDocument` parameter the name and location of a file to contain the translated data. Note that your application only specifies the name and location for the file; `TranslateFile` creates the file and puts the translated data into it. The destination file must not exist before you call `TranslateFile`.

Translation Manager

The translation is performed according to the information provided in the `howToTranslate` parameter. Usually, you'll get that information by calling `CanDocBeOpened`.

SPECIAL CONSIDERATIONS

The `TranslateFile` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `TranslateFile` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `TranslateFile` procedure are

Trap macro	Selector
<code>_TranslationDispatch</code>	<code>\$000C</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>dirFulErr</code>	-33	Directory full
<code>dskFulErr</code>	-34	Disk full
<code>nvsErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename
<code>tmfoErr</code>	-42	Too many files open
<code>fnfErr</code>	-43	File not found
<code>wPrErr</code>	-44	Disk is write protected
<code>fLckdErr</code>	-45	File is locked
<code>vLckdErr</code>	-46	Volume is locked
<code>dupFNerr</code>	-48	Duplicate filename (rename)
<code>opWrErr</code>	-49	File already open with write permission
<code>extFSErr</code>	-58	External file system
<code>dirNFErr</code>	-120	Directory not found
<code>userCanceledErr</code>	-128	User canceled
<code>badTranslationSpecErr</code>	-3031	<code>howToTranslate</code> is invalid

Resources

This section describes the resources used by the Translation Manager.

n The 'open' resource indicates which kinds of documents your application can open.

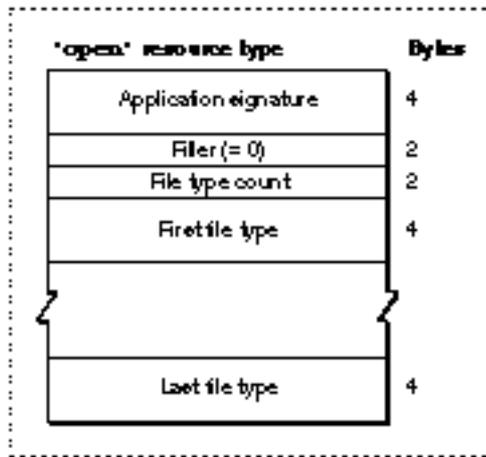
n The 'kind' resource defines a custom kind string for your application's documents.

Information from these resources is stored in a volume's desktop database. Any changes made to an application's open or kind resources won't appear until the desktop is rebuilt.

The Open Resource

To make your application compatible with the Translation Manager, you should add an open resource to your application's resource file. This resource, of type 'open', indicates which types of files your application can open. Figure 7-11 shows the format of a compiled open resource.

Figure 7-11 Structure of a compiled open ('open') resource



An open resource consists of your application's signature and a list of file types. The Finder allows the user to launch your application by dropping documents of any of those types on your application's icon. In addition, if any translation extensions are installed, all documents that can be translated into one of those file types can also be dropped onto your application's icon. Your application's open resource having resource ID 128 is used by the Standard File Package routine `StandardOpenDialog` to determine the file types displayed in the standard file-opening dialog box.

IMPORTANT

The file types in the open resource should be ordered by preference. If the Translation Manager has to choose between multiple file types as the destination file type for a translation, it chooses the file type that occurs earliest in the list.

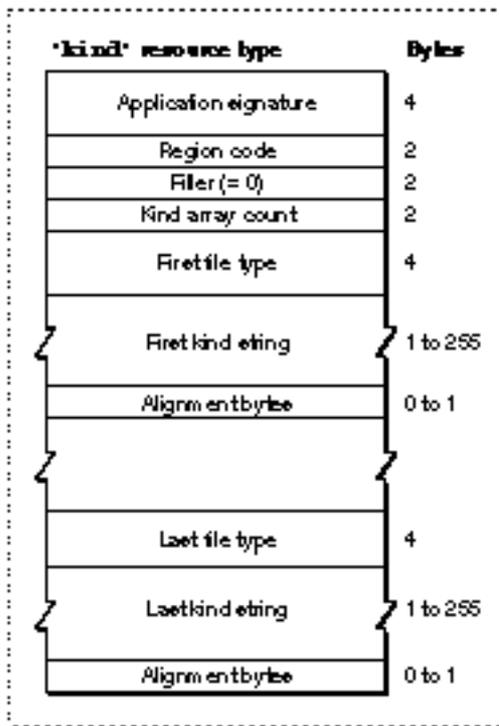
Because your application's signature is included in the open resource, the resource can be in some file other than the application's resource fork. However, an open resource located in an application's resource fork overrides any open resource for that application located elsewhere. It also overrides the openable file types as listed in the application's 'FREF' resource.

See Listing 7-2 on page 7-13 for a sample open resource.

The Kind Resource

You should add a kind resource to your application's resource file. This resource, of type 'kind', specifies custom kind strings, which override the Finder's normal algorithm for generating kind strings. Figure 7-12 shows the format of a compiled kind resource.

Figure 7-12 Structure of a compiled kind ('kind') resource



A kind resource consists of your application's signature and a list of file types and their associated custom kind strings. The Finder displays a document's kind string when a folder's contents are viewed by name, size, kind, label, or date (that is, by any method other than by icon or small icon).

Because your application's signature is included in the kind resource, the resource can be located in some file other than the application's resource fork. However, a kind resource located in an application's resource fork overrides any kind resource for that application located elsewhere.

A kind resource contains a region code, which specifies the region code of the kind strings contained in the resource. The Finder uses only custom kind strings that have the same region code as the current system itself.

In the list of file types and associated kind strings, you can use the special file type `ftApplicationName` to specify the name of your application. Whenever Macintosh Easy Open encounters a document that belongs to your application but whose file type

isn't listed in your application's kind resource, the Finder uses its standard algorithm to generate a kind string in the form "<application name> document".

Note

See Listing 7-3 on page 7-15 for a sample kind resource. u

Translation Extension Reference

This section describes the data structures and routines you can use to write a translation extension. It also describes the routines that your translation extension has to contain.

See "Translation Manager Reference" beginning on page 7-36 for a description of the routines and resources that are specific to the Translation Manager.

Translation Extension Data Structures

This section describes the data structures you'll need to use when writing a translation extension.

File Type Specifications

You use file type specifications to describe document formats in a file translation list. (See the next section for a description of file translation lists.) The interpretation of some of the fields of a file type specification depends on whether the specification occurs in the list of source document types or in the list of destination document types. A file type specification is defined by the `FileTypeSpec` data structure.

```

TYPE FileTypeSpec =
    RECORD
        format:           FileType;
        hint:             LongInt;
        flags:            TranslationAttributes;
        catInfoType:      OSType;
        catInfoCreator:   OSType;
    END;

```

Translation Manager

Field descriptions

<code>format</code>	The translation file type of the document. Macintosh Easy Open uses this field as the canonical way to describe the format of a file for translation purposes.
<code>hint</code>	A 4-byte value reserved for use by your translation extension.
<code>flags</code>	A 4-byte value consisting of bit flags that specify how to control the translation. This field is used only for destination file types; you should set it to 0 for all source file type specifications. Currently 2 bits are defined; all other bits should be cleared to 0:

```
CONST
```

```
    taDstDocNeedsResourceFork    = 1;
    taDstIsAppTranslation        = 2;
```

Before Macintosh Easy Open sends your translation extension the `kTranslateTranslateFile` request code, it has already created the destination file's data fork. The bit specified by the constant `taDstDocNeedsResourceFork` should be set if the translated document also needs a resource fork.

The bit specified by the constant `taDstIsAppTranslation` should be set if your extension doesn't actually perform the translation because an associated application can already translate the specified file type into the desired format. See "Writing Application Translation Extensions" on page 7-35 for more details.

<code>catInfoType</code>	The type of the file as contained in the volume's catalog file.
--------------------------	---

<code>catInfoCreator</code>	The creator of the file as contained in the volume's catalog file.
-----------------------------	--

In file type specifications occurring in the list of source document types in a file translation list, Macintosh Easy Open uses the `format` and `catInfoCreator` fields to determine the kind string displayed in the "From" format specification of the translation progress dialog box (see Figure 7-4 on page 7-7).

In file type specifications occurring in the list of destination document types in a file translation list, Macintosh Easy Open uses the `format` and `catInfoCreator` fields to determine the kind string displayed in the "To" format specification in the translation progress dialog box (see Figure 7-4 on page 7-7). The `format` and `catInfoCreator` fields are also used to get the information displayed in the Document Converter dialog box (Figure 7-7 on page 7-9). However, Macintosh Easy Open uses the `catInfoType` and `catInfoCreator` fields to set the catalog type and creator of the destination file.

Note

See page 7-19 for a discussion of why the translation file type described in the `format` field may differ from the catalog type described in the `catInfoType` field. u

File Translation Lists

You use the `FileTranslationList` data structure to describe which file formats your extension can translate into which other file formats. The Translation Manager uses the file translation list that it gets from each translation system to create a master database of format translations it can direct.

```

TYPE FileTranslationList =
  RECORD
    modDate:           LongInt;
    groupCount:        LongInt;
    {group1SrcCount:    LongInt;}
    {group1SrcEntrySize: LongInt;}
    {group1SrcTypes:   ARRAY[1..group1SrcCount] OF FileTypeSpec;}
    {group1DstCount:   LongInt;}
    {group1DstEntrySize: LongInt;}
    {group1DstTypes:   ARRAY[1..group1DstCount] OF FileTypeSpec;}
    {repeat above six lines for a total of groupCount times}
  END;
FileTranslationListPtr = ^FileTranslationList;
FileTranslationListHandle = ^FileTranslationListPtr;

```

A file translation list consists of a field indicating the modification date of the list and a count of the number of groups that follow those two fields. The size of the translation list prepared by an extension is variable, depending upon the number of groups, the file specification record size, and the number of file types that the extension knows about.

Field descriptions

<code>modDate</code>	The creation date of the file translation list. If your extension uses external translators, you might set this field to the modification date of a folder containing those translators.
<code>groupCount</code>	The number of translation groups that follow.
<code>group1SrcCount</code>	The number of file types that the extension can read in a group.
<code>group1SrcEntrySize</code>	The size of the file specification records in the array that follows this field. In general, you can set this field to <code>SizeOf(FileTypeSpec)</code> .
<code>group1SrcTypes</code>	An array of file specification records. You should include a file specification record in this array for each file type that your extension knows how to translate.
<code>group1DstCount</code>	The number of file types that the extension can write in a group.

Translation Manager

group1DstEntrySize

The size of the file specification records in the array that follows this field. In general, you can set this field to `SizeOf(FileTypeSpec)`.

group1DstTypes

An array of file specification records. You should include a file specification record in this array for each file type that your extension can translate into.

Scrap Type Specifications

You use the `ScrapTypeSpec` data structure to describe a specific scrap format.

```
TYPE ScrapTypeSpec =
    RECORD
        format:           ScrapType;
        hint:             LongInt;
    END;
```

Field descriptions

format	The type of the specified scrap.
hint	A 4-byte value reserved for use by your translation extension.

Scrap Translation Lists

You use the `ScrapTranslationList` data structure to describe which scrap formats your extension can translate into which other scrap formats. The Translation Manager uses the **scrap translation list** that it gets from each translation system to create a master database of its translation capability.

```
TYPE ScrapTranslationList =
    RECORD
        modDate:           LongInt;
        groupCount:       LongInt;
        {group1SrcCount:   LongInt;}
        {group1SrcEntrySize: LongInt;}
        {group1SrcTypes:  ARRAY[1..group1SrcCount] OF ScrapTypeSpec;}
        {group1DstCount:  LongInt;}
        {group1DstEntrySize: LongInt;}
        {group1DstTypes:  ARRAY[1..group1DstCount] OF ScrapTypeSpec;}
        {repeat above six lines for a total of groupCount times}
    END;
ScrapTranslationListPtr   = ^ScrapTranslationList;
ScrapTranslationListHandle = ^ScrapTranslationListPtr;
```

Translation Manager

A scrap translation list consists of a field indicating the modification date of the list and a count of the number of groups that follow those two fields. The size of the translation list prepared by an extension is variable, depending upon the number of groups, the scrap specification record size, and the number of scrap types that the extension knows about.

Field descriptions

<code>modDate</code>	The creation date of the scrap translation list. If your extension uses external translators, you might set this field to the modification date of a folder containing those translators.
<code>groupCount</code>	The number of translation groups that follow.
<code>group1SrcCount</code>	The number of scrap types that the extension can read in a group.
<code>group1SrcEntrySize</code>	The size of the scrap specification records in the array that follows this field. In general, you can set this field to <code>SizeOf(ScrapTypeSpec)</code> .
<code>group1SrcTypes</code>	An array of scrap specification records. You should include a scrap specification record in this array for each scrap type that your extension knows how to translate.
<code>group1DstCount</code>	The number of scrap types that the extension can write in a group.
<code>group1DstEntrySize</code>	The size of the scrap specification records in the array that follows this field. In general, you can set this field to <code>SizeOf(ScrapTypeSpec)</code> .
<code>group1DstTypes</code>	An array of scrap specification records. You should include a scrap specification record in this array for each scrap type that your extension can translate into.

Translation Extension Routines

This section describes two routines that you can call from within a translation extension.

Managing Translation Progress Dialog Boxes

You can use the `SetTranslationAdvertisement` function to display the progress dialog box and, optionally, to include a logo or other identifying picture in the progress dialog box. You can use the `UpdateTranslationProgress` function to show the user the progress of a translation and allow the user to cancel a translation.

SetTranslationAdvertisement

A translation extension can call `SetTranslationAdvertisement` to install an advertisement into the progress dialog box.

```
FUNCTION SetTranslationAdvertisement (refNum: TranslationRefNum;
                                   advertisement: PicHandle)
                                   : OSErr;
```

`refNum` **A translation reference number.**

`advertisement` **A handle to a picture to display in the upper portion of the dialog box.**

DESCRIPTION

The `SetTranslationAdvertisement` function installs a translation extension-specific picture into the upper portion of a translation progress dialog box, then displays the dialog box. The `advertisement` parameter should be a handle to the picture to display. If the value of `advertisement` is `NIL`, no advertisement is displayed and the upper portion of the dialog box is removed before the box is displayed to the user.

Your translation extension can read the picture data from its resource fork, but it should detach the resource from the resource fork (by calling `DetachResource`) and make the handle unpurgeable before calling `SetTranslationAdvertisement`. Because you'll usually load the picture data into the temporary heap provided for the translation extension, the picture data is automatically disposed of when that heap is destroyed. If your translation extension loads the picture data elsewhere in memory, you are responsible for disposing of it before returning from your `DoTranslateFile` or `DoTranslateScrap` routine.

The size of the picture to display can be no larger than 280 by 50 pixels. If the picture you specify is smaller than that, it is automatically centered (both vertically and horizontally) in the available space.

You should set the `refNum` parameter to the translation reference number passed to your `DoTranslateFile` or `DoTranslateScrap` routine. The Translation Manager uses that number internally.

SPECIAL CONSIDERATIONS

Your translation extension should call `SetTranslationAdvertisement` only in response to the `kTranslateTranslateFile` or `kTranslateTranslateScrap` request code (that is, in your `DoTranslateFile` or `DoTranslateScrap` routine). Do not call this function in response to any other request code or from any code that isn't a translation extension.

Translation Manager

You must call `SetTranslationAdvertisement` before you call the `UpdateTranslationProgress` procedure for the first time.

The `SetTranslationAdvertisement` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_TranslationDispatch</code>	<code>\$0002</code>

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Parameter error
<code>memFullErr</code>	-108	Not enough memory

SEE ALSO

See Figure 7-4 on page 7-7 for a sample translation progress dialog box showing an advertisement. See Listing 7-8 on page 7-34 for an example of the use of `SetTranslationAdvertisement`.

UpdateTranslationProgress

A translation extension can call `UpdateTranslationProgress` to update the progress dialog box that is displayed during file and scrap translation and to give the user a chance to click the Cancel button.

```
FUNCTION UpdateTranslationProgress (refNum: TranslationRefNum;
                                   percentDone: Integer;
                                   VAR canceled: Boolean)
                                   : OSErr;
```

`refNum` A translation reference number.

`percentDone` An integer in the range 0–100 that indicates the percentage of the translation that has been completed.

Translation Manager

`canceled` On exit, `UpdateTranslationProgress` returns `TRUE` in this parameter if the user clicked the Cancel button in the progress dialog box; otherwise, `UpdateTranslationProgress` returns `FALSE` in this parameter.

DESCRIPTION

The `UpdateTranslationProgress` function updates the translation progress dialog box. You should call this function periodically during a translation to update the progress bar and to give the user an opportunity to cancel the translation. If the user clicks the Cancel button in the dialog box (or types Command-period while the box is displayed), the `canceled` parameter is set to `TRUE`; otherwise, it is set to `FALSE`. When `canceled` returns `TRUE`, you should stop the translation, and your application-defined routine `DoTranslateFile` or `DoTranslateScrap` should return the result code `userCancelledErr`.

The `percentDone` parameter specifies the approximate percentage of time elapsed until completion. You should call `UpdateTranslationProgress` periodically at reasonable time intervals to allow the user to cancel the translation. When the translation is complete, you should call `UpdateTranslationProgress` with `percentDone` set to 100 so that the user can see that the translation is complete.

You should set the `refNum` parameter to the translation reference number passed to your `DoTranslateFile` or `DoTranslateScrap` routine. The Translation Manager uses that number internally.

SPECIAL CONSIDERATIONS

Your translation extension should call `UpdateTranslationProgress` only in response to the `kTranslateTranslateFile` or `kTranslateTranslateScrap` request code (that is, in your `DoTranslateFile` or `DoTranslateScrap` routine). Do not call this function in response to any other request code or from any code that isn't a translation extension.

You should already have called `SetTranslationAdvertisement` before calling `UpdateTranslationProgress`.

The `UpdateTranslationProgress` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

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

Trap macro	Selector
<code>_TranslationDispatch</code>	<code>\$0001</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>paramErr</code>	<code>-50</code>	Parameter error
<code>memFullErr</code>	<code>-108</code>	Not enough memory

Translation Extension-Defined Routines

This section describes the routines you'll need to define in order to write a translation extension. You can create both file and scrap translation extensions.

To construct a translation extension to translate files, you need to create a component that responds to the `kTranslateGetFileTranslationList`, `kTranslateIdentifyFile`, and `kTranslateTranslateFile` request codes. In response to these request codes, you typically dispatch to one of the extension-defined routines `DoGetFileTranslationList`, `DoIdentifyFile`, and `DoTranslateFile`. To construct a translation extension that translates scraps, you need to create a component that responds to the `kTranslateGetScrapTranslationList`, `kTranslateIdentifyScrap`, and `kTranslateTranslateScrap` request codes. In response to these request codes, you typically dispatch to one of the extension-defined routines `DoGetScrapTranslationList`, `DoIdentifyScrap`, and `DoTranslateScrap`.

All routines return result codes. If they succeed, they should return `noErr`. The Component Manager requires these routines to return a value of type `ComponentResult`—a value of type `LongInt`—to simplify dispatching.

See “Dispatching to Translation Extension-Defined Routines” beginning on page 7-24 for a description of how you call these routines from within a translation extension.

File Translation Extension Routines

To write a file translation extension, you need to define three routines:

- n `DoGetFileTranslationList`
- n `DoIdentifyFile`
- n `DoTranslateFile`

DoGetFileTranslationList

A file translation extension must respond to the `kTranslateGetFileTranslationList` request code. Whenever it first notices the extension, Macintosh Easy Open calls your extension with this request code to obtain a

Translation Manager

list of the file types that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoGetFileTranslationList` function.

```
FUNCTION DoGetFileTranslationList
    (self: ComponentInstance;
     translationList: FileTranslationListHandle)
    : ComponentResult;
```

`self` **A component instance that identifies the component containing the translation extension.**

`translationList` **A handle to a file translation list.**

DESCRIPTION

Your `DoGetFileTranslationList` function should return, through the `translationList` parameter, a handle to a list of the file types from and into which your translation extension can translate. On entry to `DoGetFileTranslationList`, the `translationList` parameter contains a handle to a structure of type `FileTranslationList`. If your translation extension can translate any files at all, your `DoGetFileTranslationList` function should resize that handle and fill the block with a list of the file types it can translate. If the translation list whose handle you return in `translationList` has the `groupCount` field set to 0, Macintosh Easy Open assumes that your extension cannot translate any file types.

Macintosh Easy Open calls your translation extension's `DoGetFileTranslationList` function when it first becomes aware of your extension. For improved performance, Macintosh Easy Open remembers each translation extension's most recently returned file translation list and passes that list to `DoGetFileTranslationList` in the `translationList` parameter. If you determine that the list hasn't changed, you should simply return the same handle to Macintosh Easy Open.

RESULT CODES

The `DoGetFileTranslationList` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

See "File Translation Lists" on page 7-48 for a description of the `FileTranslationList` data structure. See "Writing a Translation Extension" beginning on page 7-18 for more information about implementing a translation extension. See Listing 7-6 on page 7-30 for a routine that constructs a sample file translation list.

DoIdentifyFile

A file translation extension must respond to the `kTranslateIdentifyFile` request code. The Translation Manager uses this request code to allow the translation extension to identify a file as having a format that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoIdentifyFile` function.

```
FUNCTION DoIdentifyFile (self: ComponentInstance;
                        theDoc: FSSpec;
                        VAR docKind: FileType)
                        : ComponentResult;
```

<code>self</code>	A component instance that identifies the component containing the translation extension.
<code>theDoc</code>	A file system specification record that specifies the document that the translation extension must identify.
<code>docKind</code>	On exit, the file format type of the document as identified by your translation extension.

DESCRIPTION

Your `DoIdentifyFile` function returns, through the `docKind` parameter, the file type of the file specified by the `FSSpec` record passed in the `theDoc` parameter. If your translation extension does not recognize the type of the specified file, `DoIdentifyFile` should return the result code `noTypeErr`.

SPECIAL CONSIDERATIONS

Your `DoIdentifyFile` function should not return 'TEXT' as a file type unless you determine that the document consists solely of a plain, unformatted stream of ASCII characters.

RESULT CODES

<code>noErr</code>	0	No error
<code>noTypeErr</code>	-102	Unrecognized file type

DoTranslateFile

A file translation extension must respond to the `kTranslateTranslateFile` request code. The Translation Manager uses this request code to allow the translation extension to translate a file from one format to another. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoTranslateFile` function.

```
FUNCTION DoTranslateFile (self: ComponentInstance;
                          refNum: TranslationRefNum;
                          srcDoc: FSSpec;
                          srcType: FileType;
                          srcTypeHint: LongInt;
                          dstDoc: FSSpec;
                          dstType: FileType;
                          dstTypeHint: LongInt): ComponentResult;
```

<code>self</code>	A component instance that identifies the component containing your translation extension.
<code>refNum</code>	The translation reference number for this translation.
<code>srcDoc</code>	A file system specification record that specifies the source document.
<code>srcType</code>	The format of the file to be translated.
<code>srcTypeHint</code>	The value in the <code>hint</code> field of the source document's file type specification.
<code>dstDoc</code>	A file system specification record that specifies the destination document.
<code>dstType</code>	The format into which to translate the source document.
<code>dstTypeHint</code>	The value in the <code>hint</code> field of the destination document's file type specification.

DESCRIPTION

Your `DoTranslateFile` function translates a document from one format into another. The document to be translated is specified by the `srcDoc` parameter, and your routine should put the translated document into the file specified by the `dstDoc` parameter. The data fork of the destination file already exists by the time `DoTranslateFile` is called. In addition, if the `flags` field in the appropriate destination file type specification in your extension's file translation list has the `taDstDocNeedsResourceFork` bit set, the destination file already contains a resource fork. Your function should open the destination file and fill its data or resource fork (or both) with the appropriate translated data.

Translation Manager

The `refNum` parameter is a reference number that Macintosh Easy Open assigns to the translation. Each translation is assigned a unique number to distinguish the translation from any other translations that might occur. You need to pass this reference number to any Macintosh Easy Open routines you call from within the file translation extension; for instance, if by calling the `SetTranslationAdvertisement` function you display the progress dialog box, you'll pass that reference number in the `refNum` parameter.

The `DoTranslateFile` function can translate the source file itself or rely upon external translators. If it cannot translate the source file, your function should return a result code different from `noErr`. In that case, Macintosh Easy Open will automatically delete the destination file.

Your translation extension should call the `SetTranslationAdvertisement` function to display the progress dialog box and the `UpdateTranslationProgress` function to update the dialog box periodically.

Your `DoTranslateFile` function should return `noErr` if successful or an appropriate result code otherwise.

RESULT CODES

<code>noErr</code>	0	No error
<code>userCanceledErr</code>	-128	User canceled the translation
<code>invalidTranslationPathErr</code>	-3025	<code>srcType</code> to <code>dstType</code> is not a valid translation path
<code>couldNotParseSourceFileErr</code>	-3026	The source document is not of type <code>srcType</code>

Scrap Translation Extension Routines

To write a scrap translation extension, you need to define three routines:

- n `DoGetScrapTranslationList`
- n `DoIdentifyScrap`
- n `DoTranslateScrap`

DoGetScrapTranslationList

A scrap translation extension must respond to the `kTranslateGetScrapTranslationList` request code. At system startup time, the Translation Manager calls your extension with this request code to obtain a list of the

scrap types that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoGetScrapTranslationList` function.

```
FUNCTION DoGetScrapTranslationList
    (self: ComponentInstance;
     list: ScrapTranslationListHandle)
    : ComponentResult;
```

`self` A component instance that identifies the component containing the translation extension.

`list` A handle to a scrap translation list.

DESCRIPTION

The `DoGetScrapTranslationList` function returns, through the `list` parameter, a handle to a list of the scrap types from and into which your translation extension can translate. On entry to `DoGetScrapTranslationList`, the `list` parameter contains a handle to a structure of type `ScrapTranslationList`. If your translation extension can translate any scrap types at all, your `DoGetScrapTranslationList` function should resize that handle and fill the block with a list of the scrap types it can translate. If the translation list whose handle you return in `list` has the `groupCount` field set to 0, Macintosh Easy Open assumes that your extension cannot translate any scrap types.

When it first becomes aware of your extension, Macintosh Easy Open calls your translation extension's `DoGetScrapTranslationList` function. For improved performance, Macintosh Easy Open remembers each translation extension's most recently returned scrap translation list and passes that list to `DoGetScrapTranslationList` in the `list` parameter. If you determine that the list hasn't changed, you should simply return the same handle to Macintosh Easy Open.

RESULT CODES

The `DoGetScrapTranslationList` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

See "Scrap Translation Lists" on page 7-49 for a description of the `ScrapTranslationList` data structure. See "Writing a Translation Extension" beginning on page 7-18 for more information about implementing a translation extension.

DoIdentifyScrap

A scrap translation extension must respond to the `kTranslateIdentifyScrap` request code. The Translation Manager uses this request code to allow the translation extension to identify a scrap as one that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoIdentifyScrap` function.

```
FUNCTION DoIdentifyScrap (self: ComponentInstance;
                        dataPtr: Ptr;
                        dataLength: Size;
                        VAR dataFormat: ScrapType)
                        : ComponentResult;
```

`self` A component instance that identifies the component containing the translation extension.

`dataPtr` A pointer to the scrap.

`dataLength` The size of the scrap to be translated.

`dataFormat` On entry, the type of the scrap format. On exit, the type of the scrap format as recognized by your translation extension.

DESCRIPTION

Your `DoIdentifyScrap` function returns, through the `dataFormat` parameter, the scrap type of the scrap specified by the `dataPtr` and `dataLength` parameters. If your translation extension does not recognize the type of the specified scrap, `DoIdentifyScrap` should return the result code `noTypeErr`.

In general, the scrap that your `DoIdentifyScrap` function is asked to identify is always in one of the formats listed among the source formats in the translation groups contained in your extension's scrap translation list. Your scrap translation extension therefore needs only to verify that the indicated scrap is of the specified format.

RESULT CODES

<code>noErr</code>	0	No error
<code>noTypeErr</code>	-102	Unrecognized scrap type

DoTranslateScrap

A scrap translation extension must respond to the `kTranslateTranslateScrap` request code. The Translation Manager sends this request code to allow the extension to translate scraps from one format to another. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoTranslateScrap` function.

```
FUNCTION DoTranslateScrap (self: ComponentInstance;
                          refNum: TranslationRefNum;
                          srcDataPtr: Ptr;
                          srcDataLength: Size;
                          srcType: ScrapType;
                          srcTypeHint: LongInt;
                          dstData: Handle;
                          dstType: ScrapType;
                          dstTypeHint: LongInt)
                          : ComponentResult;
```

<code>self</code>	A component instance that identifies the component containing the translation extension.
<code>refNum</code>	The translation reference number for this translation.
<code>srcDataPtr</code>	A pointer to the scrap to be translated.
<code>srcDataLength</code>	The size of the scrap to be translated.
<code>srcType</code>	The format of the scrap to be translated.
<code>srcTypeHint</code>	The value in the <code>hint</code> field of the source document's scrap type specification.
<code>dstData</code>	A handle to the destination to be filled in.
<code>dstType</code>	The format into which to translate the source scrap.
<code>dstTypeHint</code>	The value in the <code>hint</code> field of the destination document's scrap type specification.

DESCRIPTION

The `DoTranslateScrap` function translates a scrap from one format into another. The scrap to be translated is specified by the `srcDataPtr` and `srcDataLength` parameters, and your routine should put the translated data into the block specified by the `dstData` parameter. Your function should resize that block as necessary and fill it with the appropriate translated data.

Translation Manager

The `refNum` parameter is a reference number that Macintosh Easy Open assigns to the translation. Each translation is assigned a unique number to distinguish the translation from any other translations that might be occurring. You need to pass this reference number to any Macintosh Easy Open routines you call from within the scrap translation extension; for instance, if you display the progress dialog box by calling the `SetTranslationAdvertisement` function, you'll pass that reference number in the `refNum` parameter.

The `DoTranslateScrap` function can translate the source file itself or rely upon external translators. If it cannot translate the source scrap, your function should return a result code different from `noErr`.

Your translation extension should call the `SetTranslationAdvertisement` function to display the progress dialog box and the `UpdateTranslationProgress` function to update the dialog box periodically.

RESULT CODES

The `DoTranslateScrap` function should return `noErr` if successful, or an appropriate result code otherwise.

Summary of the Translation Manager

This section provides Pascal, C, and assembly-language summaries for the constants, data types, and routines provided by the Translation Manager for use by applications. For a summary of the constants, data types, and routines that you can use or need to define if you're writing a translation extension, see "Summary of Translation Extensions" beginning on page 7-68.

Pascal Summary

Constants

CONST

```
{Gestalt selectors and response bit numbers}
gestaltTranslationAttr          = 'xlat';    {Translation Manager}
gestaltTranslationMgrExists    = 0;         {TM is present}

gestaltStandardFileAttr       = 'stdf';    {Standard File Package}
gestaltStandardFile58         = 0;
gestaltStandardFileTranslationAware = 1;
gestaltStandardFileHasColorIcons = 2;

gestaltEditionMgrAttr         = 'edtn';    {Edition Manager}
gestaltEditionMgrPresent      = 0;
gestaltEditionMgrTranslationAware = 1;

gestaltScrapMgrAttr           = 'scra';    {Scrap Manager}
gestaltScrapMgrTranslationAware = 0;
```

Data Types

TYPE

```
FileType          = OSType;           {file types}
ScrapType         = ResType;          {scrap types}

FileTypePtr       = ^FileType;

FileTranslationSpec = ARRAY[1..12] OF LongInt;
```

Translation Manager

```

TypesBlock          = ARRAY[0..63] OF FileType;
TypesBlockPtr       = ^TypesBlock;

DocOpenMethod       = (domCannot,
                       domNative,
                       domTranslateFirst,
                       domWildcard);

```

Translation Manager Routines

Getting Translation Information

```

FUNCTION GetFileTypesThatAppCanNativelyOpen
    (appVRefNumHint: Integer; appSignature: OSType;
     VAR nativeTypes: TypesBlock): OSErr;

FUNCTION ExtendFileTypeList
    (originalTypeList: FileTypePtr;
     numberOriginalTypes: Integer;
     extendedTypeList: FileTypePtr;
     VAR numberExtendedTypes: Integer): OSErr;

FUNCTION CanDocBeOpened
    (targetDocument: FSSpec;
     appVRefNumHint: Integer;
     appSignature: OSType;
     nativeTypes: TypesBlockPtr;
     onlyNative: Boolean;
     VAR howToOpen: DocOpenMethod;
     VAR howToTranslate: FileTranslationSpec)
    : OSErr;

```

Translating Files

```

FUNCTION TranslateFile
    (sourceDocument: FSSpec;
     destinationDocument: FSSpec;
     howToTranslate: FileTranslationSpec): OSErr;

```

C Summary

Constants

```

/*Gestalt selectors and response bit numbers*/
enum {
    #define gestaltTranslationAttr    'xlat'    /*Translation Manager*/

```

Translation Manager

```

    gestaltTranslationMgrExists          = 0          /*TM is present*/
};
enum {
    #define gestaltStandardFileAttr      'stdf'      /*Std File Package*/
    gestaltStandardFile58                = 0,
    gestaltStandardFileTranslationAware  = 1,
    gestaltStandardFileHasColorIcons     = 2
};
enum {
    #define gestaltEditionMgrAttr        'edtn'      /*Edition Manager*/
    gestaltEditionMgrPresent              = 0,
    gestaltEditionMgrTranslationAware     = 1
};
enum {
    #define gestaltScrapMgrAttr          'scra'      /*Scrap Manager*/
    gestaltScrapMgrTranslationAware      = 0
};

enum {domCannot, domNative, domTranslateFirst, domWildcard};

```

Data Types

```

typedef OSType          FileType;          /*file types*/
typedef ResType         ScrapType;        /*scrap types*/

typedef long            FileTranslationSpec[12];

typedef short           DocOpenMethod;

```

Translation Manager Routines

Getting Translation Information

```

pascal OSErr GetFileTypesThatAppCanNativelyOpen
    (short appVRefNumHint, OSType appSignature,
     FileType* nativeTypes);

pascal OSErr ExtendFileTypeList
    (const FileType* originalTypeList,
     short numberOriginalTypes,
     FileType* extendedTypeList,
     short* numberExtendedTypes);

```

Translation Manager

```
pascal OSErr CanDocBeOpened
    (const FSSpec* targetDocument,
     short appVRefNumHint,
     OSType appSignature,
     const FileType* nativeTypes,
     Boolean onlyNative,
     DocOpenMethod* howToOpen,
     FileTranslationSpec* howToTranslate);
```

Translating Files

```
pascal OSErr TranslateFile (const FSSpec* sourceDocument,
    const FSSpec* destinationDocument,
    const FileTranslationSpec* howToTranslate);
```

Assembly-Language Summary

Data Structures

File Translation Specification

0 data 48 bytes private data used by the Translation Manager

Trap Macros

Trap Macros Requiring Routine Selectors

_TranslationDispatch

Selector	Routine
\$0009	ExtendFileTypeList
\$000C	TranslateFile
\$001C	GetFileTypesThatAppCanNativelyOpen
\$001E	CanDocBeOpened

Result Codes

noErr	0	No error
dirFulErr	-33	Directory full
dskFulErr	-34	Not enough disk space to translate file
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename
tmfoErr	-42	Too many files open
fnfErr	-43	File not found
wPrErr	-44	Disk is write protected
fLckdErr	-45	File is locked
vLckdErr	-46	Volume is locked
dupFNerr	-48	Duplicate filename (rename)
opWrErr	-49	File already open with write permission
paramErr	-50	Parameter error
extFSerr	-58	External file system
noTypeErr	-102	Unrecognized file type
memFullErr	-108	Not enough RAM to translate file
dirNFerr	-120	Directory not found or incomplete pathname
wrgVolTypErr	-123	Volume does not support Desktop Manager
userCanceledErr	-128	The user canceled the translation
invalidTranslationPathErr	-3025	howToTranslate is invalid
noTransSysInstalledErr	-3027	No translation systems installed
noTranslationPathErr	-3030	Application cannot open document
badTranslationSpecErr	-3031	Translation path is invalid
noPrefAppErr	-3032	No translation preference available
afpItemNotFound	-5012	Could not determine kind string; or, application information not found

Summary of Translation Extensions

This section provides Pascal, C, and assembly-language summaries for the constants, data types, and routines you can use to write a translation extension. For a summary of the constants, data types, and routines that applications can use, see “Summary of the Translation Manager” beginning on page 7-63.

Pascal Summary

Constants

CONST

```
{component flags}
kSupportsFileTranslation      = 1;  {file translation extension}
kSupportsScrapTranslation     = 2;  {scrap translation extension}

{translation attributes}
taDstDocNeedsResourceFork    = 1;  {doc needs a resource fork}
taDstIsAppTranslation        = 2;  {app will translate doc}

{request codes for translation extensions}
kTranslateGetFileTranslationList = 0;
kTranslateIdentifyFile          = 1;
kTranslateTranslateFile         = 2;
kTranslateGetScrapTranslationList = 10;
kTranslateIdentifyScrap         = 11;
kTranslateTranslateScrap        = 12;
```

Data Types

TYPE

```
FileType          = OSType;  {file types}
ScrapType         = ResType;  {scrap types}

TranslationAttributes = LongInt;

FileTypeSpec =
RECORD
    format:      FileType;
```

CHAPTER 7

Translation Manager

```
hint:          LongInt;
flags:         TranslationAttributes;
catInfoType:   OSType;
catInfoCreator: OSType;
END;

FileTranslationList =
RECORD
  modDate:          LongInt;
  groupCount:       LongInt;
  {group1SrcCount:   LongInt;}
  {group1SrcEntrySize: LongInt;}
  {group1SrcTypes:   ARRAY[1..group1SrcCount] OF FileTypeSpec;}
  {group1DstCount:   LongInt;}
  {group1DstEntrySize: LongInt;}
  {group1DstTypes:   ARRAY[1..group1DstCount] OF FileTypeSpec;}
  {repeat above six lines for a total of groupCount times}
END;
FileTranslationListPtr   = ^FileTranslationList;
FileTranslationListHandle = ^FileTranslationListPtr;

ScrapTypeSpec =
RECORD
  format:          ScrapType;
  hint:           LongInt;
END;

ScrapTranslationList =
RECORD
  modDate:          LongInt;
  groupCount:       LongInt;
  {group1SrcCount:   LongInt;}
  {group1SrcEntrySize: LongInt;}
  {group1SrcTypes:   ARRAY[1..group1SrcCount] OF ScrapTypeSpec;}
  {group1DstCount:   LongInt;}
  {group1DstEntrySize: LongInt;}
  {group1DstTypes:   ARRAY[1..group1DstCount] OF ScrapTypeSpec;}
  {repeat above six lines for a total of groupCount times}
END;
ScrapTranslationListPtr   = ^ScrapTranslationList;
ScrapTranslationListHandle = ^ScrapTranslationListPtr;

TranslationRefNum       = LongInt;
```

Translation Extension Routines

Managing Translation Progress Dialog Boxes

```

FUNCTION SetTranslationAdvertisement
    (refNum: TranslationRefNum;
     advertisement: PicHandle): OSErr;

FUNCTION UpdateTranslationProgress
    (refNum: TranslationRefNum;
     percentDone: Integer;
     VAR canceled: Boolean): OSErr;

```

Translation Extension-Defined Routines

File Translation Extension Routines

```

FUNCTION DoGetFileTranslationList
    (self: ComponentInstance;
     translationList: FileTranslationListHandle)
    : ComponentResult;

FUNCTION DoIdentifyFile
    (self: ComponentInstance;
     theDoc: FSSpec;
     VAR docKind: FileType): ComponentResult;

FUNCTION DoTranslateFile
    (self: ComponentInstance;
     refNum: TranslationRefNum;
     srcDoc: FSSpec;
     srcType: FileType;
     srcTypeHint: LongInt;
     dstDoc: FSSpec;
     dstType: FileType;
     dstTypeHint: LongInt): ComponentResult;

```

Scrap Translation Extension Routines

```

FUNCTION DoGetScrapTranslationList
    (self: ComponentInstance;
     list: ScrapTranslationListHandle)
    : ComponentResult;

FUNCTION DoIdentifyScrap
    (self: ComponentInstance;
     dataPtr: Ptr;
     dataLength: Size;
     VAR dataFormat: ScrapType): ComponentResult;

```

Translation Manager

```

FUNCTION DoTranslateScrap    (self: ComponentInstance;
                             refNum: TranslationRefNum;
                             srcDataPtr: Ptr;
                             srcDataLength: Size;
                             srcType: ScrapType;
                             srcTypeHint: LongInt;
                             dstData: Handle;
                             dstType: ScrapType;
                             dstTypeHint: LongInt): ComponentResult;

```

C Summary

Constants

```

/*component flags*/
#define kSupportsFileTranslation    1 /*file translation extension*/
#define kSupportsScrapTranslation  2 /*scrap translation extension*/

/*translation attributes*/
#define taDstDocNeedsResourceFork  1 /*doc needs a resource fork*/
#define taDstIsAppTranslation      2 /*app will translate doc*/

/*request codes for translation extensions*/
enum {
    kTranslateGetFileTranslationList    = 0,
    kTranslateIdentifyFile,
    kTranslateTranslateFile,
    kTranslateGetScrapTranslationList  = 10,
    kTranslateIdentifyScrap,
    kTranslateTranslateScrap
};

```

Data Types

```

typedef OSType      FileType;          /*file types*/
typedef ResType    ScrapType;        /*scrap types*/

typedef unsigned long TranslationAttributes;

struct FileTypeSpec {
    FileType      format;
    long          hint;
};

```

Translation Manager

```

    TranslationAttributes    flags;
    OSType                   catInfoType;
    OSType                   catInfoCreator;
}
typedef struct FileTypeSpec FileTypeSpec;

struct FileTranslationList {
    unsigned long            modDate;
    unsigned long            groupCount;
    /*unsigned long          group1SrcCount;*/
    /*unsigned long          group1SrcEntrySize;*/
    /*FileTypeSpec           group1SrcTypes[group1SrcCount];*/
    /*unsigned long          group1DstCount;*/
    /*unsigned long          group1DstEntrySize;*/
    /*FileTypeSpec           group1DstTypes[group1DstCount];*/
    /*repeat above six lines for a total of groupCount times*/
};
typedef struct FileTranslationList FileTranslationList;
typedef FileTranslationList *FileTranslationListPtr,
**FileTranslationListHandle;

struct ScrapTypeSpec {
    ScrapType                format;
    long                     hint;
}
typedef struct ScrapTypeSpec ScrapTypeSpec;

struct ScrapTranslationList {
    unsigned long            modDate;
    unsigned long            groupCount;
    /*unsigned long          group1SrcCount;*/
    /*unsigned long          group1SrcEntrySize;*/
    /*ScrapTypeSpec          group1SrcTypes[group1SrcCount];*/
    /*unsigned long          group1DstCount;*/
    /*unsigned long          group1DstEntrySize;*/
    /*ScrapTypeSpec          group1DstTypes[group1DstCount];*/
    /*repeat above six lines for a total of groupCount times*/
};
typedef struct ScrapTranslationList ScrapTranslationList;
typedef ScrapTranslationList *ScrapTranslationListPtr,
**ScrapTranslationListHandle;

typedef long                 TranslationRefNum;

```

Translation Extension Routines

Managing Translation Progress Dialog Boxes

```
pascal OSErr SetTranslationAdvertisement
    (TranslationRefNum refnum,
     PicHandle advertisement);

pascal OSErr UpdateTranslationProgress
    (TranslationRefNum refnum,
     short percentDone,
     Boolean* canceled);
```

Translation Extension-Defined Routines

File Translation Extension Routines

```
pascal ComponentResult DoGetFileTranslationList
    (ComponentInstance self,
     FileTranslationListHandle translationList);

pascal ComponentResult DoIdentifyFile
    (ComponentInstance self,
     const FSSpec* theDoc,
     FileType* docKind);

pascal ComponentResult DoTranslateFile
    (ComponentInstance self,
     TranslationRefNum refNum,
     const FSSpec* srcDoc,
     FileType srcType,
     long srcTypeHint,
     const FSSpec* dstDoc,
     FileType dstType,
     long dstTypeHint);
```

Scrap Translation Extension Routines

```
pascal ComponentResult DoGetScrapTranslationList
    (ComponentInstance self,
     ScrapTranslationListHandle list);

pascal ComponentResult DoIdentifyScrap
    (ComponentInstance self,
     const void* dataPtr,
     Size dataLength,
     ScrapType* dataFormat);
```

```

pascal ComponentResult DoTranslateScrap
    (ComponentInstance self,
     TranslationRefNum refNum,
     const void* srcDataPtr,
     Size srcDataLength,
     ScrapType srcType,
     long srcTypeHint,
     Handle dstData,
     ScrapType dstType,
     long dstTypeHint);

```

Assembly-Language Summary

Data Structures

File Type Specification

0	format	4 bytes	the file type
4	hint	4 bytes	reserved for use by your translation extension
8	flags	4 bytes	flags for controlling translation
12	catInfoType	4 bytes	the file's catalog type
16	catInfoCreator	4 bytes	the file's catalog creator

File Translation List

0	modDate	4 bytes	the creation date of the file translation list
4	groupCount	4 bytes	the number of translation groups that follow

Scrap Type Specification

0	format	4 bytes	the scrap type
4	hint	4 bytes	reserved for use by your translation extension

Scrap Translation List

0	modDate	4 bytes	the creation date of the scrap translation list
4	groupCount	4 bytes	the number of translation groups that follow

Trap Macros

Trap Macros Requiring Routine Selectors

_TranslationDispatch

Selector	Routine
\$0001	UpdateTranslationProgress
\$0002	SetTranslationAdvertisement

Result Codes

noErr	0	No error
dskFulErr	-34	Not enough disk space to translate file
fnfErr	-43	Document not found
paramErr	-50	Parameter error
noTypeErr	-102	Unrecognized file or scrap type
memFullErr	-108	Not enough memory
dirNFErr	-120	Source or destination directory does not exist
wrgVolTypErr	-123	Volume does not support Desktop Manager
userCanceledErr	-128	The user canceled the translation
invalidTranslationPathErr	-3025	srcType to dstType is not a valid path
couldNotParseSourceFileErr	-3026	Source document is not of type srcType
afpItemNotFound	-5012	Could not determine kind string; or, application information not found

Control Panels

Contents

About Control Panels	8-4
Control Panels	8-4
A Control Panel's Resources	8-6
The Finder's Interaction With Control Panels	8-7
Control Panels and System Extensions	8-8
About User Documentation for Control Panels	8-8
The Monitors Control Panel and Extensions to It	8-9
Creating Control Panel Files	8-12
Defining the User Interface for a Control Panel	8-12
Creating a Control Panel's Resources	8-14
Resource IDs for Control Panels	8-14
Defining the Control Panel Rectangles	8-15
Creating the Item List Resource	8-17
Defining the Icon for a Control Panel	8-20
Specifying the Machine Resource	8-20
Creating the File Reference, Bundle, and Signature Resources	8-21
Providing Additional Resources for a Control Panel	8-22
Specifying the Font of Text in a Control Panel	8-23
Creating a Font Information Resource	8-23
Defining Text in a Control Panel as User Items	8-24
Writing a Control Panel Function	8-25
Determining If a Control Panel Can Run on the Current System	8-29
Initializing the Control Panel Items and Allocating Storage	8-29
Responding to Activate Events	8-33
Responding to Keyboard Events	8-37
Responding to Mouse Events	8-39
Responding to Update Events	8-43
Handling Text Defined as User Items	8-43
Responding to Null Events	8-45
Responding to the User Closing the Control Panel	8-45

Handling Edit Menu Commands	8-46
Handling Errors	8-47
Creating an Extension for the Monitors Control Panel	8-48
Designing the User Interface for a Monitors Extension	8-49
Creating the Required Resources for a Monitors Extension	8-51
Creating a Card Resource for a Monitors Extension	8-51
Defining a Rectangle for a Monitors Extension	8-52
Creating an Item List Resource for a Monitors Extension	8-54
Creating the Monitor Code Resource	8-56
Supplying Optional Resources for a Monitors Extension	8-56
Specifying an Icon for the Options Dialog Box	8-57
Specifying Version Information	8-58
Providing an Alternative Name for a Video Card	8-58
Supplying Gamma Table Resources	8-59
Creating File Reference, Bundle, and Signature Resources	8-59
Including a System Extension Resource	8-61
Writing a Monitors Extension Function	8-61
Handling the Startup Message	8-66
Performing Initialization	8-68
Responding to a Click in the OK Button	8-70
Responding to a Cancel Request	8-71
Handling Mouse Events for a Monitors Extension	8-71
Handling Keyboard Events	8-73
Including Another Control Panel Definition in a Monitors Extension File	8-73
Control Panels Reference	8-74
Application-Defined Routines	8-74
Control Device Functions	8-74
Monitors Extension Functions	8-78
Resources	8-82
The Machine Resource	8-84
The Rectangle Positions Resource	8-85
The Font Information Resource	8-86
The Control Device Function Code Resource	8-87
The Card Resource	8-87
The Monitor Code Resource	8-88
The Rectangle Resource	8-88
Summary of Control Panels	8-89
Pascal Summary	8-89
Constants	8-89
Application-Defined Routines	8-90
C Summary	8-90
Constants	8-90
Application-Defined Routines	8-92

Control Panels

This chapter describes how to develop a control panel to control the settings of systemwide features and how to create an extension for the standard Monitors control panel.

Create a control panel if you want to provide users with the ability to set preferences for global values or systemwide features. Some of the standard control panels allow users to change the speaker volume, set the date and time, and select a different desktop pattern. Although you must not develop control panels to replace the standard ones, you can create additional control panels for any features that meet the stipulations for control panels. If the feature that you want to implement as a control panel is complex or if its interface requires menu items and multiple, layered dialog boxes, you should create a small application instead of a control panel.

If you are a manufacturer of a video device, you can extend the standard Monitors control panel to include items that give users a simple way to control the features of your device. To do this, read the sections in this chapter that describe how to create an extension to the Monitors control panel. The standard Monitors control panel lets the user define the monitor's display of colors and, if more than one monitor is connected to the system, the relative position of each monitor. The Monitors control panel manages any extensions to it that you create.

To use this chapter, you should be familiar with how to create 'BNDL', 'ICN#', 'FREF', signature, and 'DITL' resources as described in the chapters "Finder Interface" and "Dialog Manager" of *Inside Macintosh: Macintosh Toolbox Essentials*. You should also understand how to handle events and change settings of controls, as explained in the chapters "Event Manager" and "Control Manager" of *Inside Macintosh: Macintosh Toolbox Essentials*.

The Finder, which performs a number of services for your control panel, uses the Dialog Manager to display your control panel's dialog box. In turn, the Dialog Manager uses the Control Manager to create and display buttons, radio buttons, checkboxes, and pop-up menus. Your control panel needs to make these controls active and inactive in response to messages from the Finder. If you include editable text items in your control panel, 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 general introduction to control panels and introduces the Monitors control panel. It then describes how to

- n define the user interface for a control panel
- n create the resources for a control panel, including the rectangle and item list resources
- n specify the font for your control panel's text
- n write a control panel function
- n write an extension for the Monitors control panel

About Control Panels

This section provides an overview of control panels, the resources that a control panel requires, and the Finder's relationship to a control panel. It also distinguishes the services the Finder performs for a control panel from those that your control panel code must implement.

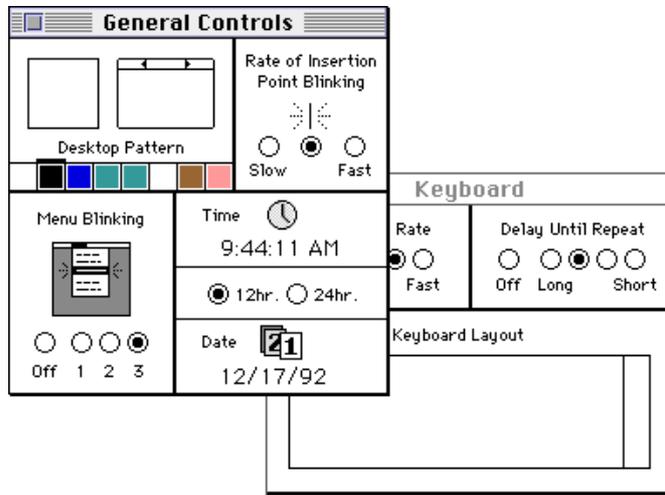
This section also provides an overview of the Monitors control panel and extensions to it, including the resources that a monitors extension requires.

Control Panels

A **control panel** manages the settings of a systemwide feature, such as the amount of memory allocated to a disk cache, the volume of the speaker, or the picture displayed by a screen saver. A control panel can also allow the user to set a global value, such as the highlight color. On the screen, a control panel appears as a modeless dialog box with controls that let users specify basic settings and preferences for the feature. A **control panel file** (a file of type 'cdev') contains the required resources that implement the feature and define the look of the control panel's user interface, including its icon. A control panel file also contains any optional resources needed to implement the feature.

Among the required resources in a control panel file is a code resource that consists of a control device function. A **control device function**, also referred to as a `cdev` function, implements the features of the control panel and performs any services offered by the control panel. Control device functions interact and communicate with the Finder. The Finder provides a number of services for control device functions, including interfacing with the Dialog Manager to create and manage each control panel's dialog box.

A control panel allows the user to modify whatever settings the particular control panel supports. A user opens a control panel from the Finder. Each control panel appears in its own dialog box. Because each control panel is an independent executable file, more than one control panel can remain open at a time, and the user can move among them or run another application while one or more control panels are open. Figure 8-1 shows two control panels open on the desktop. Like other windows, control panels can be dragged on the desktop. The frontmost control panel is the active one.

Figure 8-1 Two control panels, each with its own window

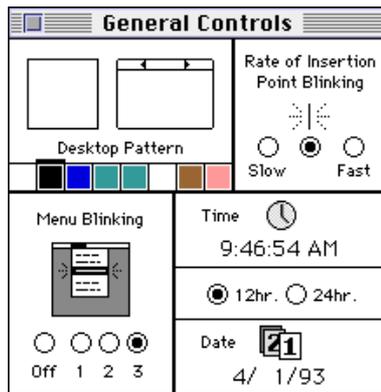
You cannot define your own menus for a control panel, but the user can use most of the Finder's Edit menu commands while working in the control panel. When your control panel is active and the user chooses a command from the Edit menu, the Finder passes the Undo, Cut, Copy, Paste, or Clear commands to your control device function for processing. Your control device function can respond to these messages from the Finder when it is appropriate to do so; for example, if your control panel has an editable text item, your function should respond to editing commands.

Many standard control panels are provided with the system software. For example, the Sound control panel lets the user specify the volume and type of alert sound. The Mouse control panel lets the user define the speed of the onscreen cursor relative to movement of the mouse; the user can also set the double-click speed. The Startup Disk control panel lets the user specify the boot drive.

Control Panels

Figure 8-2 shows the General Controls control panel, which lets the user set the Finder's desktop color and pattern, the blinking rate of the insertion point, the number of times a menu item blinks once the user chooses it, and the time and date.

Figure 8-2 The General Controls control panel



A Control Panel's Resources

A control panel file must contain certain required resources. In addition to these, your control panel can include optional resources. You can also create any other types of resources that your control device function needs and include them in the control panel file. The resources you provide in your control panel file must adhere to conventions governing the resource ID numbers; see “Resource IDs for Control Panels” on page 8-14 for information on these conventions. These are the required resources:

- n A rectangle positions ('nrct') resource. This resource specifies the number of rectangles that make up the display area of your control panel and a list of the coordinates defining the position for each rectangle. (Your control panel interface can have one or more rectangles containing the controls that let the user set and change values or otherwise manipulate the feature the control panel governs.)
- n An item list ('DITL') resource. This resource specifies the items in your control panel. You can specify in this resource items such as static text, buttons, checkboxes, radio buttons, editable text, user items, icons, QuickDraw pictures, and other types of controls, such as pop-up menus.
- n A machine ('mach') resource. This resource specifies the types of systems on which your control panel can run.
- n A black-and-white icon list ('ICN#') resource and other resources associated with an icon family. These resources define the icon for your control panel file. The icon family resources are 'ICN#', 'ics#', 'icl8', 'icl4', 'ics8', and 'ics4'. See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to create an icon family.

Control Panels

- n A bundle ('BNDL') resource. This resource groups together the control panel's signature, icon, and file reference resources.
- n A file reference ('FREF') resource. This resource associates icons with your control panel file; the Finder uses this information to display the icon for your control panel file.
- n A signature resource. This resource contains a unique four-character sequence that has the same value as your control panel's creator type.
- n A control device ('cdev') code resource. This resource contains the code that implements the control panel.

Although it is not required, you can also include a font information ('finf') resource in your control panel file. This resource type lets you specify the font of your control panel's static text items. If you don't include a font information resource, the Finder uses the default application font, which is 9-point Geneva for Roman scripts.

The control device code resource contains a control device function, which must be the first section of code in the resource. The control device function handles messages from the Finder and implements the work your control panel is designed to do. The Finder handles such actions as displaying your control panel's dialog box and tracking controls in it.

The Finder's Interaction With Control Panels

The Finder performs the following services on behalf of your control panel:

- n queries your control device function initially, to determine whether it can run on the available software and hardware configuration
- n requests your control device function to perform any needed initialization when the user first opens your control panel
- n displays dialog items defined by your control panel file
- n tracks user actions in controls defined by your control panel file
- n manages the modeless dialog box in which your control panel is displayed (For instance, the Finder responds appropriately when the user drags the modeless dialog box or clicks its close box.)
- n sends your control device function the information it needs to respond to specific events or to handle Edit menu commands
- n displays messages to the user when the control panel cannot run on the current system and when your control device function returns an error code

Your control panel should

- n provide both the required resources and any additional resources that the Finder needs to run your control panel
- n initialize, open, and close your control panel appropriately as requested by the Finder

Control Panels

- n respond to activate events as requested by the Finder
- n draw user items in response to update events as requested by the Finder
- n respond to user actions in controls as requested by the Finder
- n respond to user keystrokes as requested by the Finder

Control Panels and System Extensions

Many control panels rely on system extensions (files of type 'INIT') to implement their features. For example, you might implement a screen saver as a system extension and create a control panel that allows the user to set specific features of the screen saver, such as the color of the picture displayed. Although the extension creates and manages the screen saver, the user might control the look of the screen saver through settings in the control panel. In this scenario, which is used as an example throughout this chapter, the control device function and its system extension communicate and share values related to settings that the user changes.

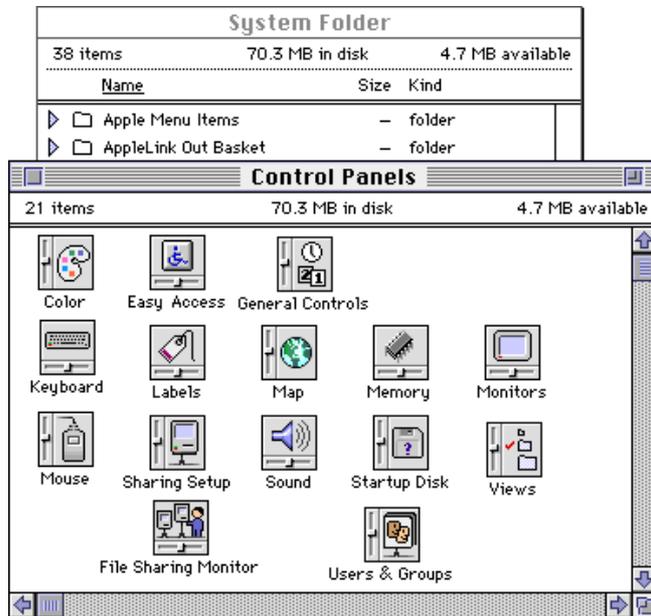
If you use a system extension with your control panel, include it in the control panel file along with the required resources and any other optional resources you use. In System 7, system extensions can be installed in the Control Panels folder or the Extensions folder (both of which are stored in the System Folder) or directly in the System Folder.

However, if it contains a system extension, your control panel file must reside in the Control Panels folder within the System Folder. At startup time, the system software opens files of type 'cdev' that reside in the Control Panels folder and executes any system extensions that it finds there. If the system extension portion of a control panel is not loaded at startup, the control panel won't function properly.

About User Documentation for Control Panels

Because control panels are like independent files, you or the user can install and store them anywhere in the file system. Users might want to store frequently used control panels in the Apple Menu Items folder or in a folder containing other utilities.

You should refer to a file of type 'cdev' as a *control panel file* in any user documentation that you provide. Don't refer by name to the file type of this file or any other file. If your control panel file includes a system extension, you should direct the user to install it in the Control Panels folder or provide an installation script for this purpose. System software provides an alias (a file that points to another file) of the Control Panels folder for quick access from the Apple menu. Figure 8-3 shows many control panel icons in the Control Panels folder.

Figure 8-3 Control panel icons in the Control Panels folder

The Monitors Control Panel and Extensions to It

The standard Monitors control panel lets the user define the monitor's display of colors or shades of gray. If more than one monitor is connected to the system, the Monitors control panel also allows the user to define the relative position of each monitor and choose which monitor is the startup screen. If you are a manufacturer of a video card, you can create a monitors extension to give users a simple way to control the features of your device through the Monitors control panel. A monitors extension controls the features of your video card only, not systemwide features. For example, a monitors extension might allow the user to set the virtual screen size for a single monitor but not the size of the menu bar, which can appear on any monitor. If you require a more complex interface, such as your own menu items or several levels of nested dialog boxes, you should create a small application rather than an extension to the Monitors control panel.

The Monitors control panel manages any extensions to it that you create, and the user can open an extension only through the Monitors control panel. Like a control panel file, a monitors extension file has a file type of 'cdev'. A monitors extension file contains resources for the monitors extension, including a code resource of type 'mtr'. If you want to create a separate control panel to let the user control the settings of another feature of the same video card, you can include the control panel's resources and code in the same file as your monitors extension. In this case, you create the control panel just as you do any other independent control panel. If a user opens your independent control

Control Panels

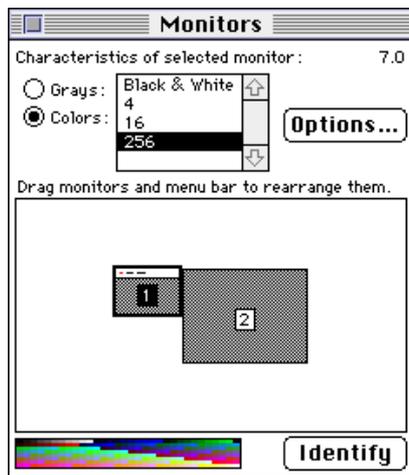
panel, the Finder displays the control panel defined in your file and ignores the monitors extension in that file, just as the Monitors control panel ignores the independent control panel defined in your file when it opens the file to display the monitors extension.

The Monitors control panel allows a user to

- n select which one of the monitors connected to the computer to use as a startup screen (that is, which monitor displays the menu bar)
- n inform system software about the relative locations of the monitors
- n control some features of the monitors, for instance, how many colors or shades of gray are displayed

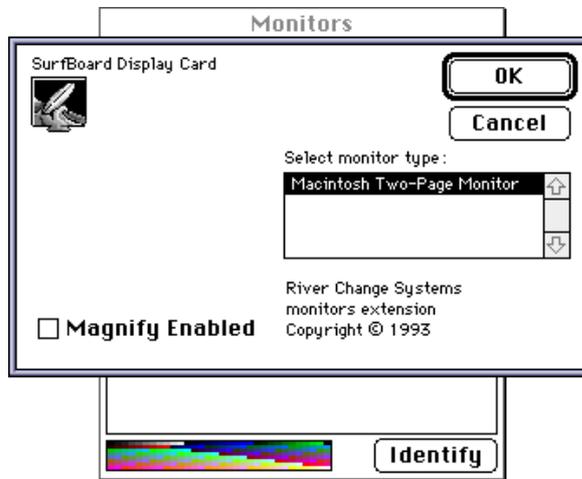
Figure 8-4 shows an example of the Monitors control panel.

Figure 8-4 The Monitors control panel



If more than one video card is installed in the computer, the Monitors control panel shows all of the connected monitors. When the user selects one monitor, then clicks the Options button, the Monitors control panel displays the Options dialog box for that monitor. When you provide a monitors extension for the Monitors control panel, the controls you add appear in this dialog box.

Figure 8-5 shows an example of an Options dialog box for the SurfBoard video card. The OK and Cancel buttons are standard for all Options dialog boxes. In this example, the developers of the SurfBoard video card have provided a monitors extension that adds two items to the the Options dialog box: the Magnify Enabled checkbox and static text listing the manufacturer's name.

Figure 8-5 An Options dialog box for the SurfBoard video card

A monitors extension file must contain these four resources:

- n A card ('card') resource. This resource contains a Pascal string identical to the name stored in the declaration ROM of the video card. You can include as many card resources as you like, so that one extension file can handle several types of video cards.
- n A monitor ('mnr') code resource. This resource carries out the functions of your monitors extension.
- n A rectangle ('RECT') resource. This resource describes the size and shape of the area that your controls occupy.
- n An item list ('DITL') resource specifying the items in your monitors extension. You can also add additional controls, separated from other controls by a horizontal line, for the benefit of advanced users (superusers).

Your monitors extension file can also include any of the following resources:

- n One or more members of an icon family ('ICN#', 'ics#', 'icl8', 'icl4', 'ics8', and 'ics4'), each with resource ID -4064, that define an icon for your video card. If you provide any of these resources, the Monitors control panel displays the appropriate icon from the icon family in the upper-left corner of the Options dialog box.
- n Additional icon family resources to provide a unique icon for your monitors extension file.
- n A version ('vers') resource. This resource provides version information for your monitors extension.
- n A string list ('STR#') resource defining one or more video card names. If you want the Options dialog box to display a name that is different from the one in the declaration ROM of the card, define the alternate name in an 'STR#' resource.

Control Panels

- n One or more gamma table (' gamma ') resources. Here you can include gamma tables that allow your video card to provide the most accurate colors possible.
- n A file reference (' FREF ') resource. This resource associates icons with your monitors extension file; the Finder uses this information to display the icon for your monitors extension file.
- n A bundle (' BNDL ') resource. This resource groups together the monitor extensions' signature, icon, and file reference resources.
- n A system extension (' INIT ') resource. Although this resource acts independently of other resources in the file, it should be related to the monitors extension.
- n A signature resource (of type ' STR ').

Creating Control Panel Files

This section describes how to create your control panel's resources, including the code resource that implements the control panel. This section discusses how to

- n define the user interface for your control panel
- n create resources for your control panel, including those that define
 - n control panel rectangles
 - n the item list resource
 - n icons
 - n the machine resource
 - n the file reference, bundle, and signature resources
- n specify the font of static text in your control panel
- n write a control panel function

Before you begin, consider whether the feature that you have in mind is best governed by a control panel. It should be a systemwide feature amenable to manipulation by the user, who would use the control panel only occasionally to set or change preferences. If you find that you need special menus or nested dialog boxes to implement your control panel, create a small application instead.

Defining the User Interface for a Control Panel

The user interface for a control panel consists of the display area defined by the dialog box and its controls, including checkboxes, buttons, static text, editable text, and user items. In addition, you need to provide an icon for your control panel file, for display by the Finder. A control panel can open in a modeless dialog box of any size, limited only by the screen display.

Control Panels

Your control panel's display area can consist of one or more rectangles; you determine the display area by defining the rectangles and their positions. You specify these values in your control panel's rectangle positions ('nrct') resource. These rectangles essentially determine the size of the dialog box. The Finder calculates the boundaries of the dialog box from the coordinate values you specify in your rectangle positions resource.

When deciding on the size and number of your rectangles, consider the number and placement of the buttons, checkboxes, text, and other items in your control panel. Allow enough space for the user to distinguish them easily. Because control panels are generally used only occasionally, make the interface as simple as possible. If you choose the default settings well, the user should seldom need to use your control panel.

Figure 8-6 shows the user interface for the River control panel used as an example throughout this chapter. It governs certain features of River, a screen saver system extension.

Figure 8-6 The River control panel interface



In System 7, you can include a font information resource that specifies the font in which the Finder displays your control panel's static text items. (For information on creating a font information resource, see "Specifying the Font of Text in a Control Panel" on page 8-23.) Choose a font that is easy to read. In System 7, the control panel interface allows ample space for larger point sizes; Apple recommends 12-point Chicago.

If you don't include a font information resource, the Finder uses the default application font for static text items. For Roman scripts, this is 9-point Geneva. (The static text of the River control panel illustrated in Figure 8-6 is 12-point Chicago because this control panel provides a font information resource for this purpose.) Note that the Finder uses the system font to draw text strings that you define as part of a control item in your item list; for Roman scripts, this is 12-point Chicago.

Control Panels

If your control panel runs in both System 7 and System 6 but you wish to display your control panel's static text in 12-point Chicago, you can define the text as user items. See "Defining Text in a Control Panel as User Items" on page 8-24 for details.

If you wish, you can create an icon family to specify the icon that the Finder uses to represent your control panel file. The icon family resources are 'ICN#', 'ics#', 'icl8', 'icl4', 'ics8', and 'ics4'.

The icons for a control panel file are square and include a horizontal or vertical slider along with a graphic representing the feature governed by the control panel. Figure 8-7 shows an icon for the River control panel file.

Figure 8-7 An icon for the River control panel file



See *Macintosh Human Interface Guidelines* for more information on designing an icon. For complete information on designing a dialog box, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Creating a Control Panel's Resources

The following sections describe the required and optional resources that you supply for your control panel. The first section contains general information that applies to all of the individual resources. Later sections discuss each of the required resources and some optional resources.

Resource IDs for Control Panels

Every resource has a resource ID. With one exception, all resource IDs for control panel resources, including standard resources and resources you define yourself, must be in the range of -4064 through -4033. The exception is the resource for the icon help balloon ('hfdx') resource, whose resource ID is -5696.

Of this range, resource IDs from -4064 through -4049 are reserved for standard resources and some optional resources.

You can assign resource IDs in the range -4048 through -4033 to any private resources that you define for your control panel.

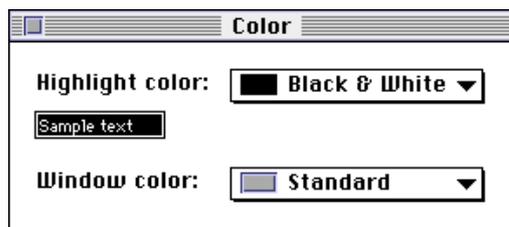
Note

You can use a high-level tool such as the ResEdit application, which is available through APDA, to create your resources. (See *ResEdit Reference* for details on using ResEdit.) You can also use the Rez utility. u

Defining the Control Panel Rectangles

Your control panel can consist of one rectangle, as in Figure 8-8, or several (see Figure 8-2 on page 8-6 and Figure 8-6 on page 8-13). You define these rectangles in a rectangle positions ('nrect') resource. You specify in this resource the number of rectangles for your control panel and a list of the coordinates for each rectangle. You must specify a resource ID of -4064 for a rectangle positions resource.

Figure 8-8 The Color control panel



In the rectangle positions resource you specify a rectangle's coordinates in this order: top, left, bottom, and right. Although you can define a control panel of any size (limited only by the screen display), you must specify the coordinates (-1,87) as the origin (upper-left point) of the upper-left rectangle. To provide for backward compatibility with the Control Panel desk accessory, the Finder accepts only these coordinates as the origin of a control panel. If you are designing for System 7 only, you can extend the bottom and right edges of a control panel as far as you like. If you want your control panel to run in System 7 and previous versions of system software, you must limit your control panel's size to the area bounded by (-1,87,255,322). These are the coordinates used by the Control Panel desk accessory.

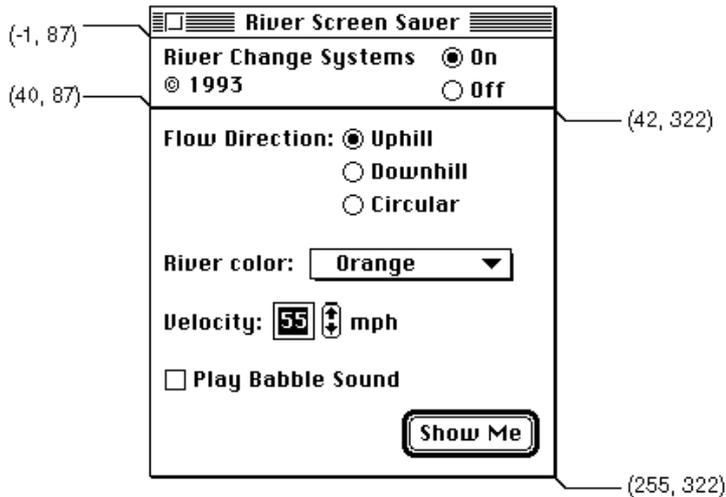
The Control Panel Desk Accessory

In System 6, the Control Panel desk accessory is a single interface shared by all control panels. It has two parts: a scrollable list of icons representing the control panels a user can open as part of the desk accessory and a display area of fixed size. If you want to make your control panel compatible with the Control Panel desk accessory, it must fit in this area. The Control Panel desk accessory acts as a driver interfacing with and managing the control panels whose icons it displays. All of the control panels represented by icons in the scrollable list share the same display area. For this reason, a user can open only one control panel at a time. u

If you want to make your control panel backward compatible, remember that the Control Panel desk accessory draws a frame that is 2 pixels wide around each rectangle. To join two parts of a panel neatly, overlap their rectangles by 2 pixels on the side where they meet.

Figure 8-9 shows the coordinates of the two rectangles that make up the River control panel. Because the River control panel has relatively few items, they fit well within the space constraints imposed by the Control Panel desk accessory. Thus, this control panel can run in both the Finder in System 7 and the Control Panel desk accessory in System 6.

Figure 8-9 Coordinates defining the rectangles of the River control panel display area



Listing 8-1 shows the Rez input for the rectangle positions resource that specifies the rectangles for the River control panel.

Listing 8-1 Rez input for a rectangle positions list ('nrct') resource

```
resource 'nrct' (-4064, purgeable) {
  /*array RectArray: 2 elements*/
  /*[1]*/
  {-1, 87, 42, 322},
  /*[2]*/
  {40, 87, 255, 322}
}
};
```

If you define two or more rectangles that together do not form a complete square or rectangle in relation to the bounding dialog box that the Finder creates, the Finder fills in any blank space on the control panel with a gray pattern.

Note

In System 6, the Control Panel desk accessory first fills in the area defined by the coordinates (-1,87,255,322) with a gray background pattern. It then creates white areas corresponding to the rectangles you define. In these, it draws the items of your control panel. The Control Panel desk accessory outlines the rectangles with a 2-pixel-wide frame. u

Creating the Item List Resource

You define the items in your control panel and their positions within its rectangles using an item list ('DITL') resource. These items can include static text, buttons, checkboxes, radio buttons, editable text, the resource IDs of icons and QuickDraw pictures, and the resource IDs of other types of controls, such as pop-up menus. You must specify a resource ID of -4064 for your control panel's item list resource.

An item list contains a display rectangle for each item. A display rectangle determines the size and location of the item. You must specify the coordinates of an item's display rectangle relative to the origin of the control panel's upper-left rectangle.

Recall that the origin (the point at the extreme upper left) of your control panel must coincide with the coordinates (-1,87). In the Control Panel desk accessory, the origin is at the upper left of the rectangle containing the scrollable list of icons, to the left of the display area. A 2-pixel-wide frame borders the rectangle containing the scrollable list of icons.

Listing 8-2 shows the item list resource for the River control panel. Notice that the item list includes a static text item (item 2) giving the control panel's name and copyright. The upper-left point of the display rectangle for the static text lies at the coordinates (4,95).

In Listing 8-2, some items are defined as enabled and some as disabled. By specifying each item in the item list as enabled or disabled, you inform the Dialog Manager whether or not to report user clicks in the item.

Depending on the type of item, you usually provide a text string or a resource ID for the item.

Note that text in a control panel is defined either as part of a control (such as labels for buttons, checkboxes, radio buttons, and pop-up menus), or as separate items (static text, editable text, or user items). For example, the text "River color" is defined as part of a pop-up control in a separate menu resource and the text "mph" is defined as a static text item.

The item list resource for the River control panel defines text that is not provided by a control as static text items; in addition to the product name, these static text items include "Flow Direction:" and "Velocity:" (see Figure 8-6 on page 8-13). The item list resource defines one editable text item, setting the default text for this item to 55. It also defines the editable text item as disabled. If you define an editable text item as disabled, the Dialog Manager and TextEdit handle user input in the editable text item.

Control Panels

IMPORTANT

If you want to use a font other than the default application font for your control panel's text *and* you want your control panel to run in the Control Panel desk accessory of System 6, you must define the text as user items instead of static text items. For more information on this, see "Defining Text in a Control Panel as User Items" on page 8-24. u

In Listing 8-2, the first item in this resource is an enabled button labeled "Show Me." This is the River control panel's default button. (The Control Manager positions the label inside the button and draws it using the system font.) Notice that the outline around the button, which identifies it as the default button, is defined as a separate item (a disabled user item) toward the end of the listing.

All of the other controls with which the user interacts are defined as enabled—the On and Off radio buttons, the radio buttons beside the label Flow Direction, the Play Babble Sound checkbox, and the River color pop-up control. When these controls are active, the user can click them, changing settings and making selections. The up and down arrows are defined as enabled user items, and the item list resource includes a picture item that refers to a resource containing a QuickDraw picture of the arrows. Finally, the item list resource includes a help item referencing the resource ID that defines the help balloons for the River control panel.

Listing 8-2 Rez input for an item list ('DITL') resource

```
resource 'DITL' (rControlPanelDialog, purgeable) {
    { /*array: 18 elements*/
    /*[1]*/
    {219, 237, 239, 308},
    Button {          enabled,          "Show Me"          },
    /*[2]*/
    {4, 95, 44, 247},
    StaticText {     disabled,          "River Change Systems\n© 1993" },
    /*[3]*/
    {2, 254, 21, 302},
    RadioButton {   enabled,          "On"                },
    /*[4]*/
    {22, 254, 40, 302},
    RadioButton {   enabled,          "Off"               },
    /*[5]*/
    {51, 95, 70, 196},
    StaticText {     disabled,          "Flow Direction:"   },

    /*[6]*/
    {50, 197, 68, 303},
    RadioButton {   enabled,          "Uphill"            },
```

CHAPTER 8

Control Panels

```
    /*[7]*/
    {69, 197, 87, 303},
    RadioButton { enabled,          "Downhill"          },
    /*[8]*/
    {88, 197, 106, 303},
    RadioButton { enabled,          "Circular"          },
    /*[9]*/
    {157, 95, 178, 156},
    StaticText { disabled,          "Velocity:"        },
    /*[10]*/
    {156, 162, 172, 180},
    EditText { disabled,            "55"                },
    /*[11] (up arrow)*/
    {150, 184, 162, 201},
    UserItem { enabled,              },
    /*[12] (down arrow)*/
    {163, 184, 175, 201},
    UserItem { enabled,              },
    /*[13] (picture of up/down arrows)*/
    {150, 184, 175, 201},
    Picture { disabled,              -4048                },
    /*[14]*/
    {157, 202, 176, 242},
    StaticText { disabled,            "mph"                },
    /*[15] (outline around default button)*/
    {212, 231, 247, 314},
    UserItem { disabled,              },
    /*[16]*/
    {188, 95, 208, 241},
    Checkbox{ enabled,                "Play Babble Sound"  },
    /*[17] (title & menu items defined by menu w/res ID mPopUp)*/
    {122, 92, 142, 297},
    Control { enabled,                mPopUp                },
    /*[18] get help balloon information from 'hdlg' resource*/
    {0,0,0,0},
    HelpItem { disabled,
                HMScanhdlg /*scan resource type-'hdlg' or 'hrct'*/
                {-4064}
            }
    }
};
```

For complete information on creating an item list resource, see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Defining the Icon for a Control Panel

You create an icon family to specify the icon that the Finder uses to represent your control panel file. The icon family resources are 'ICN#', 'ics#', 'icl8', 'icl4', 'ics8', and 'ics4'. You must specify a resource ID of -4064 for the icon family resources of a control panel and mark these resources as purgeable. If you provide the complete icon family, the Finder displays the appropriate icon family member according to the bit depth of the monitor. For more information on these icons, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Specifying the Machine Resource

When the user opens your control panel, the Finder reads your machine ('mach') resource from your control panel file. Depending on the value you specify in the machine resource, the Finder takes one of two actions: (1) calls your control device function, directing your function to check the current hardware and software configuration to determine whether your control panel can run on the current system; or (2) performs the check itself. You must specify a resource ID of -4064 for a machine resource.

The machine resource consists of a hard mask and a soft mask. The Finder handles the check if you set these masks to values indicating that your control panel runs on all systems or to values representing the requirements for your control panel; the Finder checks the current configuration in the latter case. If the Finder handles the check, it never calls your control device function with a `macDev` message; instead, the Finder calls your function for the first time with an initialization message. If the Finder determines that your control panel cannot run on the current system, the Finder displays an alert box to the user and does not open the control panel. (In System 6, the Control Panel does not display the icon for a control panel file if the machine resource indicates the control panel cannot run on the current system.)

If you set the hard mask to \$FFFF and the soft mask to \$0000, indicating your control device function performs its own requirements check, the Finder calls your function with a `macDev` message once only, and this is the first call the Finder makes to your function. (See "Determining If a Control Panel Can Run on the Current System" on page 8-29 for a discussion of how to handle a `macDev` message.)

Table 8-1 shows the values you use to set the machine resource masks.

Control Panels

Table 8-1 Possible settings for the machine resource masks

Soft mask	Hard mask	Action
\$0000	\$FFFF	The Finder calls this control device function with a <code>macDev</code> message, and the function must perform its own hardware and software requirements check.
\$3FFF	\$0000	This control panel runs on Macintosh II systems only.
\$7FFF	\$0400	This control panel runs on all systems with an Apple Desktop Bus (ADB).
\$FFFF	\$0000	This control panel runs on all systems.

Listing 8-3 shows the Rez input for a machine resource. The values in this machine resource indicate to the Finder that the control panel performs its own hardware and software requirements check.

Listing 8-3 Rez input for a machine ('mach') resource

```
resource 'mach' (-4064, purgeable) {
    0xFFFF,          /*hard mask*/
    0                /*soft mask*/
};
```

Note

The machine resource allows the Finder to cache information about each control panel. The user can force the Finder to rebuild the cache by pressing Command-Option while opening the control panel. u

Creating the File Reference, Bundle, and Signature Resources

You must create a file reference resource, a signature resource, and a bundle resource to enable the Finder to display the icon for your control panel. You must specify a resource ID of -4064 for both a bundle resource and a file reference resource.

The file reference resource specifies a file type (for a control panel, 'cdev'), the local resource ID of an icon list resource, and an empty string. The local ID maps the file type ('cdev') to your icon list resource that is assigned the same local ID in the bundle resource. Listing 8-4 shows the file reference resource for the River control panel.

Listing 8-4 Rez input for a file reference ('FREF') resource

```
resource 'FREF' (-4064, purgeable) {
    'cdev', 0, ""
};
```

Control Panels

The Finder uses the signature resource with the bundle resource to establish your control panel's identity. You define a signature resource as a string resource (that is, a resource of type 'STR ') and specify as its resource type a unique four-character sequence that has the same value as your control panel's creator type. A signature resource has a resource ID of 0.

The signature resource contains a string that identifies your control panel; typically the string specifies the name, version number, and release date of your control panel.

Listing 8-5 shows the River control panel's signature resource, which has a signature of 'rivr', in Rez input format.

Listing 8-5 Rez input for a signature resource

```
type 'rivr' as 'STR ';
resource 'rivr' (0, purgeable) {
    "River Control Panel 1.0"
};
```

A bundle ('BNDL') resource associates all of the resources that the Finder uses for your control panel. It associates your control panel file and your control panel's signature with its icon. The Finder requires the information in the bundle resource in order to display icons for your control panel. In the bundle resource, you must assign a local ID to your icon list resource that matches the local ID you assigned inside the corresponding file reference resource. In the bundle resource shown in Listing 8-6, local ID 0 is assigned to the icon list resource with a resource ID -4064, which maps the icon defined for the River control panel to the control panel file.

Listing 8-6 Rez input for a bundle ('BNDL') resource

```
resource 'BNDL' (-4064, purgeable) {
    'rivr', 0,
    { 'ICN#', {0, -4064},
      'FREF', {0, -4064}
    }
};
```

(See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information on how to create file reference, signature, and bundle resources.)

Providing Additional Resources for a Control Panel

In addition to providing required resources, you can supply optional resources for your control panel. For example, you can supply resources to store the settings of controls, text strings, or font information. The River control panel stores its controls' settings in a resource that it defines.

Control Panels

If you wish, you can provide help balloon resources. For example, you can include a resource to define a help balloon for your control panel's icon in the Finder. The resource type of an icon help balloon resource is `'hfdrr'`, and its resource ID is `-5696`.

This is the only control panel resource whose resource ID is outside the range of `-4064` through `-4033`.

You can also include help balloon resources for specific items or areas of your control panel. For example, you might include a help balloon resource to explain how to use a control. For this purpose, you supply a resource of type `'hdlg'` or `'hrct'` with a resource ID of `-4064`. For information on how to create help balloon resources, see the chapter "Help Manager" in this book.

If you define any other types of resources for your control panel, you must assign them resource IDs in the range `-4048` through `-4033`.

Specifying the Font of Text in a Control Panel

A control panel typically contains uneditable text that is part of a control item or defined as static text. See Listing 8-7 on page 8-24 for examples.

The Finder uses the default application font to draw control panel items that you define as static text. However, you can specify that a different font be used for this purpose.

There are two ways to do so. The easiest way is to define a font information (`'finf'`) resource. This is the method you should use if you intend your control panel to run in System 7 only.

If you want your control panel to be compatible with the Control Panel desk accessory, you cannot use this method because the Control Panel desk accessory does not recognize font information resources. In this case, you can use an alternative method, which entails defining your control panel's static text as user items, setting the font, and drawing the text. This section explains both methods.

You can also specify the font to be used for each item by creating an item color table (`'ictb'`) resource whose entries correspond to the items in your item list. However, you cannot use this method in System 6, because the Control Panel desk accessory appends your control panel's item list to its own. For more information about the item color table (`'ictb'`) resource, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Creating a Font Information Resource

You create a font information (`'finf'`) resource to specify the font in which the Finder displays your control panel's static text. You include the font information resource in your control panel file, and the Finder reads this resource when it opens your file. You must use the resource ID `-4049` for a font information resource.

In the font information resource, you specify the font ID number, the font style, and its size. The Finder sets the graphics port's `txFont`, `txFace`, and `txSize` fields to the values you specify, and QuickDraw draws the text using these values.

Defining Text in a Control Panel as User Items

If you want to specify the font for your control panel's text and also want your control panel to run in both System 6 and System 7, you can define your control panel's text as disabled user items rather than as disabled static text items. Your control device function must call `QuickDraw` to set the graphics port fields for the font and its characteristics, and then draw the text at initialization and in response to update events. See "Handling Text Defined as User Items" on page 8-43 for more information.

For these user items, you can define a string list ('STR#') resource to store the text strings that make up your text. Your control device function can read the text from the string list resource and store the text in a data structure in your control device function's private storage. If you do this, then your control device function can read the values from its private storage whenever it needs to update user items.

Listing 8-7 shows a part of the River control panel's item list with the control panel's text defined as user items. Because the user does not need to read the product title and copyright regularly to interact with the control panel, the control panel defines this string as a static text item; the Finder draws this text string only in 9-point Geneva. The control panel defines all other text strings as user items, and the control device function sets the font and draws those user items containing text.

Listing 8-7 A control panel's static text defined as user items

```
resource 'DITL' (rControlPanelDialog, purgeable) {
    { /* array DITLarray: 18 elements */ }
    /* . . . */
    /* [2] */
    {4,95,44,247},
    StaticText { disabled, "River Change Systems\n© 1993"
},
    /* . . . */
    /* [5] */
    {51, 95, 70, 196},
    UserItem { disabled, /*Flow Direction:*/ },

    /* . . . */
    /* [9] */
    {157, 95, 178, 156},
    UserItem { disabled, /*Velocity:*/ },
    /* . . . */
```

Control Panels

```

    /* [14] */
    {157, 202, 176, 242},
    UserItem {      disabled,      /*mph*/      },
    /* . . . */
    }
};

```

Writing a Control Panel Function

A control panel requires a control device ('cdev') code resource, which contains the code that implements the feature your control panel provides. The first piece of code in this resource must be a control device function that adheres to a defined interface. When the user opens your control panel, the Finder loads your code resource (of type 'cdev') into memory.

The Finder calls your control device function, requesting it to perform the action indicated by the message parameter, in response to events and the user's interaction with your control panel. Your control device function should perform the requested action and return a function result to the Finder. Your control device function should return as its function result either a standard value indicating that it has not allocated storage, a handle to any storage it has allocated, or an error code. Here is how you declare a control device function:

```

FUNCTION MyCdev(message, item, numItems, CPrivateValue: Integer;
                VAR theEvent: EventRecord;
                cdevStorageValue: LongInt;
                CPDialog: DialogPtr): LongInt;

```

The message parameter can contain any of the values defined by these constants:

```

CONST
    macDev      = 8;  {determine whether control panel can run}
    initDev     = 0;  {perform initialization}
    hitDev      = 1;  {handle click in enabled item}
    updateDev   = 4;  {respond to update event}
    activDev    = 5;  {respond to activate event}
    deActivDev  = 6;  {respond to control panel becoming inactive}
    keyEvtDev   = 7;  {respond to key-down or auto-key event}
    undoDev     = 9;  {handle Undo command}
    cutDev      = 10; {handle Cut command}
    copyDev     = 11; {handle Copy command}
    pasteDev    = 12; {handle Paste command}
    clearDev    = 13; {handle Clear command}
    nulDev      = 3;  {respond to null event}
    closeDev    = 2;  {respond to user closing control panel}

```

Control Panels

These constants (as specified in the `message` parameter) indicate that your control device function should perform the following actions:

- n `macDev`. Determine whether the control panel can run on the current system, and return a function result of 1 if it can and 0 if it cannot.
- n `initDev`. Perform initialization.
- n `hitDev`. Handle a click in an enabled item.
- n `updateDev`. Update any user items and redraw any controls that are not standard dialog items handled by the Dialog Manager.
- n `activDev`. Respond to your control panel becoming active by making the default button and any other controls in your control panel active.
- n `deActivDev`. Respond to your control panel becoming inactive by making the default button and any other controls in your control panel inactive.
- n `keyEvtDev`. Handle a key-down or auto-key event.
- n `undoDev`. Handle an Undo command.
- n `cutDev`. Handle a Cut command.
- n `copyDev`. Handle a Copy command.
- n `pasteDev`. Handle a Paste command.
- n `clearDev`. Handle the Clear command.
- n `nulDev`. Handle a null event by performing any idle processing.
- n `closeDev`. Handle a click in the close box by terminating, after disposing of any handles and pointers created by your function.

The control device function that implements the River control panel used as an example in this chapter shows one way of handling messages from the Finder. In this scenario, the user sets the screen saver's characteristics using the River control panel. The River control panel (`'cdev'`) file includes a system extension that displays the screen saver when the user signals it to do so. The River control panel uses the system extension to display the screen saver using the current settings whenever the user clicks the panel's default button (Show Me). (See Figure 8-6 on page 8-13.)

The River control device function reads control settings from a resource stored in its preferences file, which is stored in the Preferences folder, and writes new values to that file at certain points after the user changes control settings. The control device function alerts the system extension of changes in the preferences file, and the system extension gets the new values to use from the preferences file.

In addition to the required resources, the River control device function uses a number of private resources that are included in the control panel file.

Listing 8-8 shows the River control panel's control device function, called `main`. To respond to requests from the Finder, the function uses a `CASE` statement that handles each type of message sent by the Finder.

The remainder of this section discusses each of these messages in detail and includes code showing how the River control panel processes the messages.

CHAPTER 8

Control Panels

Listing 8-8 A control device function

```
UNIT RiverCP;
INTERFACE
    {include a Uses statement if your programming environment requires it}
CONST
    kShowMe                = 1;
    kOnRadButton           = 3;
    kOffRadButton          = 4;
    kUphillRadButton       = 6;
    kDownhillRadButton     = 7;
    kCircularRadButton     = 8;
    kVelocityEditText      = 10;
    kUserItemUpArrow       = 11;
    kUserItemDownArrow    = 12;
    kPict                  = 13;
    kUserItemButtonOutline = 15;
    kBabbleCheckBox        = 16;
    kRiverColorMenu        = 17;
TYPE
    MyRiverStorage =
        RECORD
            err:           LongInt;
            count:        LongInt;
            settingsChanged: Boolean;
        END;
    MyRiverStoragePtr = ^MyRiverStorage;
    MyRiverStorageHndl = ^MyRiverStoragePtr;
FUNCTION main (message, item, numItems, CPrivateValue: Integer;
              VAR theEvent: EventRecord; cdevStorageValue: LongInt;
              CPDialog: DialogPtr): LongInt;
IMPLEMENTATION
FUNCTION main;
{any support routines used by your control panel function}
VAR
    myRiverHndl:           MyRiverStorageHndl;
    initDevOrMacDevMsg:   Boolean;
    okToRun:              LongInt;
    cpMemError:           Boolean;
BEGIN
    cpMemError := MyRoomToRun(message, cdevStorageValue);
```

CHAPTER 8

Control Panels

```
IF cpMemError THEN      {an error occurred or there isn't enough memory }
  main := cdevMemErr    { to run, return immediately}
ELSE {handle the message}
BEGIN
  IF (message <> macDev) AND (message <> initDev) THEN
    myRiverHndl := MyRiverStorageHndl(cdevStorageValue);
  CASE message OF
    macDev: {check machine characteristics}
      BEGIN
        MyCheckMachineCharacteristics(okToRun);
        main := okToRun;
      END;
    initDev: {perform initialization}
      MyInitializeCP(cdevStorageValue, CPDialog, myRiverHndl);
    hitDev: {user clicked dialog item}
      BEGIN
        item := item - numItems;
        MyHandleHitInDialogItem(item, cdevStorageValue,
                                CPDialog, myRiverHndl);
      END;
    activDev: {control panel is becoming active}
      MyActivateControlPanel(cdevStorageValue, CPDialog,
                             myRiverHndl, TRUE);
    deActivDev: {control panel is becoming inactive}
      MyActivateControlPanel(cdevStorageValue, CPDialog,
                             myRiverHndl, FALSE);
    updateDev: {update event -- draw any user items}
      MyUpdateControlPanel(cdevStorageValue, CPDialog, myRiverHndl);
    cutDev, copyDev, pasteDev, clearDev: {editing command}
      MyHandleEditCommand(message, CPDialog);
    keyEvtDev: {keyboard-related event}
      MyHandleKeyEvent(theEvent, CPDialog, message);
    nulDev: {null event -- perform idle processing}
      MyHandleIdleProcessing(cdevStorageValue, CPDialog, myRiverHndl);
    closeDev: {user closed control panel, release memory before exiting}
      MyCloseControlPanel(myRiverHndl, cdevStorageValue);
  END; {of CASE}
  IF message <> macDev THEN
    main := LongInt(cdevStorageValue);
  END; {of handle message}
END; {of main program}
END.
```

Control Panels

When the Finder first calls your control device function, the current resource file is set to your control panel ('cdev') file, the current graphics port is set to your control panel's dialog box, and the default volume is set to the System Folder of the current startup disk. Your control device function must preserve all of these settings.

Although the Finder intercedes with the system software and performs services on behalf of your control device function, it is your control device function's responsibility to detect and, if possible, recover from any error conditions. To avoid a memory error condition, your function should ensure that enough memory is available to handle the message from the Finder. On entry, the `main` function calls its `MyRoomToRun` procedure to perform this check.

The next sections describe how to handle each message passed in the `message` parameter.

Determining If a Control Panel Can Run on the Current System

If you want your control device function to determine if your control panel can run on the current system, specify the values in your machine resource accordingly (see Table 8-1 on page 8-21). In this case, the Finder calls your function for the first time with a `macDev` message. The Finder calls your control device function with a `macDev` message only once.

In response to the `macDev` message, your control device function can check the hardware configuration of the current system. As necessary, your control device function should determine which computer it is being run on, what hardware is connected, and what is installed in the slots, if there are slots. The application-defined `MyCheckMachineCharacteristics` procedure, used in Listing 8-8 on page 8-27, performs these checks for the River control panel. Your control device function should return either a 0 or a 1 as its function result in response to the `macDev` message. These values have specific meanings in response to a `macDev` message, and the Finder does not interpret them as error codes. If your control panel file can run on the current system, return a function result of 1; if your control panel file cannot run on it, return a function result of 0. If your function returns a result of 0, the Finder does not open your control panel; instead, it displays an alert box to the user.

Note

If your machine resource specifies that your control panel runs on all systems, or if the machine resource identifies the restrictions that apply to your control panel, the Finder does not call your control device function with a `macDev` message. u

Initializing the Control Panel Items and Allocating Storage

If your control panel can run on the current system, the Finder calls your control device function and specifies `initDev` in the `message` parameter. Except for a `macDev` message, your control device function should not process any other messages before it receives and successfully processes an `initDev` message. In response to an `initDev` message, your function should allocate any private storage it needs to implement its

Control Panels

features, initialize the settings of controls in the control panel, and perform any other necessary initialization tasks.

Because control panels cannot use normal global variables to retain information once the control device function returns, the interface between the Finder and the control device function provides a way to preserve memory that your control device function might allocate. If, for example, your control device function allocates memory to save data between calls, you return a handle to the allocated memory as the function result in response to the Finder's `initDev` message. The next time it calls your function, the Finder passes this handle back as the value of the `cdevStorageValue` parameter. After sending an `initDev` message, the Finder always passes to your function the function result previously returned as the value of the `cdevStorageValue` parameter. In this way, the Finder makes the handle available to your function, until your function returns an error code.

When the Finder calls your function with the `initDev` message, it passes the constant `cdevUnset` in the `cdevStorageValue` parameter; this value indicates that your function has not allocated any memory. If you do not create a handle and allocate memory in response to the `initDev` message, you should return this value (`cdevUnset`) as your function result. In this case, the Finder continues to pass this value to your control device function, and your function should continue to return this value until your control device function encounters an error.

Before the Finder calls your function with an `initDev` message, it has already drawn the dialog box and any items defined in your item list resource, except for user items. During initialization, you set the default value for any controls, and, if necessary, draw any user items. You can store the default values for controls in a resource located in a preferences file within the Preferences folder. To initially set the values for your panel's controls (such as radio buttons and checkboxes) and editable text, retrieve the default values from the resource and then use the Dialog Manager's `GetDialogItem` procedure and the Control Manager's `SetControlValue` procedure.

The Finder calls `QuickDraw` to draw the static text for your control panel. `QuickDraw` uses the default application font for this purpose; for Roman scripts, this is 9-point Geneva. For System 7, you can include a font information (`'finf'`) resource in your control panel file to specify a font to be used for static text.

For example, you can use a font information resource to specify 12-point Chicago, which is the recommended font for Roman scripts. You can also use an `'finf'` resource to change the font of static text in control panels localized for other system scripts. If you include an `'finf'` resource, the Finder sets the font, font style, and font size for the graphics port to the values you specify and uses these values to draw any static text. See "Specifying the Font of Text in a Control Panel" on page 8-23 for more information.

Note

The Control Manager uses the system font for text strings that are part of a control item. u

Control Panels

Listing 8-9 shows the `MyInitializeCP` procedure, which the River control device function calls to handle the `initDev` message. This procedure calls the `NewHandle` function to create a handle to a record of type `MyRiverStorage` (see Listing 8-8 on page 8-27). The procedure then initializes the fields of this record. It also calls its own procedure, `MyGetUserPreferenceSettings`, which reads a resource file containing the initial settings for the controls. This resource contains either the original default values or new values set by the user from the control panel.

The `MyInitializeCP` procedure sets initial values for any controls in its control panel. For each control, `MyInitializeCP` calls the Dialog Manager's `GetDialogItem` procedure to get a handle to the control and then calls the Control Manager's `SetControlValue` procedure to restore the last setting of the control. The first time a user uses the control panel, the initial values are the default values; after that, the initial values are those last set by the user. The `MyInitializeCP` procedure restores the last settings of radio buttons and checkboxes, sets the menu item to the last item chosen by the user in pop-up menus, and restores the text that the user last entered in editable text items.

Finally, the `MyInitializeCP` procedure returns in the `cdevStorageValue` parameter a handle to the memory it has allocated. The control device function then returns this value as its function result. In all subsequent calls to the control device function, the Finder passes this value back in the `cdevStorageValue` parameter.

The River control panel uses the memory it allocates to save values indicating that the user has changed a setting. When the user clicks the Show Me button or closes the control panel, the control device function notifies the River screen saver system extension that the settings have changed. The River screen saver then uses the new settings when it displays the river on the screen.

Listing 8-9 Initializing a control panel: Allocating memory and setting controls

```
PROCEDURE MyInitializeCP (VAR cdevStorageValue: LongInt; CPDialog: DialogPtr;
                        VAR myRiverHndl: MyRiverStorageHndl);
VAR
    initOnSetting, initOffSetting, initUphillSetting, initDownhillSetting,
    initCircularSetting, initBabbleSetting, initRiverColorSetting: Integer;
    initVelocityText:      Str255;
    startSel, endSel:      Integer;
    itemType:              Integer;
    itemHandle:             Handle;
    itemRect:               Rect;
```

CHAPTER 8

Control Panels

```
BEGIN
myRiverHndl := MyRiverStorageHndl(NewHandle(Sizeof(MyRiverStorage)));
IF myRiverHndl <> NIL THEN
BEGIN    {initialize fields in myRiver record}
    myRiverHndl^^.count := 0;
    myRiverHndl^^.err := 0;
    myRiverHndl^^.settingsChanged := FALSE;
END;
{set default or saved values for each setting in this control panel-- }
{ usually a control panel reads these values from a resource file}
MyGetUserPreferenceSettings(initOnSetting, initOffSetting,
                            initUphillSetting, initDownhillSetting,
                            initCircularSetting, initBabbleSetting,
                            initRiverColorSetting, initVelocityText,
                            startSel, endSel);
{set the initial values of buttons and other controls using the Dialog }
{ Manager's GetDialogItem & the Control Mgr's SetControlValue procedures}

GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle, itemRect);
SetControlValue(ControlHandle(itemHandle), initOnSetting);

GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle, itemRect);
SetControlValue(ControlHandle(itemHandle), initOffSetting);

GetDialogItem(CPDialog, kUphillRadButton, itemType, itemHandle, itemRect);
SetControlValue(ControlHandle(itemHandle), initUphillSetting);

GetDialogItem(CPDialog, kDownhillRadButton, itemType, itemHandle, itemRect);
SetControlValue(ControlHandle(itemHandle), initDownhillSetting);

GetDialogItem(CPDialog, kCircularRadButton, itemType, itemHandle, itemRect);
SetControlValue(ControlHandle(itemHandle), initCircularSetting);

GetDialogItem(CPDialog, kBabbleCheckBox, itemType, itemHandle, itemRect);
SetControlValue(ControlHandle(itemHandle), initBabbleSetting);

GetDialogItem(CPDialog, kRiverColorMenu, itemType, itemHandle, itemRect);
SetControlValue(ControlHandle(itemHandle), initRiverColorSetting);

GetDialogItem(CPDialog, kVelocityEditText, itemType, itemHandle, itemRect);
SetDialogItemText(itemHandle, initVelocityText);
SelectDialogItemText(CPDialog, kVelocityEditText, startSel, endSel);
cdevStorageValue := Ord4(myRiverHndl);
END;
```

Control Panels

If you define your text items as user items, your control device function must draw the text in response to an `initDev` message. See “Handling Text Defined as User Items” on page 8-43 for details.

Responding to Activate Events

When a control panel is active, your control device function is responsible for making each control active or inactive, as appropriate. For example, your function should draw a bold outline around the control panel’s default button. By contrast, when your control panel is inactive, your control device function should make all its controls inactive, causing the Control Manager to draw them in gray or in grayscale, depending on the bit depth of the monitor. This provides a visual indication to the user that a control panel is inactive, and it distinguishes the active window from inactive ones.

Whenever the Event Manager generates an activate event for your control panel in response to a user action, the Finder intercepts the activate event and calls your control device function with either an `activDev` message or a `deActivDev` message. In either case, the Finder passes to your function, in the parameter `theEvent`, the event record for the activate event and, in the `cdevStorageValue` parameter, a handle to the memory previously allocated by your function.

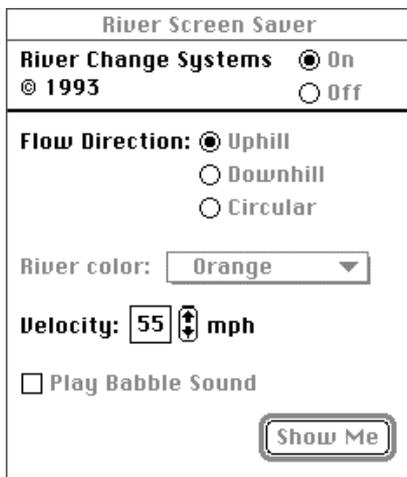
For example, the Finder calls your control device function with an `activDev` message (after sending an `initDev` message) when the user opens your control panel or clicks your inactive control panel after using another control panel or an application. Your function should respond to an `activDev` message by drawing a bold outline around the default button. It should also make the default button and any other controls in your control panel active. You can use the Control Manager’s `HiliteControl` procedure to make a control active or inactive. (See the chapter “Control Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information about `HiliteControl`.)

In general, your function does not need to update user items in response to an activate event, apart from drawing a bold outline around the default button. If, however, your control panel includes a user item that requires updating, such as a clock that shows the current time, your control device function should update that user item.

Control Panels

The Finder calls your control device function with a `deActivDev` message when the user clicks another control panel, runs an application, or otherwise brings another window to the front. In this case, your function should respond by drawing the outline of the default button in gray and making inactive any other controls in your control panel. While a control is inactive, the Control Manager does not respond to mouse events in it. See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to make buttons, radio buttons, checkboxes, and pop-up menus inactive and active in response to activate events. Figure 8-10 shows the River control panel when it is inactive. Note that all of its controls are dimmed.

Figure 8-10 Example of an inactive control panel



The River control device function calls its own procedure to handle both `activDev` and `deActivDev` messages. Listing 8-10 shows the `MyActivateControlPanel` procedure, which either makes the controls active in response to an `activDev` message or inactive in response to a `deActivDev` message.

In response to activate events, this procedure calls the Dialog Manager’s `GetDialogItem` procedure to get a handle to the default button and then calls the Control Manager’s `HiliteControl` procedure to make the control active. To draw the bold outline around the default button, the `MyActivateControlPanel` procedure calls its own procedure, `MyDrawDefaultButtonOutline`. (See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for detailed instructions on drawing an outline around a default button.) The procedure then makes all other controls active.

In response to a `deActivDev` message, the `MyActivateControlPanel` procedure makes all its controls inactive. In addition, it uses its own procedure, `MyDrawDefaultButtonOutline`, to draw a gray outline around the default button.

Note

If the dialog box uses a color graphics port, you can use the `Color QuickDraw` function `GetGray` to return a blended gray based on the foreground and background colors. `u`

Listing 8-10 Responding to an activate event

```

PROCEDURE MyActivateControlPanel (VAR cdevStorageValue: LongInt;
                                CPDialog: DialogPtr;
                                myRiverHndl: MyRiverStorageHndl;
                                activate: Boolean);

VAR
    itemType:      Integer;
    itemHandle:    Handle;
    itemRect:      Rect;
BEGIN
    IF activate THEN
        BEGIN {control panel becoming active}
            {activate the default button (ShowMe) and draw bold outline around it}
            GetDialogItem(CPDialog, kShowMe, itemType, itemHandle, itemRect);
            HiliteControl(ControlHandle(itemHandle), 0);
            MyDrawDefaultButtonOutline(CPDialog, kShowMe);

            {make other controls active}
            GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle, itemRect);
            HiliteControl(ControlHandle(itemHandle), 0);

            GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle, itemRect);
            HiliteControl(ControlHandle(itemHandle), 0);

            GetDialogItem(CPDialog, kUphillRadButton, itemType, itemHandle, itemRect);
            HiliteControl(ControlHandle(itemHandle), 0);

            GetDialogItem(CPDialog, kDownhillRadButton, itemType, itemHandle,
                          itemRect);
            HiliteControl(ControlHandle(itemHandle), 0);

            GetDialogItem(CPDialog, kCircularRadButton, itemType, itemHandle,
                          itemRect);
            HiliteControl(ControlHandle(itemHandle), 0);
        END
    END

```

CHAPTER 8

Control Panels

```
GetDialogItem(CPDialog, kBabbleCheckBox, itemType, itemHandle,
              itemRect);
HiliteControl(ControlHandle(itemHandle), 0);

GetDialogItem(CPDialog, kRiverColorMenu, itemType, itemHandle,
              itemRect);
HiliteControl(ControlHandle(itemHandle), 0);
END
ELSE
BEGIN    {control panel becoming inactive}
    {make the default button inactive and draw gray outline around it}
    GetDialogItem(CPDialog, kShowMe, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);
    MyDrawDefaultButtonOutline(CPDialog, kShowMe);

    {make other controls inactive}
    GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);

    GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);

    GetDialogItem(CPDialog, kUphillRadButton, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);

    GetDialogItem(CPDialog, kDownhillRadButton, itemType, itemHandle,
                  itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);

    GetDialogItem(CPDialog, kCircularRadButton, itemType, itemHandle,
                  itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);

    GetDialogItem(CPDialog, kBabbleCheckBox, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);

    GetDialogItem(CPDialog, kRiverColorMenu, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), 255);
END;
END;
```

Control Panels

Using Multiple Dialog Boxes

The use of nested dialog boxes is not recommended in control panels. If you decide to use them nevertheless, keep in mind that the Finder may send your control device function a `deActivDev` message before your code that displays and initializes the second dialog box completes. This is because when your control device function calls `DialogSelect` to handle an event in a second dialog box, `DialogSelect` issues a call to `GetNextEvent`. In turn, the system software sends to the Finder an activate event instructing it to deactivate the main control panel's dialog box. However, this situation should not cause unusual problems, and your code should handle the `deActivDev` message, then continue its processing for the second dialog box. u

Responding to Keyboard Events

The Finder intercepts all key-down and auto-key events for your control panel. The Finder sends your control device function a keyboard event through the `keyEvtDev` message for all keystrokes except Command-key equivalents. The Finder processes all Command-key equivalents on behalf of your control panel except those that it maps to its own Edit menu commands. The Finder converts these Command-key equivalents to messages and passes them on (as `cutDev`, `copyDev`, `pasteDev`, `undoDev`, and `clearDev` messages) to your control panel for processing. (See "Handling Edit Menu Commands" on page 8-46 for more information.)

Note

In System 6, the Control Panel desk accessory does not convert Command-key equivalents for Edit menu commands to edit messages; instead it passes the Command-key equivalent to your control device function as a `keyEvtDev` message. For backward compatibility, when your control device function receives a `keyEvtDev` message, it should check for Command-key equivalents as follows: it should examine the `modifiers` field and the `message` field of the event record to identify the Command-key equivalent, process it, and set the event record's `what` field to `nullEvent`. In this way, you prevent the Control Panel desk accessory from passing the keystroke to `TextEdit` for further handling. Listing 8-11 illustrates this technique. u

In addition to handling Command-key equivalents, your control device function should respond appropriately when the user presses the Enter key or the Return key. In either case, your function should map the keypress to your control panel's default button, if any, and perform the action corresponding to that button. For instance, the `MyHandleKeyEvent` procedure shown in Listing 8-11 calls its `MyShowMe` routine whenever the user presses Enter or Return. This routine signals the River system extension to display the river on the screen.

Control Panels

Your control device function does not need to process most other keystrokes. The Finder passes keyboard events on to `DialogSelect`, which calls `TextEdit` to handle text entry in editable text items. However, in some cases you might want your function to process the keypress and return the constant `nullEvent` in the `what` field of the event record. For example, if your control panel includes an editable text item that accepts only numeric characters, your function can detect an invalid value, signal the user by beeping, then modify the `what` field to prevent the Finder from passing the event to the Dialog Manager. Listing 8-11 illustrates this technique: the user can enter only numeric values in the Velocity editable text item.

Listing 8-11 Responding to a keyboard event

```
PROCEDURE MyHandleKeyEvent (VAR theEvent: EventRecord; CPDialog: DialogPtr;
                           message: Integer);

VAR
    theChar:    Char;
    itemType:   Integer;
    itemHandle: Handle;
    itemRect:   Rect;
    finalTicks: LongInt;
BEGIN
    {in System 6, you need to check for Command-key equivalents}
    {get the character from the message field of the event record}
    theChar := CHR(BAnd(theEvent.message, charCodeMask));
    IF BAnd(theEvent.modifiers, cmdKey) <> 0 THEN
    BEGIN {Command key down}
        theEvent.what := nullEvent; {change the event to a null event so that }
                                   { TextEdit will ignore it}

        CASE theChar OF
            'X', 'x':
                message := cutDev;
            'C', 'c':
                message := copyDev;
            'V', 'v':
                message := pasteDev;
            OTHERWISE
                message := nulDev; {ignore any other Command-key equivalents}
        END; {of CASE}
        MyHandleEditCommand(message, CPDialog);
    END; {of command-key down}
    CASE theChar OF
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
            ; {valid input, let DialogSelect/TextEdit handle key input}
```

Control Panels

```

OTHERWISE
BEGIN
  IF (theChar = Char(kCRkey)) OR (theChar = Char(kEnterKey)) THEN
  BEGIN {user pressed Return or Enter, map to default button}
    GetDialogItem(CPDialog, kShowMe, itemType, itemHandle, itemRect);

    HiliteControl(ControlHandle(itemHandle), inButton);
    Delay(8, finalTicks);
    HiliteControl(ControlHandle(itemHandle), 0);
    MyShowMe(CPDialog); {perform action defined by default button}
    theEvent.what := nulloEvent;
  END {of Return or Enter}
  ELSE IF (theChar = Char(kDeleteKey)) THEN
    {let DialogSelect/TextEdit handle it}
  ELSE
  BEGIN {invalid input, don't allow this character as input}
    SysBeep(40);
    theEvent.what := nulloEvent;
  END;
  END; {of otherwise}
END; {of CASE}
END;

```

Responding to Mouse Events

When the user clicks any active, enabled controls in your control panel, system software generates a mouse event. The Finder intercepts this event and passes it to your control device function as a `hitDev` message. Your control device function typically changes the setting of the control or performs the appropriate action in response to a `hitDev` message.

Along with the `hitDev` message, the Finder passes three values that your control device function uses to determine which item the user clicked.

- n In the `CPDialog` parameter, the Finder passes a pointer to your control panel's dialog box.
- n In the `item` parameter, the Finder passes the number of the item, as defined in your item list, that the user clicked.
- n In the `numItems` parameter, a value provided for backward compatibility with the Control Panel desk accessory, the value passed depends on the system currently in effect. In System 6, this number is the number of items in the item list of the Control Panel desk accessory. In System 7, the Finder always passes a value of 0 in `numItems`.

In System 6, the Control Panel desk accessory uses the `numItems` parameter to pass the number of items in its own item list. The Control Panel desk accessory appends your control panel's item list to its own. To get the correct number of the clicked item, you need to subtract the number of items in the desk accessory's item list (`numItems`) from

Control Panels

the number passed in the `item` parameter. Although the `numItems` parameter contains 0 in System 7, to maintain backward compatibility, you should always determine an item number by subtracting the value of `numItems` from the value of `item`. If you do so, your control panel can operate correctly with both the Finder and the Control Panel desk accessory. For more information about item lists, see “Creating the Item List Resource” on page 8-17, and the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

The River control device function determines the correct item number in its `CASE` statement before it calls its `MyHandleHitInDialogItem` procedure to handle the `hitDev` message. Here is the code segment, also shown in Listing 8-8 on page 8-27, that determines the item number:

```
hitDev: {user clicked dialog item}
  BEGIN
    item := item - numItems;
    MyHandleHitInDialogItem(item, cdevStorageValue,
                           CPDialog, myRiverHndl);
  END;
```

Listing 8-12 shows the River control panel’s `MyHandleHitInDialogItem` procedure, which takes the appropriate action in response to the item the user clicked. For the Show Me button, the procedure calls its `MyShowMe` procedure, which instructs its system extension to display the River screen saver using any new values.

For the On and Off radio buttons, `MyHandleHitInDialogItem` first calls the Dialog Manager’s `GetDialogItem` procedure to get a handle to each radio button and then the Control Manager’s `GetControlValue` function to determine the current setting. If the radio button clicked was previously off, `MyHandleHitInDialogItem` reverses its setting and also reverses the setting of the radio button that was previously on. If the user clicks any one of the group of radio buttons governing flow direction (Uphill, Downhill, Circular), `MyHandleHitInDialogItem` calls another application-defined routine, the `MyHandleFlowRadioButton` procedure. Although not shown in this listing, this procedure handles each of the three radio buttons, checking whether a button’s value has changed and, if so, resetting the control.

If the user clicked the Play Babble Sound checkbox, `MyHandleHitInDialogItem` reverses its setting.

The River control panel defines two user items that enclose the up arrow and the down arrow. If the user clicks either of these areas, `MyHandleHitInDialogItem` calls its own `MyHandleHitInArrows` procedure to handle this event. The routine either increments or decrements the number displayed in its editable text item accordingly.

The River control panel ignores clicks in any other item, because the Dialog Manager automatically handles clicks in pop-up controls and editable text items.

After handling the `hitDev` message, `MyHandleHitInDialogItem` sets the `settingsChanged` field of the `MyRiverStorage` record. Other routines use this value

to determine if the preferences file needs updating or if its system extension needs to read the preferences file and use the new values when displaying the screen saver.

Listing 8-12 Responding to the user's interaction with controls

```

PROCEDURE MyHandleHitInDialogItem (item: Integer;
                                   VAR cdevStorageValue: LongInt;
                                   CPDialog: DialogPtr;
                                   myRiverHndl: MyRiverStorageHndl);

VAR
    newOnSetting, newOffSetting: Integer;
    newUphillSetting, newDownhillSetting, newCircularSetting: Integer;
    newBabbleSetting: Integer;
    newVelocityText: Str255;
    newRiverColorSetting: Integer;
    itemType: Integer;
    itemHandle: Handle;
    itemRect: Rect;
BEGIN
    CASE item OF
        kShowMe:
            MyShowMe(CPDialog);
        kOnRadButton:
            BEGIN
                {get handle to the On radio button, get its current value, }
                { and then if it was off, change it to on}
                GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle, itemRect);
                newOnSetting := GetControlValue(ControlHandle(itemHandle));
                IF (newOnSetting = 0) THEN
                    BEGIN
                        newOnSetting := 1 - newOnSetting;
                        SetControlValue(ControlHandle(itemHandle), newOnSetting);
                        {get handle to the Off radio button, get its current value, }
                        { and then change it}
                        GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle,
                                     itemRect);
                        newOffSetting := 1 - newOnSetting;
                        SetControlValue(ControlHandle(itemHandle), newOffSetting);
                    END;
                END;
        kOffRadButton:
            BEGIN
                {get handle to the Off radio button, get its current value, }

```

CHAPTER 8

Control Panels

```
{ and then if it was off, change it to on}
GetDialogItem(CPDialog, kOffRadioButton, itemType, itemHandle, itemRect);
newOffSetting := GetCtlValue(ControlHandle(itemHandle));
IF (newOffSetting = 0) THEN
BEGIN
    newOffSetting := 1 - newOffSetting;
    SetControlValue(ControlHandle(itemHandle), newOffSetting);
    newOffSetting := GetCtlValue(ControlHandle(itemHandle));
    {get handle to the On radio button, get its current value, }
    { and then change it}
    GetDialogItem(CPDialog, kOnRadioButton, itemType, itemHandle,
        itemRect);
    newOnSetting := 1 - newOffSetting;
    SetControlValue(ControlHandle(itemHandle), newOnSetting);
END;
END;
kUpHillRadioButton, kDownHillRadioButton, kCircularRadioButton:
    {this routine handles the Flow Direction radio buttons}
    MyHandleFlowRadioButton(item, CPDialog);
kBabbleCheckBox:
    BEGIN
        {get handle to Play Babble Sound checkbox, get its current value, }
        { and then change it}
        GetDialogItem(CPDialog, kBabbleCheckBox, itemType, itemHandle,
            itemRect);
        newBabbleSetting := GetControlValue(ControlHandle(itemHandle));
        newBabbleSetting := 1 - newBabbleSetting;
        SetControlValue(ControlHandle(itemHandle), newBabbleSetting);
    END;
kUserItemUpArrow, kUserItemDownArrow:
    MyHandleHitInArrows(item, CPDialog);
END; {of CASE}
myRiverHndl^^.settingsChanged := TRUE;
END;
```

Responding to Update Events

Whenever the Event Manager generates an update event for your control panel, the Finder intercepts the update event and calls your control device function with an `updateDev` message. Your control device function should perform any updating necessary, apart from the standard dialog item updating that the Dialog Manager performs. An update event gives your control device function the opportunity to redraw user items that might require updating, such as a clock. You should also redraw the outline around your default button in response to an update event. Notice that the `MyUpdateControlPanel` procedure in Listing 8-13 does this by calling its `MyDrawDefaultButtonOutline` procedure, which the control device function also calls in response to an `activDev` or `deActivDev` message. If your control panel has an editable text item, you don't need to include code to make the caret blink. The Dialog Manager calls `TEIdle` for this purpose.

Listing 8-13 Responding to update events

```
PROCEDURE MyUpdateControlPanel (VAR cdevStorageValue: LongInt;
                                CPDialog: DialogPtr;
                                myRiverHndl: MyRiverStorageHndl);

BEGIN
    {draw the outline around the default button on an update event}
    MyDrawDefaultButtonOutline(CPDialog, kShowMe);
END;
```

Handling Text Defined as User Items

If you want to use a font other than the default application font for your control panel's text, you should either include an `'finf'` resource in your control panel file and define your text as static text items or define your text using user items. See "Creating a Font Information Resource" on page 8-23 for details on changing the font using an `'finf'` resource. This section gives details on how to define text using user items. You might want to use this approach so that your control panel can run in the Finder and the Control Panel desk accessory.

If you define the text in your control panel using user items, you need to draw the text in response to an `updateDev` message, just as you would any other user item that requires updating. (You draw the text initially in response to an `initDev` message.)

Control Panels

For each item, this process entails

- n Setting the text font, style, and size fields to be used. (You use the QuickDraw procedures `TextFont`, `TextFace`, and `TextSize` for this purpose.)
- n Positioning the pen where you want to draw the text. You draw the text in the rectangle defined for it in the item list resource. (You can use the QuickDraw procedure `MoveTo` to set the initial location of the pen.)
- n Drawing the text string. (You can use the QuickDraw `DrawString` procedure for this purpose.)

Listing 8-14 shows the `MyDrawText` procedure. The River control device function might use this procedure to draw any text that it defined as user items. First the `MyDrawText` procedure calls the QuickDraw `TextFont`, `TextFace`, and `TextSize` procedures to set the graphics port font to 12-point Chicago.

Then, for each text item, `MyDrawText` calls its own `MyGetUserText` procedure to get the text string and the coordinates of the text string as defined by the display rectangle of the user item. (See “Defining Text in a Control Panel as User Items” on page 8-24 for details about the item list.) Next, `MyDrawText` calls the QuickDraw `MoveTo` procedure to position the pen and QuickDraw’s `DrawString` procedure to draw the text.

Listing 8-14 Drawing text defined as user items

```
PROCEDURE MyDrawText;
VAR
    textForUserItem: Str255;
    textH, textV: Integer;
BEGIN
    TextFont(0);           {set the font to the system font (Chicago)}
    TextFace([]);         {set the text face to normal}
    TextSize(12);         {set the font size to 12-point}
    {get the text and location for the first text string}
    MyGetUserText(kFlow, textForUserItem, textH, textV);
    MoveTo(textH, textV);
    DrawString(textForUserItem);    {draw the text}
    {get the text and location for the next text string}
    MyGetUserText(kVelocity, textForUserItem, textH, textV);
    MoveTo(textH, textV);
    DrawString(textForUserItem);    {draw the text}
    {get the text and location for the next text string}
    MoveTo(textH, textV);
    MyGetUserText(kMph, textForUserItem, textH, textV);
    DrawString(textForUserItem);    {draw the text}
END;
```

Responding to Null Events

Whenever the Event Manager generates a null event for your control panel, the Finder intercepts the event and calls your control device function with a `nulDev` message. Your control device function should respond to a `nulDev` message by performing any needed idle processing. However, your control device function should do minimal processing in response to a null event; for example, it should not refresh control settings.

Responding to the User Closing the Control Panel

When the user closes your control panel, the Finder calls your control device function with a `closeDev` message, signaling it to terminate gracefully. In response to this message, your control device function must dispose of any memory it has allocated, including any pointers or handles it has allocated.

Before your function begins this process, however, it can perform other needed tasks. For example, the River control device function checks whether the user changed the values of any settings. If so, it updates its preferences file to reflect the changes.

Listing 8-15 shows the `MyCloseControlPanel` procedure, which the River control device function calls to handle the `closeDev` message. The `MyCloseControlPanel` procedure checks the `settingsChanged` field of its `MyRiverStorage` record to determine if the user changed the settings (the control device function sets this field whenever the user changes a setting). If necessary, `MyCloseControlPanel` calls a procedure to update the preferences file with the new values stored in the `MyRiverStorage` record. Next, `MyCloseControlPanel` disposes of the memory that the control device function previously allocated by disposing of the handle in the `myRiverHndl` parameter. It then sets the `cdevStorageValue` parameter to 0. The control device function returns this value as its function result.

Listing 8-15 Terminating a control device function when the user closes the control panel

```
PROCEDURE MyCloseControlPanel (myRiverHndl: MyRiverStorageHndl;
                               VAR cdevStorageValue: LongInt);
BEGIN
    {if the user changed any of the settings, }
    { write the new settings to the River preferences file}
    IF myRiverHndl^^.settingsChanged THEN
        MyWriteUserPreferences(myRiverHndl);
    {dispose of any allocated storage}
    IF myRiverHndl <> NIL THEN
        BEGIN
            DisposeHandle(Handle(myRiverHndl));
            cdevStorageValue := 0;
        END;
    END;
```

Handling Edit Menu Commands

Although you cannot implement a menu bar in your control panel, the user can choose the Finder's Edit menu Undo, Cut, Copy, Paste, and Clear commands when working in an editable text item. When the user chooses one of these commands from the Edit menu or presses its Command-key equivalent, the Finder maps the command to a message and calls your control device function with the message. The values in the message parameter for these commands are `undoDev` for Undo, `cutDev` for Cut, `copyDev` for Copy, `pasteDev` for Paste, and `clearDev` for Clear.

Note

In System 6, the Control Panel desk accessory does not convert Command-key equivalents for Edit menu commands to edit messages; instead it passes the Command-key equivalent to your control device function as a `keyEvtDev` message. See "Responding to Keyboard Events" beginning on page 8-37 for details on handling keyboard events, including Command-key equivalents. \cup

Listing 8-16 show the `MyHandleEditCommand` procedure. The River control device function calls this procedure from within its `CASE` statement to handle an edit message. For the Cut, Copy, and Clear commands, `MyHandleEditCommand` calls Dialog Manager routines to perform the desired operation. For the Paste command, `MyHandleEditCommand` first uses its `MyCheckLength` function to ensure that the length of any text to be pasted does not exceed the TextEdit text buffer limit of 32 KB; only then does it call `DialogPaste`. The Dialog Manager calls `TextEdit` to perform the operation.

Listing 8-16 Responding to Edit menu commands

```
PROCEDURE MyHandleEditCommand (message: Integer;
                               CPDialog: DialogPtr);

BEGIN
  CASE message OF
    cutDev:           {use Dialog Manager to cut the text}
      DialogCut(CPDialog);
    copyDev:          {use Dialog Manager to copy the text}
      DialogCopy(CPDialog);
    clearDev:         {use Dialog Manager to clear the text}
      DialogDelete(CPDialog);
    pasteDev:
      BEGIN           {check length, then paste the text}
        IF MyCheckLength(CPDialog) THEN
          DialogPaste(CPDialog);
        END;
      END;
  END; {of CASE}
END;
```

Handling Errors

Your control device function is responsible for detecting and, if possible, recovering from error conditions. If your function cannot recover from an error condition, it must dispose of any memory that it previously allocated, restore the system stack, and return as its function result one of three error codes.

If your control panel encounters an error due to missing resources or lack of memory, your control device function should return `cdevResErr` or `cdevMemErr`. When the Finder receives either of these error codes, it closes the control panel and displays an alert box reporting the problem.

Your control device function should return a generic error code (`cdevGenErr`) for all other errors. When the Finder receives this generic error code, it closes the control panel but does not display an alert box; if it can do so, your function should display an alert box to the user before completing. Your function can also return this error code to signal a missing-resources or lack-of-memory error. Use this error code instead of `cdevResErr` or `cdevMemErr` if you want your function, not the Finder, to display a meaningful error message that directs the user in resolving the problem.

The Finder in System 7 and the Control Panel desk accessory in System 6 respond differently to any error codes that your control panel returns. In System 6, after your control device function terminates, the Control Panel desk accessory fills the area previously occupied by your control panel with the background pattern, in effect dimming it. The Control Panel desk accessory dialog box remains open because the user can use other control panels represented in the icon list. Your control panel's area remains dimmed until the user selects another control panel.

Table 8-2 shows the constants defined for these error codes and the corresponding responses by the Finder and the Control Panel desk accessory.

Table 8-2 Error codes and their meaning

Constant	Value	Meaning
<code>cdevGenErr</code>	-1	<p>Generic error</p> <p>In System 7, the Finder closes your control panel but does not display an alert box to the user.</p> <p>In System 6, the Control Panel desk accessory dims your control panel's area in the Control Panel window and passes 0 in the <code>cdevStorageValue</code> parameter the next time it calls your function.</p>
<code>cdevMemErr</code>	0	<p>Insufficient memory</p> <p>In System 7, the Finder closes the control panel and displays an out-of-memory alert box to the user.</p> <p>In System 6, the Control Panel desk accessory dims your control panel's area in the Control Panel window, displays an out-of-memory alert box to the user, and passes 0 in the <code>cdevStorageValue</code> parameter the next time it calls your function.</p>

continued

Table 8-2 Error codes and their meaning (continued)

Constant	Value	Meaning
<code>cdevResErr</code>	1	<p>Missing resource</p> <p>In System 7, the Finder closes the control panel and displays a missing-resources alert box to the user.</p> <p>In System 6, the Control Panel desk accessory dims your control panel's area in the Control Panel window, displays a missing-resources alert box to the user, and passes 0 in the <code>cdevStorageValue</code> parameter the next time it calls your function.</p>

Creating an Extension for the Monitors Control Panel

This section describes how to create an extension for the Monitors control panel. A monitors extension typically adds controls to the Options dialog box so that the user can set values for one or more features of a video card.

A monitors extension consists of a file of type `'cdev'` that contains the resources for a monitors extension, including a code resource of type `'mnr'`. This code resource, called a monitors extension function, communicates with the Monitors control panel, responding to requests from the Monitors control panel to handle events or perform actions. This section begins with a discussion of the interface components of a monitors extension. Then it describes how to

- n create resources for your monitors extension, including how to define
 - n a card resource to identify your monitors extension and display the name of your video card at the top of the Options dialog box
 - n a rectangle resource to define the area in which to display your video card's controls
 - n an item list resource to define additional items for display in the Options dialog box
- n create and supply optional resources for your monitors extension
- n write a monitors extension function

Control Panels

Before you develop an extension for the Monitors control panel, consider these three important points:

- n You should develop a monitors extension *only* if you are the manufacturer of the video card for which you are providing the feature or features whose values the user can control.
- n There can be only one extension to the Monitors control panel for each video card. Apple Computer, Inc., reserves the right to supply monitors extensions for its own video cards.
- n If the features that you want to implement require an extensive or complex set of controls—for example, if you need to use nested dialog boxes—you should probably write a small application rather than an extension to the Monitors control panel.

Designing the User Interface for a Monitors Extension

When the user clicks the Options button, the Monitors control panel displays the Options dialog box for the selected monitor. The Options dialog box contains standard controls that the Monitors control panel provides, such as the OK and Cancel buttons. Beneath these two buttons is a scrollable list of monitor types if the selected monitor belongs to a family of monitors. Beneath the icon is a scrollable list of gamma tables if the user is a **superuser** (a very knowledgeable user; a user indicates superuser status by pressing the Option key while clicking the Options button). These items are also defined by the Monitors control panel.

If you provide a monitors extension for your video card, the Monitors control panel adds any controls you define beneath the two scrollable lists, if one or both are displayed, or beneath the Cancel button. Figure 8-11 shows the Options dialog box for the Macintosh display card.

Figure 8-11 An Options dialog box with standard controls

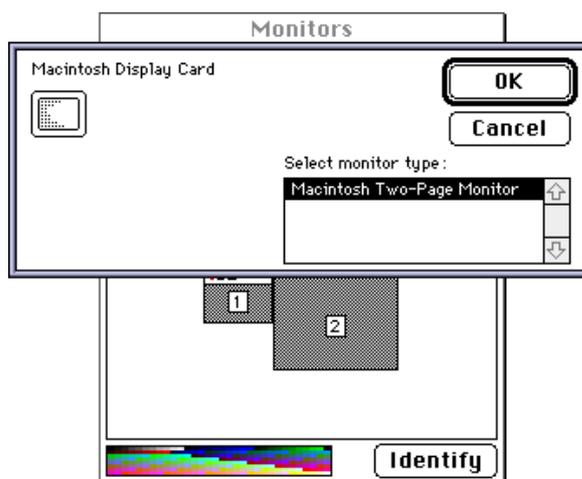
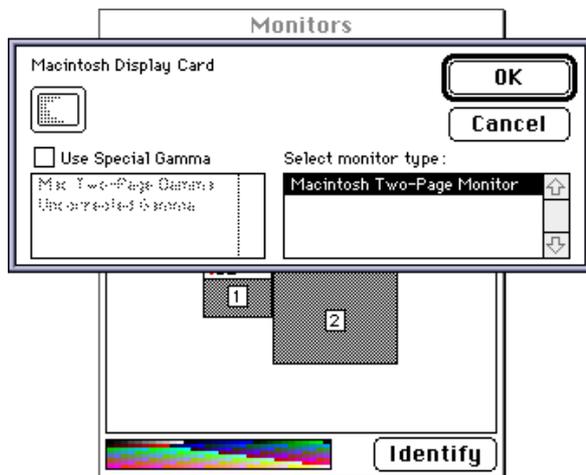


Figure 8-12 shows the Options dialog box for the Macintosh display card as it appears when the user presses the Option key while clicking the Options button.

Figure 8-12 An Options dialog box with superuser controls



To provide the user interface for your video card's feature, you define a rectangle resource of type 'RECT' specifying the amount of space you need to display your controls and an item list resource of type 'DITL' specifying the controls themselves.

At the upper-left corner of the Options dialog box, the Monitors control panel displays the name of your video card and an icon representing it. The Monitors control panel defines the coordinates of these items. You must supply your video card's name in a required card resource of type 'card' (see "Creating a Card Resource for a Monitors Extension" on page 8-51). You can optionally provide one or more members of an icon family (with resource ID -4064) that define an icon for your video card. If you do not provide icon resources with this resource ID for this purpose, the Monitors control panel displays the icon defined in the sResource data structure in the ROM on your video card. If your video card does not supply a default icon in the ROM, the Monitors control panel displays a generic monitors icon.

You can also supply an additional icon family to specify the icon that the Finder uses to represent your monitors extension file. The icon family resources are 'ICN#', 'ics#', 'icl8', 'icl4', 'ics8', and 'ics4'. When creating an icon for a monitors extension, design it so that it is square, except include at the bottom of the icon a tab-like form, indicating that the file the icon represents is an extension. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to create an icon family. Figure 8-13 shows an icon of type 'icl8' for the monitors extension file supplied with the SurfBoard video card.

Figure 8-13 The SurfBoard monitors extension icon

MonitorExtend

If you wish, you can design two sets of controls for your monitors extension: one set for ordinary users and one for superusers. When a user indicates superuser status by holding down the Option key while clicking the Options button in the Monitors control panel, the Monitors control panel notifies your monitors extension function to display the superuser controls. For more information, see “Creating an Item List Resource for a Monitors Extension” beginning on page 8-54.

Creating the Required Resources for a Monitors Extension

This section describes the four required resources that you supply for your monitors extension.

To create these resources, either you can specify the resource description in an input file and compile the resource using a resource compiler, such as Rez, or you can directly create your resources in a resource file using a tool such as ResEdit.

The required resources and their resource IDs are

- n the card ('card') resource: resource ID from -4080 through -4065
- n the rectangle ('RECT') resource: resource ID -4096
- n the item list ('DITL') resource: resource ID -4096
- n the monitor ('mnltr') resource: resource ID -4096

Creating a Card Resource for a Monitors Extension

You create a card resource of type 'card' to identify the monitors extension for your video card and to specify the name of your video card. When the monitor to which your card is connected is the selected one and the user clicks the Options button, the Monitors control panel checks all monitors extension files for a card resource that contains the name of your video card. If it finds a match, the Monitors control panel extends the Options dialog box to display the monitors extension containing the matching card resource. The Monitors control panel also displays, at the very top of the Options dialog box, your video card's name as defined in the card resource. This title indicates to the user that the Options dialog box pertains to your card. For example, Figure 8-11 on page 8-49 shows the Macintosh Display Card name at the top of the Options dialog box. The card resource is required, and its resource ID must be in the range of -4080 through -4065.

Control Panels

Your card resource must contain a Pascal string identical to the name of your video card as specified in the `sResource` data structure in the ROM of the card. (For more information on the `sResource` data structure, see *Designing Cards and Drivers for the Macintosh Family*, third edition.)

If you do not want to use the video card name specified in the ROM of the card, you can include in your monitors extension file a string list resource of type `'STR#'`. In that resource, specify an alternative name for the Monitors control panel to display. See “Providing an Alternative Name for a Video Card” on page 8-58 for more information.

You use a card resource to ensure that your monitors extension is called when the user selects the monitor to which your card is connected and clicks the Options button. Because your monitors extension file can contain as many card resources as you wish, one extension file can handle several types of video cards. For example, Listing 8-17 shows two card resources; thus, when the user selects the monitor connected to the SurfBoard Display Card or the SurfBoard Super Display Card, the monitors extension `MyMonExtend` is called. (See Listing 8-25 on page 8-64 for the `MyMonExtend` function.)

Listing 8-17 Rez input for a card ('card') resource

```
resource 'card' (-4080, purgeable)
{
    "SurfBoard Display Card"
};
resource 'card' (-4079, purgeable)
{
    "SurfBoard Super Display Card"
};
```

Defining a Rectangle for a Monitors Extension

You create a rectangle resource of type `'RECT'` to define the display area for the controls of your monitors extension. When the user clicks the Options button in the Monitors control panel, the Monitors control panel uses your monitors extension to expand the Options dialog box under these circumstances: if the monitor connected to your video card is currently selected, and if you have provided a monitors extension for your card. Before displaying it, the Monitors control panel expands the Options dialog box to include the space defined by the rectangle resource. The rectangle resource is required, and its resource ID must be `-4096`.

To specify the top coordinate of your rectangle, determine the height in pixels of the space required to display your controls and specify that value as a negative number. For example, if you need a display area that is 60 pixels high, specify `-60` as the top coordinate. Specify `0` as the left coordinate. This is the same value used to define the left edge of the Options dialog box, and your rectangle should have the same left edge.

Specify `0` as the bottom coordinate. You can think of the distance from the bottom coordinate to the top coordinate—60 pixels, in this example—as the height of your

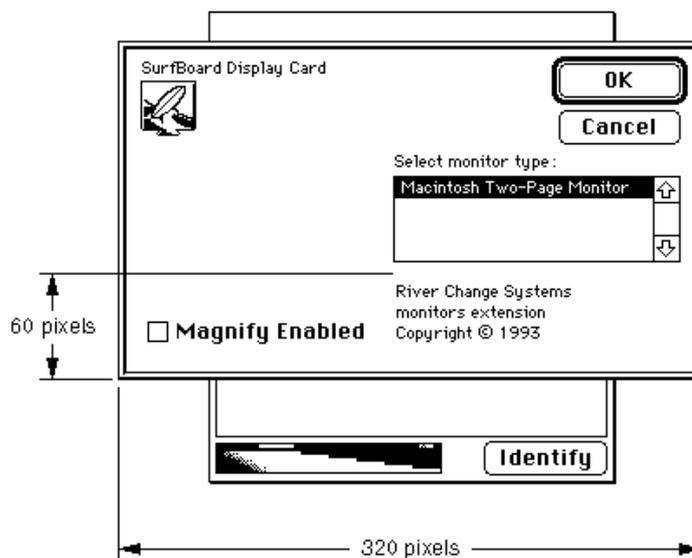
rectangle. Specify 320 as the right coordinate. This is the same value used to define the right edge of the Options dialog box, and your rectangle should have the same right edge.

Note

Although you specify other coordinate values for your rectangle's origin, when you assign coordinates to your controls, assume that the origin of the local coordinate system for your dialog items is (0,0). u

Figure 8-14 shows the Options dialog box for the SurfBoard Display Card. The OK and Cancel buttons and the scrollable list for the monitor type are standard controls. The Magnify Enabled checkbox and three lines of text have been added by the SurfBoard monitors extension. This figure shows the height and width, in pixels, defined in the rectangle resource; this is the area required to display the additional controls.

Figure 8-14 Display area defined by a rectangle resource



Listing 8-18 shows, in Rez input, the rectangle resource used in this example. Notice that the top coordinate is -60 and the bottom coordinate is 0. In other words, the space to be added to the Options dialog box is 60 pixels high.

Listing 8-18 Rez input for a rectangle ('RECT') resource

```
resource 'RECT' (-4096, purgeable)
{
    {-60, 0, 0, 320}
};
```

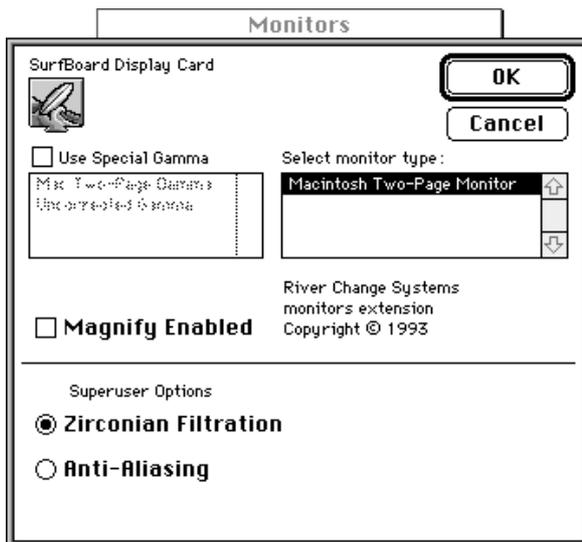
Creating an Item List Resource for a Monitors Extension

You provide an item list resource of type 'DITL' to specify which items you want to appear in the rectangle display area (see the previous section for information about the rectangle resource). In an item list, you specify static text, buttons, checkboxes, radio buttons, editable text, user items, the resource IDs of icons and QuickDraw pictures, and the resource IDs of other types of controls, such as pop-up menus. The item list is required, and its resource ID must be -4096.

When you assign coordinates to your controls, assume that the origin (that is, the upper-left corner) of the local coordinate system is (0,0). The Monitors control panel transforms the coordinates of your controls to the coordinate system that it uses for the Options dialog box. Thus, you must use the `GetDialogItem` procedure to get the true locations of your dialog items. See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the `GetDialogItem` procedure.

If you add additional controls for superusers, you should place them below a horizontal line separating them from other controls, as illustrated in Figure 8-15.

Figure 8-15 The SurfBoard Options dialog box with superuser controls



To draw a dividing line, specify a separate dialog item of type `userItem`. Listing 8-19 shows the item list resource for the SurfBoard monitor extension. Notice that the dividing line (item 2) is defined as a user item.

Listing 8-19 Rez input for the SurfBoard monitors extension item list resource

```
resource 'DITL' (-4096, purgeable) {
    {
        /* [1] */
        {10, 151, 50, 314},
        StaticText { disabled, "River Change Systems\nmonitors extension"
                    "\nCopyright © 1993" },
        /* [2] dividing line for superuser controls*/
        {60, 1, 61, 319},
        UserItem { enabled },
        /* [3] */
        {70, 28, 80, 236},
        StaticText { enabled, "Superuser Options" },
        /* [4] */
        {82, 7, 110, 200},
        RadioButton { enabled, "Zirconian Filtration" },
        /* [5] */
        {112, 7, 132, 200},
        RadioButton { enabled, "Anti-Aliasing" },
        /* [6] */
        {22, 7, 58, 160},
        CheckBox { enabled, "Magnify Enabled" }
    }
};
```

Listing 8-29 on page 8-70 shows the procedure that the SurfBoard monitors extension function uses to draw a line separating the items for normal users from the items displayed for superusers. It uses the `QuickDraw FrameRect` procedure to draw the item as a 1-pixel-high rectangle. After calling the `FrameRect` procedure, a monitors extension can also dither the line in the same manner used to dither menu divider lines. (For information on the `FrameRect` procedure, see *Inside Macintosh: Imaging With QuickDraw*.)

Control Panels

If you use an item color table resource of type 'ictb' to draw your items in color or in a different font, you must include placeholder entries for the standard Options dialog box items before you define the item color table entries to be mapped to the items in your monitors extension item list. This step is necessary because the Monitors control panel appends your monitors extension item list to that of the Options dialog box. To maintain the mapping between entries in the item color table ('ictb') and your item list, you must account for the Options dialog box items.

Currently, the Options dialog box contains 10 items (although this number is subject to change in future implementations of the Monitors control panel). An item color table entry contains two words for each corresponding item. For this implementation of the Monitors control panel, you can ensure that the first item in your item list is mapped to the correct item color table entry as follows: create 10 entries in the item color table to correspond to the 10 items in the Options dialog box, and specify a value of 0 for both words of each entry.

Creating the Monitor Code Resource

A monitor code resource (of type 'mntr') contains the code that carries out the functions of a monitors extension. In MPW, you can set the code resource type to 'mntr' when you link the program. When you create such a resource, the resource must begin with a function that you provide, called the monitors extension function.

The Monitors control panel passes to your monitors extension function parameters that specify actions to perform. You can use the function result to keep a handle to allocated memory or to return an error code. For more information about the monitors extension function, see "Writing a Monitors Extension Function" beginning on page 8-61.

Supplying Optional Resources for a Monitors Extension

Your monitors extension file can also include any of the optional resources described in this section. To create these resources, either you can specify the resource description in an input file and then use a resource compiler, such as Rez, to compile the resource, or you can use a tool such as ResEdit to create your resources in a resource file.

The optional resources and their resource IDs are

- n The icon family resources ('ICN#', 'ics#', 'icl8', 'icl4', 'ics8', and 'ics4'), which specify an icon for display in the upper-left corner of the Options dialog box: resource ID -4096.
- n The version ('vers') resources: resource ID 1 and 2.
- n The string list ('STR#') resource: resource ID -4096.
- n The gamma table ('gama') resource: resource ID from -4080 through -4065.

Control Panels

- n The file reference ('FREF') resource. The resource ID follows the normal conventions (typically, you assign a resource ID of 128).
- n The bundle ('BNDL') resource. The resource ID follows the normal conventions (typically, you assign a resource ID of 128).
- n The icon family resources ('ICN#', 'ics#', 'icl8', 'icl4', 'ics8', and 'ics4'), which define the monitors extension file icon. The resource ID follows the normal conventions (typically, you assign a resource ID of 128).
- n The system extension ('INIT') resource.
- n The signature resource: resource ID 0.

In addition to the optional resources that these sections describe, you can include private optional resources whose resource ID numbers must fall within the range -4080 through -4065.

Specifying an Icon for the Options Dialog Box

To specify an icon that the Monitors control panel displays in the upper-left corner of the Options dialog box, you can define one or more members of an icon family. For each of these resources, you must assign a resource ID of -4064. If you provide an icon family, the Monitors control panel displays the appropriate icon according to the bit depth of the monitor. (Note that in System 6 you provide 'ICON' or 'cicn' icons instead of an icon family.) For more information on these icons, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

If you do not supply either of these icon resources, the Monitors control panel displays the icon defined in the `sRsrcIcon` entry of the `sResource` data structure of your video card's ROM. If you do not supply either of these resources and your video card does not include an icon, the Monitors control panel displays a generic icon that represents a monitor.

Listing 8-20 shows a partial listing of the icon family resources that define the SurfBoard video card icon shown in Figure 8-13 on page 8-51.

Listing 8-20 Rez input for icon family resources for a monitors extension

```
data 'ICN#' (-4064, purgeable) {
    /*icon data goes here*/
};
data 'icl8' (-4064, purgeable) {
    /*icon data goes here*/
};
data 'icl4' (-4064, purgeable) {
    /*icon data goes here*/
};
```

Specifying Version Information

You can include two kinds of version resources of type 'vers' to provide version information for your monitors extension file. The version resource with a resource ID of 1 specifies the version of your monitors extension file. The version resource with a resource ID of 2 specifies the version of the group to which your file belongs—for example, the version number of the video card that your extension file supports.

The Finder displays version information about your monitors extension for the user. For complete information on how to specify the version resources and how the Finder displays the information from these resources in its information window, see the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.

Listing 8-21 shows a version resource with a resource ID of 1 that specifies the version number of the SurfBoard’s monitor extension file. This version resource includes the copyright of the River Change Systems company, which manufactures the card.

Listing 8-21 Rez input for a version ('vers') resource

```
resource 'vers' (1) {
    0x01, 0x00, release, 0x00,
    0, /*verUS*/
    "1.00",
    "1.00, Copyright © 1993 River Change Systems."
};
```

Providing an Alternative Name for a Video Card

The Monitors control panel displays the name of your video card in the upper-left corner of the Options dialog box. By default, it displays the name defined in the declaration ROM of the video card. To display a name for your video card that is different from the name in the declaration ROM of the video card, you can include a string list ('STR#') resource with resource ID -4096. This resource must contain pairs of Pascal strings. The first string in each pair must be identical to the name of your video card as specified in the sResource data structure in the ROM of the card. (For more information on the sResource data structure, see *Designing Cards and Drivers for the Macintosh Family*, third edition.) The second string in each pair is the name that you want to display in the Options dialog box. You can have as many pairs of names in one string list resource as you wish; the Monitors control panel uses the first match it finds.

It is unlikely that you will need to override the name specified in the declaration ROM. However, if you have misspelled the card name on the board, or if you want to display a name that is more descriptive, you can include a string list resource. Listing 8-22 shows the string list resource for the SurfBoard video card monitors extension. This monitors extension includes two card resources (see Listing 8-17 on page 8-52), but the string list resource includes only one entry to override the name SurfBoard Super Display Card. In this example, when the Monitors control panel displays the Options dialog box for the SurfBoard Super Display Card, it displays the name SurfBoard Super Fast Display Card instead of the name in the card's declaration ROM.

Listing 8-22 Rez input for the SurfBoard string list resource

```
resource 'STR#' (-4096, purgeable)
{
    { "SurfBoard Super Display Card";
      "SurfBoard Super Fast Display Card"};
};
```

Supplying Gamma Table Resources

To indicate status as a superuser, the user presses the Option key while clicking the Options button in the Monitors control panel. In response, the Monitors control panel displays a list of gamma tables (see Figure 8-12 on page 8-50).

The software driver for a video card uses a gamma table to correct for the fact that the intensity of each color on a video display is not linearly proportional to the intensity of the electron beam; in other words, the gamma table helps the video driver to provide the most accurate colors possible for a video display. Because the user might prefer a nonstandard color correction, many developers of video cards provide more than one gamma table for a given card.

To supply one or more gamma tables for a video card, include in the monitors extension file a named resource of type 'gama' for each gamma table. To change the default gamma table for a monitor, the user clicks the Use Special Gamma checkbox and then selects a table by clicking its name in the list. The default gamma table for a monitor is the one listed in the screen resource of type 'scrn'. For a complete discussion of gamma tables, see *Designing Cards and Drivers for the Macintosh Family*, third edition. For information on the screen ('scrn') resource, see *Inside Macintosh: Devices*.

Creating File Reference, Bundle, and Signature Resources

The file reference ('FREF'), bundle ('BNDL'), and signature resources work together to give your file a distinctive appearance on the desktop. The Finder uses these resources to display the icon for your monitors extension.

Control Panels

The file reference resource specifies the file type for a monitors extension ('cdev'), the local ID of your icon list resource, and an empty string. The local ID maps the monitors extension file type to the icon list resource that is assigned the same local ID in the bundle resource. Listing 8-23 shows the file reference resource for the SurfBoard monitors extension.

Listing 8-23 Rez input for a file reference resource of a monitors extension

```
resource 'FREF' (128, purgeable) {
    'cdev', 0, ""
};
```

Note

If you provide the complete icon family, the Finder displays the appropriate member of the icon family according to the bit depth of the monitor. ^u

The Finder uses the signature resource with the bundle resource to establish the identity of your monitors extension. You define a signature resource as a string resource (that is, a resource of type 'STR') and specify as its resource type a unique four-character sequence that has the same value as your monitors extension's creator type. The signature resource contains a string that identifies your monitors extension; typically the string specifies the name, version number, and release date of the monitors extension.

A bundle ('BNDL') resource associates all of the resources that the Finder uses for your monitors extension. It associates your monitors extension and its signature with its icon. The Finder requires the information in the bundle resource to display icons for your monitors extension. In the bundle resource, you must specify a local ID for your icon list resource that matches the local ID you assigned inside the corresponding file reference resource. In the bundle resource shown in Listing 8-24, local ID 0 is assigned to the icon list resource with resource ID 128, mapping the icon defined for the SurfBoard monitors extension to the monitors extension file.

Listing 8-24 Rez input for a bundle resource of a monitors extension

```
resource 'BNDL' (128, purgeable) {
    'kcah',
    0,
    {
        'ICN#', {0, 128},
        'FREF', {0, 128}
    }
};
```

(See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information on how to create file reference, signature, and bundle resources.)

Including a System Extension Resource

A file that contains an extension to the Monitors control panel can contain a system extension resource of type 'INIT'. If your monitors extension file is in the Control Panels folder, the Extensions folder, or the base level of the System Folder, then the system software executes the system extension resource when the user starts or restarts the computer.

Although the system extension resource acts independently of other resources in the file, it should be related to the monitors extension.

Writing a Monitors Extension Function

You create a monitors extension function to implement the feature for your video card and manage the controls that allow the user to set values for that feature. The Monitors control panel calls your monitors extension function, requesting it to perform an action or handle an event in response to the user's manipulation of the controls for your video card. The `message` parameter identifies the action or event.

Your monitors extension function should perform the requested action and return a function result to the Monitors control panel. This function result should be either a standard value indicating that your monitors extension function has not allocated memory, a handle to any memory you allocate, or an error code. Here is how you declare a monitors extension function:

```
FUNCTION MyMntrExt (message, item, numItems: Integer; monitorValue: LongInt;
    mDialog: DialogPtr; theEvent: EventRecord;
    screenNum: Integer; VAR screens: ScrnRsrcHandle;
    VAR scrnChanged: Boolean): LongInt;
```

The `message` parameter can contain any of the values defined by these constants:

```
CONST
    startupMsg    = 12; {status of user (whether a superuser)}
    initMsg      = 1;  {perform initialization}
    okMsg        = 2;  {user clicked OK button}
    cancelMsg    = 3;  {user clicked Cancel button}
    hitMsg       = 4;  {user clicked enabled control}
    nulMsg       = 5;  {null event}
    keyEvtMsg    = 9;  {keyboard event}
    updateMsg    = 6;  {update event}
```

Control Panels

The value of the `message` parameter indicates the action your monitors extension function should perform:

- n `startupMsg`. Informs your monitors extension function that it has been loaded into memory. Your function can determine whether the user has superuser status by examining the `item` parameter. The Monitors control panel sets the `item` parameter to 1 if the user is a superuser. Your code should load any resources and modify them if necessary for the capabilities of the computer system or selection of superuser status. You can also allocate memory in response to this message, and store the value identifying the user's status.
- n `initMsg`. Requests your monitors extension function to perform initialization.
- n `okMsg`. Indicates that the user clicked the OK button. Your function should check for any values the user changed, release any memory it allocated, and return control to the Monitors control panel.
- n `cancelMsg`. Indicates that the user clicked the Cancel button. Your function should restore the system to the state it was in before the user clicked the Options button, release any memory it allocated, and return control to the Monitors control panel. If the user modified any values before clicking the Cancel button, reinstate the original values.
- n `hitMsg`. Indicates that the user clicked an enabled control in your monitors extension. Your function should handle the click.
- n `nullMsg`. Requests your control device function to handle a null event by performing any idle processing. Your monitors extension function should do minimal processing in response to a null event; for example, it should not refresh control settings. The Monitors control panel passes the event record for the null event in the parameter `theEvent`.
- n `keyEvtMsg`. Requests your monitors extension function to handle a key-down or auto-key event.
- n `updateMsg`. Requests your monitors extension function to update any user items and redraw any controls that are not standard dialog items handled by the Dialog Manager.

In addition, the `message` parameter can contain any of the values defined by these constants:

```
CONST
    activateMsg      = 7;  {becoming active (not currently used)}
    deactivateMsg    = 8;  {becoming inactive (not currently used)}
    superMsg         = 10; {user is a superuser}
    normalMsg        = 11; {user is not a superuser}
```

Control Panels

These messages either are provided for backward compatibility or are not currently used:

- n `activateMsg`. Requests your monitors extension function to respond to an activate event by making your video card's controls active. Currently, this message is not used because the Options dialog box is modal. However, your monitors extension function should handle this message as it would any activate event because in future implementations the Options dialog box might be modeless.
- n `deactivateMsg`. Requests your monitors extension function to respond to an activate event by making your video card's controls inactive. Currently, this message is not used because the Options dialog box is modal. However, your monitors extension function should handle this message as it would any activate event because in future implementations the Options dialog box might be modeless.
- n `superMsg`. Informs your monitors extension function that the user has selected superuser status. This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls that you have reserved for superusers, if your function has not already done so in response to either the `startupMsg` or `initMsg` message. If your function does not handle this message, it should return as its function result a handle to any memory it previously allocated. The Monitors control panel sends the message `superMsg` or `normalMsg` immediately following the initialization message.
- n `normalMsg`. Informs your monitors extension function that the user has not selected superuser status. This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls, if your code has not already done so in response to either the `startupMsg` or `initMsg` message. If your function does not handle this message, it should return as its function result a handle to any memory it previously allocated. The Monitors control panel sends the message `normalMsg` or `superMsg` immediately following the initialization message.

IMPORTANT

If your monitors extension function cannot handle a message, it should return as its function result a handle to any memory it previously allocated. Otherwise, it should return the value passed in the `monitorValue` parameter. s

For a description of the remaining parameters of the monitors extension function, see "Monitors Extension Functions" beginning on page 8-78.

Your monitors extension function can return either an error code or a handle to memory it allocated. Each time the Monitors control panel calls your monitors extension function, the `monitorValue` parameter contains the value that your function returned as its function result the last time it was called.

If an error occurs, your monitors extension function should display an error dialog box and then return a value between 1 and 255. If your function returns a value in this range, the Monitors control panel closes the Options dialog box immediately and does not call your monitors extension function again.

The monitors extension used as an example in this chapter adds controls to the Options dialog box for a video card called SurfBoard. The Magnify Enabled checkbox allows the user to magnify the display of text and graphics on the monitor connected to the

Control Panels

SurfBoard video card. The SurfBoard monitors extension also includes controls for superusers, which illustrate how to implement the rectangle extension in which the superuser controls are displayed. The SurfBoard monitors extension shows one way of handling messages from the Monitors control panel.

Listing 8-25 shows the SurfBoard monitors extension function, `MyMonExtend`. It includes a `CASE` statement that handles messages that the Monitors control panel passes to `MyMonExtend`. First the function sets up a handle for memory that it allocates in response to the startup message. The function returns a handle to the storage it allocates as its function result in response to the startup message, unless an error occurs (see Listing 8-26 on page 8-66). For all subsequent messages, the Monitors control panel passes, in the `monitorValue` parameter, the previous function result. The `MyMonExtend` function returns the handle to the allocated memory as its function result for any messages that it does not handle.

Listing 8-25 A monitors extension function

```
UNIT SurfBoardMonExt;
INTERFACE
  {include a Uses statement if your programming environment requires it}
CONST
  kTextItem          = 1;          {static text item}
  kSuperUserDivLine = 2;          {separation line}
  kFilterControl     = 4;          {radio button filter}
  kAntiAliasingCntl = 5;          {radio button aliasing}
  kMagnifyControl    = 6;          {checkbox for Magnify Enabled}
  kMemErrAlert       = 130;        {resource ID of out-of-memory alert box}
  kdeepAlert         = 131;        {resource ID of alert box}
  kResID             = 133;        {all other errors}
TYPE
  MonitorDataRec =
    RECORD
      {local data for the extension}
      isSuperUser:      Boolean;
      filteringSetting: Integer;
      oldFiltering:     Integer;
      toggleMagnifyValue: Integer;
    END;
  MonitorDataPtr      = ^MonitorDataRec;
  MonitorDataHandle   = ^MonitorDataPtr;

  MyRectHandle = ^RectPtr;
  MyIntPtr      = ^Integer;
  MyIntHandle   = ^MyIntPtr;
```

CHAPTER 8

Control Panels

```
FUNCTION MyMonExtend (message, item, numItems: Integer;
                    monitorValue: LongInt; mDialog: DialogPtr;
                    theEvent: EventRecord; ScreenNum: Integer;
                    VAR Screens: ScrnRsrcHandle;
                    VAR ScrnChanged: Boolean): LongInt;

IMPLEMENTATION
{any support routines your monitors extension function uses}
PROCEDURE MyHandleStartupMsg(item: Integer; mDialog: DialogPtr;
                            VAR monitorValue: LongInt); FORWARD;
PROCEDURE MyHandleInitMsg(numItems: Integer; mDialog: DialogPtr;
                          dataRecHand: MonitorDataHandle); FORWARD;
PROCEDURE MyDrawRect(theWindow: WindowPtr; itemNo: Integer); FORWARD;
FUNCTION MySetUpData (superUser: Integer; storage: MonitorDataHandle): OSErr;
                    FORWARD;
PROCEDURE MyHandleHits (mDialog: DialogPtr; whichItem, numItems: Integer;
                      dataRecHand: MonitorDataHandle); FORWARD;
PROCEDURE MySaveNewValues (dataRecHand: MonitorDataHandle); FORWARD;
PROCEDURE MyUndoChanges (item, numItems: Integer; mDialog: DialogPtr;
                        dataRecHand: MonitorDataHandle); FORWARD;

FUNCTION MyMonExtend (message, item, numItems: Integer; monitorValue: LongInt;
                    mDialog: DialogPtr; theEvent: EventRecord;
                    ScreenNum: Integer; VAR Screens: ScrnRsrcHandle;
                    VAR ScrnChanged: Boolean): LongInt;

VAR
    dataRecHand: MonitorDataHandle;
BEGIN
    IF message <> startupMsg THEN
        dataRecHand := MonitorDataHandle(monitorValue); {set up handle}
    CASE message OF
    startupMsg:
        MyHandleStartupMsg(item, mDialog, monitorValue);
    initMsg:
        MyHandleInitMsg(numItems, mDialog, dataRecHand);
    hitMsg:
        MyHandleHits(mDialog, item, numItems, dataRecHand);
    okMsg:
        MySaveNewValues(dataRecHand);
    cancelMsg:
        MyUndoChanges(item, numItems, mDialog, dataRecHand);
    END; {of CASE}
    MyMonExtend := monitorValue; {return value with handle}
END; {MyMonExtend}
```

Handling the Startup Message

After the code in your monitors ('mntr') code resource is loaded and before the Monitors control panel finds any resources to which your monitors extension function refers, the Monitors control panel calls your function with a startup (startupMsg) message. If the user is a superuser, the Monitors control panel sets the `item` parameter to 1 for the startup message.

The startup message requests your monitors extension function to load and modify any resources that must allow for the capabilities of the computer or for superusers. For example, your monitors extension function should modify the rectangle resource if the user is a superuser.

In response to a startup message, your function can also create a handle and allocate any memory that it needs to store values between calls from the Monitors control panel. For example, if your function initializes its controls in response to the initialization (initMsg) message, it should store a value indicating whether or not the user is a superuser. When the Monitors control panel calls your monitors extension function with an initialization message, the `item` parameter no longer indicates the user's status. If your code allocates memory, your function should return as its function result a handle to the memory it allocates in response to the startup message, unless an error occurs. If an error occurs, your function can display an error dialog box and return a function result of 255, indicating an error condition. Listing 8-26 shows how the `MyMonExtend` function handles the startup message.

Listing 8-26 Handling the startup message

```
PROCEDURE MyHandleStartupMsg (item: Integer; mDialog: DialogPtr;
                             VAR monitorValue: LongInt);
VAR
    dataRecHand:   MonitorDataHandle;
    result:        OSErr;
    i:             Integer;
BEGIN
    {allocate memory to store data}
    dataRecHand :=
        MonitorDataHandle(NewHandle(sizeof(MonitorDataRec)));
    IF dataRecHand <> NIL THEN
        BEGIN
            result := MySetUpData(item, dataRecHand);
            IF result = noErr THEN
                monitorValue := LongInt(dataRecHand)
            ELSE {error function result stops any further action}
                monitorValue := result;
        END
    END
```

Control Panels

```

ELSE
BEGIN {dataRecHand not allocated}
    i := StopAlert(kMemErrAlert, NIL);
    {error function result stops any further action}
    monitorValue := 255;
END;
END;

```

Allocating Storage in Response to the Initialization Message

If your monitors extension function does not allocate memory in response to a startup message, it can do so in response to an initialization message, and then use the superuser (`superMsg`) or the normal user (`normalMsg`) message to initialize control values and user items, if any. The Monitors control panel does not display the Options dialog box until after your monitors extension function returns from either of these messages. \cup

If your function returns an error in response to the startup message, the Monitors control panel does not display the Options dialog box. Your code can display an alert box describing the error before returning control to the Monitors control panel.

After it allocates storage, the function shown in Listing 8-26 calls its own `MySetUpData` function to check the value of the `item` parameter. This value indicates whether the user has selected superuser status.

Listing 8-27 shows the `MySetUpData` function. If the user is not a superuser, the SurfBoard monitors extension uses the default values for the rectangle resource. (This rectangle ends just before the dividing line, so that the superuser controls are not displayed.) If the user is a superuser, `MySetUpData` extends the rectangle in the rectangle ('RECT') resource to include all of the controls in the item list resource ('DITL') resource. If an error occurs, the function notifies the user and returns an error code value of 255 as its function result.

Listing 8-27 Using a normal user rectangle or extending it to display superuser controls

```

FUNCTION MySetUpData(superUser: Integer; storage: MonitorDataHandle): OSErr;
VAR
    magnifyHdl:           Handle;
    intensityLevelHdl:   Handle;
    resHandle:           Handle;
    i:                   Integer;
    result:              OSErr;
BEGIN
    result := noErr;
    HLock(Handle(storage));
    WITH storage^^ DO
    BEGIN

```

Control Panels

```

{open preferences file first if needed}
magnifyHdl := GetResource('MAGN', kResID);
IF magnifyHdl <> NIL THEN
BEGIN
    toggleMagnifyValue := MyIntHandle(magnifyHdl)^^;
    ReleaseResource(magnifyHdl);
END;
IF superUser = 1 THEN
BEGIN
    isSuperUser := TRUE;
    intensityLevelHdl := GetResource('INTE', kResID);
    IF intensityLevelHdl <> NIL THEN
    BEGIN
        oldFiltering := MyIntHandle(intensityLevelHdl)^^;
        filteringSetting := oldFiltering;
        ReleaseResource(intensityLevelHdl);
        resHandle := GetResource('RECT', -4096);
        IF resHandle <> NIL THEN
            RectHandle(resHandle)^^.top := -160
        ELSE
            result := 255
        END
    ELSE
        result := 255;
    END {of superuser = 1}
    {close preferences file}
END; {of WITH}
IF result = 255 THEN
BEGIN
    DisposeHandle(Handle(storage));
    i := StopAlert(kdeepAlert, NIL);
END;
HUnlock(Handle(storage));
MySetUpData := result;
END;

```

Performing Initialization

Before it displays the Options dialog box and after it has located any resources that your monitors extension includes, such as gamma table ('gama') resources, the Monitors control panel calls your monitors extension function with an `initMsg` message. When your monitors extension function receives this message, it should set default values for controls. To handle this message, your function can initialize the settings of its controls. If it hasn't already allocated memory in response to the startup message, your function can

Control Panels

allocate memory when it performs initialization. The Monitors control panel calls your monitors extension with an initialization message after the startup message and before either the superuser or normal message.

If your function returns an error in response to the `initMsg` message, the Monitors control panel does not display the Options dialog box. Your function can display an alert box describing the error before returning control to the Monitors control panel.

Listing 8-28 shows the `MyHandleInitMsg` procedure, which the `MyMonExtend` function calls to handle the initialization message. First `MyHandleInitMsg` sets its controls to their initial values; `MyHandleInitMsg` calls the Dialog Manager's `GetDialogItem` and the Control Manager's `SetControlValue` procedures for this purpose. Then, if the user is a superuser, the procedure installs the procedure that draws the dividing line between the normal controls and superuser controls, then initializes the settings of its superuser controls.

Listing 8-28 Initializing a monitors extension

```
PROCEDURE MyHandleInitMsg (numItems: Integer; mDialog: DialogPtr;
                          dataRecHand: MonitorDataHandle);

VAR
    itemType:      Integer;
    itemHandle:    Handle;
    itemRect:      Rect;
BEGIN
    GetDialogItem(mDialog, numItems+kMagnifyControl, itemType,
                  itemHandle, itemRect);
    SetControlValue(ControlHandle(itemHandle),
                    (dataRecHand^.toggleMagnifyValue));
    IF dataRecHand^.isSuperUser THEN
    BEGIN
        GetDialogItem(mDialog, numItems+kSuperUserDivLine, itemType,
                      itemHandle, itemRect);
        SetDialogItem(mDialog, numItems+kSuperUserDivLine, itemType,
                      @MyDrawRect, itemRect);
        IF dataRecHand^.oldFiltering = 0 THEN
            GetDialogItem(mDialog, numItems+kAntiAliasingCntl,
                          itemType, itemHandle, itemRect)
        ELSE
            GetDialogItem(mDialog, numItems+kFilterControl,
                          itemType, itemHandle, itemRect);
        SetControlValue(ControlHandle(itemHandle), 1);
    END;
END;
```

Control Panels

Listing 8-29 shows the `MyDrawRect` procedure, which draws the line dividing superuser controls from other controls. The `MyDrawRect` procedure uses the `FrameRect` procedure to draw a 1-pixel-high rectangle. Note that `MyDrawRect` specifies the coordinates for the dividing line in the coordinate system used by its rectangle ('RECT') resource. If you wish, you can draw this line in a gray pattern so that it looks similar to the dividers in menus. (For information on the `FrameRect` procedure, see *Inside Macintosh: Imaging With QuickDraw*.)

Listing 8-29 Drawing a line to separate superuser controls

```
PROCEDURE MyDrawRect (theWindow: WindowPtr; itemNo: Integer);
VAR
    itemType:      Integer;
    itemHdl:       Handle;
    itemRect:      Rect;
BEGIN
    GetDialogItem(theWindow, itemNo, itemType, itemHdl, itemRect);
    FrameRect(itemRect);
END;
```

Responding to a Click in the OK Button

The Monitors control panel calls your monitors extension function with an OK (`okMsg`) message when the user clicks the OK button. The OK button is a standard control defined for the Options dialog box by the Monitors control panel. When the user clicks the OK button, the Monitors control panel hides the Options dialog box.

This message is a signal to put user preferences into effect. You should not make any changes requested by the user irreversible until you receive this message. This is your last chance to check the values of any controls or editable text items that the user might have changed. Your monitors extension function should update the resources in which it saves values; it should also make any hardware changes necessary. Your function should release any memory it has allocated before returning control to the Monitors control panel.

Control Panels

The `MyMonExtend` function (see Listing 8-25 on page 8-64) calls its own `MySaveNewValues` procedure to handle an OK message from the Monitors control panel. This procedure checks if the user has changed the setting of the Magnify Enabled checkbox. If the user is a superuser, it also checks the values of the Anti-Aliasing and Zirconian Filtration radio buttons. If the user changed values, `MyMonExtend` writes the values to its preferences file, which is stored in the Preferences folder, and releases any memory it has allocated before it returns to the Monitors control panel.

Responding to a Cancel Request

When the user clicks the Cancel button, the Monitors control panel calls your monitors extension function with a cancel (`cancelMsg`) message. The Cancel button is a standard control defined for the Options dialog box by the Monitors control panel. To handle the cancel request, your monitors extension function should restore the system to its former state, before the user clicked the Options button; release any memory it allocated; and return control to the Monitors control panel. If your function modified any values the user specified before clicking the Cancel button, reinstate the original values.

Handling Mouse Events for a Monitors Extension

When the user clicks any active enabled control that your monitors extension defined for the Options dialog box, system software generates mouse events. The Monitors control panel intercepts these events and passes them to your monitors extension function as a `hitMsg` message. Your monitors extension function typically changes the setting of the control or performs the appropriate action in response to a `hitMsg` message.

Along with the `hitMsg` message, the Monitors control panel passes three values that your monitors extension function uses to determine which item the user clicked.

- n In the `item` parameter, the number of the item clicked. This is not the number you assign in your item list, but the number after the Monitors control panel appends your item list to the item list of the Options dialog box.
- n In the `numItems` parameter, the number of items in the item list of the standard Options dialog box.
- n In the parameter `theEvent`, the event record for the mouse event that generated the `hitMsg` message.

The Monitors control panel appends the items you define in your monitors extension item list to the item list for the standard controls in the Options dialog box. Therefore, to get the actual number of your item, subtract `numItems` from `item`.

Listing 8-30 shows the `MyHandleHits` procedure, which `MyMonExtend` calls to handle a `hitMsg` message. This procedure determines the item number of the clicked control, as defined in the monitors extension's item list resource. It does this by subtracting the number of items in the item list of the Options dialog box (`numItems`) from the item the user clicked (`whichItem`) to get the correct item number. Then `MyHandleHits` calls the Dialog Manager's `GetDialogItem` procedure and the Control Manager's `SetControlValue` procedure to set the control to the new value indicated by the user.

Listing 8-30 Responding when a user clicks a control

```
PROCEDURE MyHandleHits (mDialog: DialogPtr; whichItem, numItems: Integer;
                        dataRecHand: MonitorDataHandle);

VAR
    itemType: Integer;
    itemHandle: Handle;
    itemRect: Rect;
BEGIN
    HLock(Handle(dataRecHand));
    WITH dataRecHand^^ DO
    BEGIN
        CASE whichItem - numItems OF
            kFilterControl:
                BEGIN
                    GetDialogItem(mDialog, whichItem, itemType, itemHandle,
                                itemRect);
                    SetControlValue(ControlHandle(itemHandle),1);
                    GetDialogItem(mDialog, numItems+kAntiAliasingCntl, itemType,
                                itemHandle, itemRect);
                    SetControlValue(ControlHandle(itemHandle),0);
                    filteringSetting := 1;
                END;
            kAntiAliasingCntl:
                BEGIN
                    GetDialogItem(mDialog, numItems+kFilterControl, itemType,
                                itemHandle, itemRect);
                    SetControlValue(ControlHandle(itemHandle),0);
                    GetDialogItem(mDialog, whichItem, itemType, itemHandle,
                                itemRect);
                    SetControlValue(ControlHandle(itemHandle),1);
                    filteringSetting := 0;
                END;
        END;
    END;
END;
```

Control Panels

```

kMagnifyControl:
    BEGIN
        GetDialogItem(mDialog, whichItem, itemType, itemHandle,
            itemRect);
        toggleMagnifyValue := 1 - toggleMagnifyValue;
        SetControlValue(ControlHandle(itemHandle),
            toggleMagnifyValue);
    END;
END; {end of CASE}
END;
HUnlock(Handle(dataRecHand));
END;

```

Handling Keyboard Events

The Monitors control panel intercepts all key-down and auto-key events for your monitors extension and sends your monitors extension function a keyboard event through the `keyEvtMsg` message. The Monitors control panel passes, in the parameter `theEvent`, the event record for the keyboard event. If your monitors extension includes an editable text item and the user issues a Cut, Copy, or Paste command using the Command-key equivalent, the Monitors control panel passes this event to your monitors extension function in the event record.

Including Another Control Panel Definition in a Monitors Extension File

A control panel file that contains an extension to the Monitors control panel can also contain a definition for another, separate control panel. You might want to include both an extension to the Monitors control panel and a new control panel definition in the same file, for example, if each controls some features of the same video card. Any control panel definition must include a resource of type `'cdev'` and the other resources described in “Creating a Control Panel’s Resources” beginning on page 8-14.

Because the control panel resources and the monitors extension resources in the file have different resource ID numbers, the Finder handles them separately. If the user opens a control panel file containing both a control panel definition and an extension to the Monitors control panel, the control panel defined in that file appears on the screen, and the Finder ignores the monitors extension in that file. If the user opens the Monitors control panel file, then the Monitors control panel searches the other control panel files in the same folder for extensions and ignores any control resources of type `'cdev'` it finds in those files. The user cannot open a control panel file that contains only an extension to the Monitors control panel; only the Monitors control panel can open such a file.

Control Panels Reference

This section describes the application-defined routines and the resources that are specific to control panels and extensions to the Monitors control panel.

The section “Application-Defined Routines” describes the control device function that you must provide for a control panel and the monitors extension function that you must provide for an extension to the Monitors control panel. You create a control device function to implement a control panel. A control device function should respond to messages from the Finder, handling any events or performing any actions as requested by the Finder. A monitors extension function extends the Monitors control panel to provide support for a video device so that users can control its settings.

The “Resources” section lists the resources required for a control panel or an extension to the Monitors control panel. It includes specific sections for the resources you must supply for a control panel or a monitors extension if those resources are not fully documented elsewhere in *Inside Macintosh*, and it indicates where to find information about required resources that are not covered in that section.

Application-Defined Routines

This section describes the control device function and the monitors extension function.

Control Device Functions

A control device ('cdev') code resource contains a control device function that implements the features of a control panel.

MyCdev

You provide a control device function to implement your control panel. In the `message` parameter, the Finder passes a value indicating which action your control device function should perform. Here's how you declare a control device function called `MyCdev`:

```
FUNCTION MyCdev(message, item, numItems, CPrivateValue: Integer;  
                VAR theEvent: EventRecord;  
                cdevStorageValue: LongInt; CPDialog: DialogPtr)  
                : LongInt;
```

Control Panels

message	A value that identifies the event or action to which your control device function should respond. See Table 8-3 on page 8-76 for the constants your function can receive in this parameter.
item	The number of the item that the user clicked. In System 7, this is always the actual number of the item in your item list. In System 6, the Control Panel desk accessory appends your item list to its own. Although you begin numbering your item list with 1, the Control Panel adds the number of items in its item list to your item. Therefore, to get the actual number of the clicked item, and to provide for backward compatibility, your control device function should always subtract <code>numItems</code> from <code>item</code> .
numItems	In System 7, the Finder passes a value of 0 for this parameter. This parameter is provided for backward compatibility with the Control Panel desk accessory. In System 6, this parameter contains the number of items in the item list belonging to the Control Panel desk accessory. To get the actual number of the item that the user clicked, subtract <code>numItems</code> from <code>item</code> .
CPrivateValue	Reserved for use by the Finder or the Control Panel desk accessory.
theEvent	The event record for the event that caused the Finder to send a <code>hitDev</code> , <code>nulDev</code> , <code>activDev</code> , <code>deActivDev</code> , <code>updateDev</code> , or <code>keyEvtDev</code> message to your control device function. See the chapter “Event Manager” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for a description of events and event records.
cdevStorageValue	<p>The first time the Finder calls your control device function, this parameter is set to the constant <code>cdevUnset</code>. After the first call, this parameter contains the function result last returned by your control device function. Typically, in response to an <code>initDev</code> message, a control device function allocates a handle to memory and returns this handle as its function result. It does this so that it can store values between calls from the Finder. On all subsequent calls, the Finder passes the handle back to your function as the value of <code>cdevStorageValue</code>, and your function returns this value as its function result until an error condition occurs or the user closes the control panel.</p> <p>If your function does not create a handle, your function and the Finder pass <code>cdevUnset</code> back and forth, instead of the handle, until an error condition occurs or the user closes the control panel.</p>
CPDialog	The dialog pointer for your control panel’s dialog box. The dialog can be a color dialog on systems that support color windows. See the chapter “Dialog Manager” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for a description of dialog pointers.

DESCRIPTION

The Finder calls your control device function repeatedly with various messages in response to user actions and events from the time the user opens your control panel until the user closes the control panel or your function reports an error condition from which it cannot recover. Before attempting to handle messages from the Finder, your control device function should determine whether enough memory is available to perform the requested action.

Depending on how you define your control panel's machine resource, the Finder calls your control device function for the first time with a `macDev` message or an `initDev` message. Apart from a `macDev` message, your control device function should ignore any messages that it receives before an `initDev` message. Your function should also ignore any messages it receives after a `closeDev` message, which the Finder sends under normal conditions free of error as a signal that your function should begin its termination process: releasing any allocated memory, handles, pointers, and so on. Between the `initDev` and the `closeDev` calls, the Finder calls your control device function to direct it to handle activate, update, keyboard-related, mouse-related, and null events. When the user chooses a command from the Finder's Edit menu, the Finder passes the command to your function as an edit message. Table 8-3 lists the constant names for the values that the Finder passes in the `message` parameter and provides a description of the action your function should perform.

Table 8-3 Messages from the Finder

Constant	Value	Description
<code>initDev</code>	0	Your control device function should perform any initialization, set default values for controls, and create a handle to any memory that it needs.
<code>hitDev</code>	1	The user clicked an enabled item, and your control device function should handle the click.
<code>closeDev</code>	2	The user closed the control panel; your function should terminate after disposing of any handles and pointers it created. (In System 6 and earlier, the user could have also selected another control panel.)
<code>nulDev</code>	3	A null event occurred. Your control device function should perform idle processing. Do not assume any particular timing for this message.
<code>updateDev</code>	4	An update event occurred. Your control device function should update any user items and redraw any controls that are not standard dialog items handled by the Dialog Manager.
<code>activDev</code>	5	Your control panel is becoming active as the result of an activate event. Your control device function should make the default button and any other controls in your control panel active.

Table 8-3 Messages from the Finder (continued)

Constant	Value	Description
deActivDev	6	Your control panel is becoming inactive as the result of an activate event. Your control device function should make the default button and any other controls in your control panel inactive.
keyEvtDev	7	A key-down or an auto-key event occurred. Your control device function should process the keyboard event.
macDev	8	Your control device function should check the hardware and software configuration to determine whether the control panel can run on it. Your function should return a function result of 1 if it can run and 0 if it cannot.
undoDev	9	The user chose the Undo command from the Finder's Edit menu. Your control device function should handle the command.
cutDev	10	The user chose the Cut command from the Finder's Edit menu. Your control device function should handle the command.
copyDev	11	The user chose the Copy command from the Finder's Edit menu. Your control device function should handle the command.
pasteDev	12	The user chose the Paste command from the Finder's Edit menu. Your control device function should handle the command.
clearDev	13	The user chose the Clear command from the Finder's Edit menu. Your control device function should handle the command.

In System 7, the Finder processes all Command-key equivalents on behalf of your control panel, except those that it maps to its own Edit menu commands. The Finder converts these Command-key equivalents to edit messages, which it then passes to your control panel for processing. In System 6, the Control Panel passes both commands from the Edit menu and their Command-key equivalents to your control device function for processing. See the sections “Responding to Keyboard Events” on page 8-37 and “Handling Edit Menu Commands” on page 8-46 for more information on how to handle Command-key equivalents.

If your function cannot recover from an error condition, it must return one of three error codes to the Finder after disposing of any memory, handles, and pointers that it created and restoring the system stack to the state it would be in after successful execution. See Table 8-2 on page 8-47 for the error codes that your control device function can return.

SEE ALSO

For information on how to write a control device function, see “Writing a Control Panel Function” beginning on page 8-25. For information on the required and optional resources for your control panel, see “Creating a Control Panel’s Resources” beginning on page 8-14.

Monitors Extension Functions

A monitor ('mnr') code resource contains a monitors extension function, which adds controls to the Options dialog box of the Monitors control panel. This function implements the features that allow users to set values for the added controls.

MyMnrExt

You provide a monitors extension function to implement the features that allow users to set the controls for your video card. Your function should respond appropriately to any messages sent to it by the Monitors control panel. In the `message` parameter, the Monitors control panel passes a value indicating which action your function should perform. Here’s how you declare a monitors extension function called `MyMnrExt`:

```
FUNCTION MyMnrExt (message, item, numItems: Integer;
                  monitorValue: LongInt; mDialog: DialogPtr;
                  theEvent: EventRecord;
                  screenNum: Integer; VAR screens: ScrnRsrcHandle;
                  VAR scrnChanged: Boolean): LongInt;
```

- `message` A value that identifies the event or action to which your monitors extension function should respond. See Table 8-4 on page 8-80 for the values your function can receive in this parameter.
- `item` For `hitDev` messages, the number of the item that the user clicked. The Monitors control panel appends your item list to its own. So, although you begin numbering your item list with 1 in your item list resource, the Monitors control panel adds the number of standard items in the Options dialog box’s item list to your item. Therefore, to get the actual number of the clicked item, your monitors extension function should always subtract `numItems` from `item`.
- For the `startupMsg` message, the `item` parameter indicates whether the user has selected superuser status. If so, the `item` parameter is 1; if not, it is 0.
- `numItems` The item list number of the last standard item in the Options dialog box.
- `monitorValue` The first time the Monitors control panel calls your monitors extension function, that is, when the `message` parameter equals `startupMsg`, the value of the `monitorValue` parameter is 0. After the first call, this

Control Panels

parameter contains the result your monitors extension function returned the last time the Monitors control panel called it. Because control panel routines, including a monitors extension function, cannot use global variables to store data between calls, your function can use its function result to return a handle to any memory it allocates. The next time the Monitors control panel calls your monitors extension function, it passes the handle back to your function in the `monitorValue` parameter.

If your monitors extension function returns a function result in the range 1 through 255, the Monitors control panel interprets this result as an error and closes your Options dialog box. Therefore, your monitors extension function will not receive a value in this range in the `monitorValue` parameter.

<code>mDialog</code>	The dialog pointer for the Options dialog box. See the chapter “Dialog Manager” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for a description of dialog pointers.
<code>theEvent</code>	The event record for an event that caused the Monitors control panel to pass a <code>hitMsg</code> , <code>nulMsg</code> , or <code>keyEvtMsg</code> message to your monitors extension function. See the chapter “Event Manager” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for a description of events and event records.
<code>screenNum</code>	The number of the screen device (that is, the monitor) that the user selected. The Monitors control panel numbers monitors consecutively, in the same order as the slots in which the cards are installed, starting with 1.
<code>screens</code>	A handle to a screen (‘ <code>scrn</code> ’) resource. See <i>Inside Macintosh: Devices</i> for information on the screen resource.
<code>scrnChanged</code>	<p>A Boolean value that you can use to indicate whether you have modified the screen (‘<code>scrn</code>’) resource. Set this parameter to <code>TRUE</code> if you have modified the screen resource. When you set the <code>scrnChanged</code> parameter to <code>TRUE</code>, the Monitors control panel checks whether the values in the screen resource are still valid; if there is a problem, the Monitors control panel tries to correct it.</p> <p>This parameter makes it easier to implement a control that changes the apparent area displayed on the screen. For example, your monitor might be able to display either two pages of a document or a magnified view of a single page. If the user changes the area displayed on one screen in a system with multiple screens, the displays on adjacent screens could overlap or show gaps. When you change the screen resource to implement this change, the coordinates of the global rectangles for adjacent screens are no longer contiguous. In this case, if you have set the <code>scrnChanged</code> parameter to <code>TRUE</code>, the Monitors control panel shifts the virtual locations of the screens to eliminate the gaps or overlaps.</p>

DESCRIPTION

The Monitors control panel calls your monitors extension function repeatedly with messages requesting your function to perform an action or handle an event that occurs while the Options dialog box is displayed. Table 8-4 lists the constant names for the

values that the Monitors control panel passes in the `message` parameter and provides a description of the action your function should perform.

Table 8-4 Messages from the Monitors control panel

Constant	Value	Description
<code>initMsg</code>	1	<p>Your monitor extension function should perform initialization; it should allocate any memory it needs and set default values for its controls.</p> <p>The Monitors control panel sends this message to your function before it displays the Options dialog box but after it locates any resources, such as gamma tables, that your extension includes.</p>
<code>okMsg</code>	2	<p>When the user clicks the OK button, the Monitors control panel hides the Options dialog box and calls your monitors extension function with this message. This is your function's last chance to check the values of dialog items that the user might have changed. Your function should release any memory that it previously allocated before returning control to the Monitors control panel.</p> <p>The OK button is a standard control put in the Options dialog box by the Monitors control panel.</p>
<code>cancelMsg</code>	3	<p>The user clicked the Cancel button. Your monitors extension function should return the device that your monitors extension controls to the condition it was in before the user clicked the Options button, release any memory that your function previously allocated, and return control to the Monitors control panel.</p> <p>The Cancel button is a standard control put in the Options dialog box by the Monitors control panel.</p>
<code>hitMsg</code>	4	<p>The user clicked an enabled control in the Options dialog box, and your extension function should handle the click.</p> <p>The Monitors control panel appends your item list to the standard list of items in the Options dialog box and passes, in the <code>item</code> parameter, the item's item number in the combined list. To get the actual number of the clicked item as defined in your item list, subtract <code>numItems</code> from <code>item</code>.</p>
<code>nullMsg</code>	5	<p>A null event occurred. Your monitors extension function should perform tasks that have to be done repeatedly, if any. Do not assume any particular timing for this message.</p>
<code>updateMsg</code>	6	<p>An update event occurred. Your monitors extension function should update any user items and redraw any controls that are not standard items handled by the Dialog Manager.</p>
<code>activateMsg</code>	7	<p>An activate event occurred, indicating that the Options dialog box is becoming active. Currently, the Monitors control panel does not call your monitors extension function with this message, because the Options dialog box is modal. However, your function should handle this message as it would any activate event, because in future versions of the Operating System the Options dialog box might be modeless.</p>

Table 8-4 Messages from the Monitors control panel (continued)

Constant	Value	Description
<code>deactivateMsg</code>	8	An activate event occurred, indicating that the Options dialog box is becoming inactive. Currently, the Monitors control panel does not call your extension function with this message, because the Options dialog box is modal. However, your function should handle this message as you would any activate event, because in future versions of the Operating System the Options dialog box might be modeless.
<code>keyEvtMsg</code>	9	A keyboard event occurred. Your monitors extension function should process the keyboard event.
<code>superMsg</code>	10	<p>The user has selected superuser status. Your monitors extension function should display any controls that are reserved for superusers.</p> <p>The Monitors control panel sends this message when the user holds down the Option key while clicking the Options button.</p> <p>This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls that you have reserved for superusers, if your function has not already done this in response to either the <code>startupMsg</code> or <code>initMsg</code> message. If your code does not handle this message, it should return as its function result a handle to any previously allocated memory.</p> <p>The Monitors control panel sends this message or the normal message immediately following the initialization message.</p>
<code>normalMsg</code>	11	<p>The user is not a superuser. This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls, if your function has not already done this in response to either the <code>startupMsg</code> or <code>initMsg</code> message. If your function does not handle this message, it should return as its function result a handle to any previously allocated memory.</p> <p>The Monitors control panel sends this message or the superuser message immediately following the initialization message.</p>
<code>startupMsg</code>	12	<p>The Monitors control panel sends this message as soon as the code in your monitors code (<code>'mnr'</code>) resource has been loaded, and before the Monitors control panel finds any resources that your monitors extension function refers to. If the user is a superuser, the Monitors control panel sets the <code>item</code> parameter to 1 when it sends the startup message.</p> <p>When your monitors extension function receives this message, it can load and modify any resources that must allow for the capabilities of the system or for superusers. For example, your function can modify the item list resource to display special controls for superusers.</p>

Control Panels

Your monitors extension function can return either an error code or a value that you want to have available the next time the Monitors control panel calls your function. For example, if your monitors extension function allocates memory, it can return a handle to the memory as its function result. Each time the Monitors control panel calls your monitors extension function, the `monitorValue` parameter contains the value that your function returned the last time it was called.

Your monitors extension function must also detect and recover from any error conditions or report them to the user. If it cannot recover from an error, your monitors extension function should display an error dialog box and then return a value between 1 and 255. If your function returns a value in this range, the Monitors control panel closes the Options dialog box immediately and does not call your function again. If your function returns an error in response to the `initMsg` or `startupMsg` message, the Monitors control panel does not display the Options dialog box. Your function can display an alert box describing the error before returning control to the Monitors control panel.

SEE ALSO

For more information about the messages the Monitors control panel sends to your monitors extension function and how to handle them, see “Writing a Monitors Extension Function” beginning on page 8-61.

Resources

This section identifies the resources you supply for a control panel and monitors extension. The required resources for a control panel are

- n A machine ('mach') resource that describes the systems on which your control panel can run or signals the Finder to call your control device function to perform this check.
- n A rectangle positions ('nrct') resource to define the number of rectangles that make up the control panel and their positions.
- n An item list ('DITL') resource to specify all of the items that are to appear in the control panel. These items can include static text, buttons, checkboxes, radio buttons, editable text, the resource IDs of icons and QuickDraw pictures, and the resource IDs of other types of controls, such as pop-up menus.
- n An icon list ('ICN#') resource and other icon family resources ('ics#', 'icl8', 'icl4', 'ics8', 'ics4') to define the icons for the control panel file.
- n A control device function ('cdev') code resource that contains the code to implement the control panel.
- n A file reference ('FREF') resource to associate your control panels's icons with your control panel file so that the Finder can display the icons with the file type they represent.

Control Panels

- n A bundle ('BNDL') resource to associate your control panel's signature, icon list, and file reference resources.
- n A signature resource—defined using a string ('STR') resource—to identify your control panel.

The following required resources are described completely in chapters of *Inside Macintosh: Macintosh Toolbox Essentials* and are not included in this reference section:

- n For the item list ('DITL') resource, see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.
- n For the icon family, file reference ('FREF'), bundle ('BNDL'), and signature resources, see the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.

The two remaining required resources—machine ('mach') and rectangle positions ('nrct') resources—are described in this section. The font information ('font') resource is also covered in this section; it is an optional resource that you can supply to specify the font to be used for static text items.

Note

You can include additional resources in your control panel file that are not required. See “Providing Additional Resources for a Control Panel” on page 8-22 for more information. u

The resources required for an extension to the Monitors control panel are

- n A card ('card') resource that contains a Pascal string identical to the name of the video card. (This is the name in the declaration ROM of the card.) Because a monitors extension can include as many card resources as you like, one extension file can handle several types of video cards.
- n A monitor ('mntr') code resource that contains the code to implement and handle the controls and features of your monitors extension.
- n A rectangle ('RECT') resource to describe the size and shape of the area used to display your controls.
- n An item list ('DITL') resource to specify which items you want to appear in your monitors extension. You can add additional controls for superusers, separating them from the other controls with a horizontal dividing line.

Of these required resources, the card ('card'), monitor ('mntr'), and rectangle ('RECT') resources are described in this reference section. See the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information about the item list ('DITL') resource.

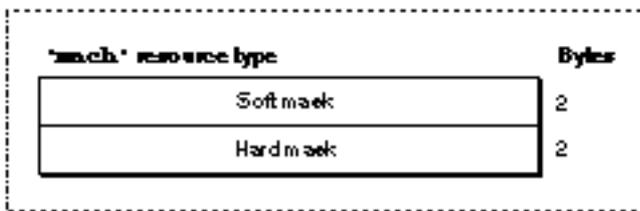
For information about the optional resources you can provide for a monitors extension, see “Supplying Optional Resources for a Monitors Extension” beginning on page 8-56.

The Machine Resource

You can identify to the Finder the hardware and software components on which your control panel runs, or you can signal the Finder to call your control device function to perform this check. In either case, create a machine resource of type 'mach'. A machine resource must have a resource ID of -4064.

The machine resource consists of two word-sized masks: a hard and a soft mask. Figure 8-16 shows the structure of a compiled machine resource.

Figure 8-16 Structure of a compiled machine ('mach') resource



A compiled version of a machine resource contains these elements:

- n Soft mask. See Table 8-5 for a description of this mask.
- n Hard mask. See Table 8-5 for a description of this mask.

The Finder performs the check if you set these masks to values representing the requirements for your control panel.

Note

In System 6, the Control Panel does not display the icon for a control panel file if the machine resource indicates that the control panel cannot run on the current system. u

If you set these masks to values indicating that the Finder is to call your control device function to perform the check, the Finder calls your function for the first time with a `macDev` message. (See “Determining If a Control Panel Can Run on the Current System” on page 8-29 for a discussion of how to handle a `macDev` message.)

Table 8-5 shows the values you use to set the machine resource masks.

Table 8-5 Possible settings for the machine resource masks

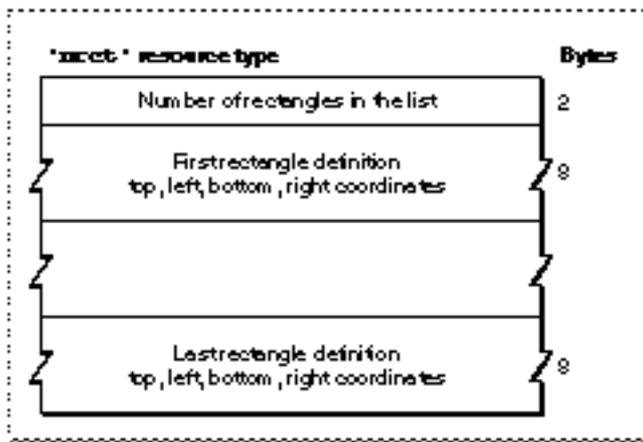
Soft mask	Hard mask	Action
\$0000	\$FFFF	The Finder calls this control device function with a macDev message, and the function must perform its own hardware and software requirements check.
\$3FFF	\$0000	This control panel runs on Macintosh II systems only.
\$7FFF	\$0400	This control panel runs on all systems with an Apple Desktop Bus (ADB).
\$FFFF	\$0000	This control panel runs on all systems.

For more information about the machine resource, see “Specifying the Machine Resource” on page 8-20.

The Rectangle Positions Resource

Your control panel can consist of one or more rectangles. To define a list of rectangles that determine the display area for your control panel, create a rectangle positions resource of type 'nrct'. A rectangle positions resource must have a resource ID of -4064. Figure 8-17 shows the structure of a compiled rectangle positions resource.

Figure 8-17 Structure of a compiled rectangle positions ('nrct') resource



Control Panels

A compiled version of a rectangle positions resource contains these elements:

- n Number of rectangles in the list.
- n Coordinates for each rectangle. You specify the coordinates as top, left, bottom, and right.

To provide for backward compatibility with the Control Panel desk accessory, the Finder accepts only the coordinates (-1,87) as the origin of a control panel. If you are designing for System 7 only, you can extend the bottom and right edges of a control panel as far as you like. If you want your control panel to run in System 7 and previous versions of system software, you must limit your control panel's size to the area bounded by (-1,87,255,322). These are the coordinates used by the Control Panel desk accessory.

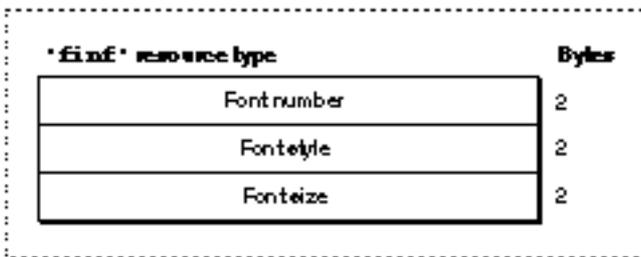
In System 6, the Control Panel desk accessory draws a frame that is 2 pixels wide around each rectangle. To join two parts of a panel neatly, overlap their rectangles by 2 pixels on the side where they meet.

For more information about the rectangle positions resource, see "Defining the Control Panel Rectangles" beginning on page 8-15.

The Font Information Resource

The Dialog Manager uses the default application font when it displays the static text items in your control panel. To specify a different font, create a font information resource of type 'finf'. A font information resource must have a resource ID of -4049. This is an optional resource for control panels. Figure 8-18 shows the structure of a compiled font information resource.

Figure 8-18 Structure of a compiled font information ('finf') resource



A font information resource contains three 2-byte words. A compiled version of a rectangle positions resource contains these elements:

- n Font ID number. The Finder sets the graphics port's `txFont` field to this value.
- n Font style. The Finder sets the graphics port's `txFace` field to this style.
- n Font size. The Finder sets the graphics port's `txSize` field to this size.

Control Panels

For more information about the font information resource, see “Specifying the Font of Text in a Control Panel” on page 8-23.

Note

The Control Panel desk accessory in System 6 does not support font information resources. If your control panel can run in System 6 and you want to specify a different font, see “Defining Text in a Control Panel as User Items” on page 8-24. u

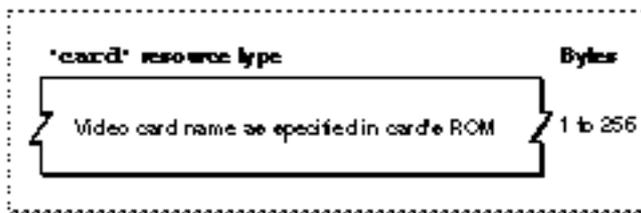
The Control Device Function Code Resource

A control device function code resource contains the code to implement a control panel and respond to messages from the Finder. A control device function code resource is a resource of type 'cdev' and must have a resource ID of -4064. This resource must begin with a control device function (see “Control Device Functions” beginning on page 8-74 for more information).

The Card Resource

A card resource specifies a video card's name. A card resource is a resource of type 'card' and must have a resource ID within the range -4080 through -4065. A card resource contains a Pascal string—that is, a length byte followed by an ASCII string—identical to the name of a video card. The name of a video card is located in the ROM of the card, as described in *Designing Cards and Drivers for the Macintosh Family*, third edition. Figure 8-19 shows the structure of a compiled card resource.

Figure 8-19 Structure of a compiled card ('card') resource



Because a monitors extension file can contain as many card resources as you wish, one extension file can handle several types of video cards. The Options dialog box displays the name in the card resource unless you also include a string ('STR#') resource in the extension file. For more information about the string resource, see “Providing an Alternative Name for a Video Card” on page 8-58.

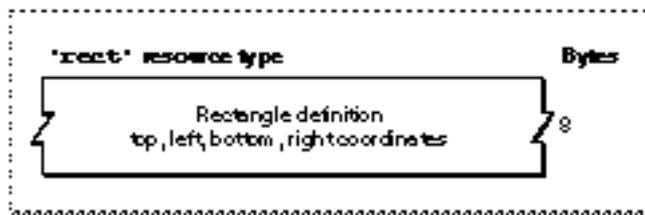
The Monitor Code Resource

A monitor code resource contains the code that carries out the functions of a monitors extension. A monitor code resource is a resource of type 'mnr' and must have a resource ID of -4096. This resource must begin with a monitors extension function that you provide. The Monitors control panel calls your monitors extension function with requests to perform an action or handle an event. A monitors extension should return as a function result a handle to memory that the function allocated or an error code. In MPW, you can set the code resource type to 'mnr' when you link the program.

The Rectangle Resource

A rectangle resource describes the display area for the controls of a monitors extension. A rectangle resource is a resource of type 'RECT' and must have a resource ID of -4096. You specify the rectangle coordinates as top, left, bottom, and right. Figure 8-20 shows the compiled version of a rectangle positions resource.

Figure 8-20 Structure of a compiled rectangle ('RECT') resource



When enlarging the Options dialog box, the Monitors control panel places the upper edge of the new display area immediately below the lower edge of the area containing the standard controls.

When you assign coordinates to your controls, assume that the origin (that is, the upper-left corner) of the display area for your items is at (0,0). In this coordinate system, the area bounding the standard controls (such as the OK and Cancel buttons) has a right coordinate of 319 and a negative top coordinate. See "Defining a Rectangle for a Monitors Extension" on page 8-52 for an example.

Before displaying the controls defined by your monitors extension, the Monitors control panel changes the coordinates of your controls, using the coordinate system of the Options dialog box. To get the true locations of your dialog items, use the Dialog Manager's `GetDialogItem` procedure; see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on this procedure.

Summary of Control Panels

Pascal Summary

Constants

CONST

```

{values for the message parameter for control device functions}
initDev      = 0;  {perform initialization}
hitDev       = 1;  {handle click in enabled item}
closeDev     = 2;  {respond to user closing the control panel}
nulDev      = 3;  {handle null event}
updateDev   = 4;  {handle update event}
activDev    = 5;  {handle activate event}
deActivDev  = 6;  {respond to control panel becoming inactive}
keyEvtDev   = 7;  {handle key-down or auto-key event}
macDev      = 8;  {check whether control panel can run }
              { on current system}
undoDev     = 9;  {handle Undo command}
cutDev      = 10; {handle Cut command}
copyDev     = 11; {handle Copy command}
pasteDev    = 12; {handle Paste command}
clearDev    = 13; {handle Clear command}
{initial value of cdevStorageValue}
cdevUnset   = 3;  {the control device function has not }
              { returned a handle}

{error codes}
cdevGenErr  = -1; {general error; no error dialog box is displayed }
              { to the user}
cdevMemErr  = 0;  {not enough memory available to continue; an }
              { out-of-memory error dialog box is displayed to }
              { the user}
cdevResErr  = 1;  {needed resource is not available or is missing; }
              { error dialog box is displayed to the user}

{values for the message parameter for a monitors extension function}
initMsg     = 1;  {perform initialization}
okMsg       = 2;  {user clicked OK button}

```

Control Panels

```

cancelMsg      = 3;  {user clicked Cancel button}
hitMsg         = 4;  {user clicked enabled control}
nulMsg         = 5;  {handle null event}
updateMsg      = 6;  {handle update event}
activateMsg    = 7;  {not used}
deactivateMsg  = 8;  {not used}
keyEvtMsg      = 9;  {handle keyboard event}
superMsg       = 10; {show superuser controls}
normalMsg      = 11; {show only normal controls}
startupMsg     = 12; {gives user status (whether a superuser)}

```

Application-Defined Routines

Control Device Functions

```

FUNCTION MyCdev      (message, item, numItems, CPrivateValue:
                    Integer;VAR theEvent: EventRecord;
                    cdevStorageValue: LongInt;
                    CPDialog: DialogPtr): LongInt;

```

Monitors Extension Functions

```

FUCNTION MyMntrExt  (message, item, numItems: Integer;
                    monitorValue: LongInt; mDialog: DialogPtr;
                    theEvent: EventRecord; screenNum: Integer;
                    VAR screens: ScrnRsrcHandle;
                    VAR scrnChanged: Boolean): LongInt;

```

C Summary

Constants

```

enum {
    /*values for the message parameter for control device functions*/
    initDev      = 0,  /*perform initialization*/
    hitDev       = 1,  /*handle click in enabled item*/
    closeDev     = 2,  /*respond to user closing control panel*/
    nulDev       = 3,  /*handle null event*/
    updateDev    = 4,  /*handle update event*/
    activDev     = 5,  /*handle activate event*/
    deActivDev   = 6,  /*respond to control panel becoming inactive*/
    keyEvtDev    = 7,  /*handle key-down or auto-key event*/

```

CHAPTER 8

Control Panels

```
macDev      = 8, /*determine whether control panel can run */
              /* on current system*/
undoDev     = 9, /*handle Undo command*/
cutDev      = 10, /*handle Cut command*/
copyDev     = 11, /*handle Copy command*/
pasteDev    = 12, /*handle Paste command*/
clearDev    = 13, /*handle Clear command*/
/*initial value of cdevStorageValue*/
cdevUnset   = 3, /*the control device function has not */
              /* returned a handle*/

/*error codes*/
cdevGenErr  = -1, /*general error; no error dialog box is displayed */
              /* to the user*/
cdevMemErr  = 0, /*not enough memory available to continue; an */
              /* out-of-memory error dialog box is displayed to */
              /* the user*/
cdevResErr  = 1 /*needed resource is not available or is missing; */
              /* error dialog box is displayed */
              /* to the user*/
};

enum {
/*values for the message parameter for a monitors extension*/
initMsg     = 1, /*perform initialization*/
okMsg       = 2, /*user clicked OK button*/
cancelMsg   = 3, /*user clicked Cancel button*/
hitMsg      = 4, /*user clicked enabled control*/
nulMsg     = 5, /*handle null event*/
updateMsg   = 6, /*update event*/
activateMsg = 7, /*not used*/
deactivateMsg = 8, /*not used*/
keyEvtMsg   = 9, /*handle keyboard event*/
superMsg    = 10, /*show superuser controls*/
normalMsg   = 11, /*show only normal controls*/
startupMsg  = 12 /*gives user status (whether a superuser)*/
};
```

Application-Defined Routines

Control Device Functions

```
pascal unsigned long MyCdev
    (short message, short item, short numItems,
     short CPrivateVal, const EventRecord *theEvent,
     unsigned long cdevStorageValue,
     DialogPtr CPDialog);
```

Monitors Extension Functions

```
pascal unsigned long MyMntrExt
    (short message, short item, short numItems,
     unsigned long monitorValue,
     DialogPtr mDialog,
     const EventRecord *theEvent, short screenNum,
     ScrnRsrcHandle screens, Boolean scrnChanged);
```

Desktop Manager

Contents

About the Desktop Database	9-4
Using the Desktop Manager	9-4
Desktop Manager Reference	9-6
Data Structure	9-6
The Desktop Parameter Block	9-7
Routines	9-8
Locating, Opening, and Closing the Desktop Database	9-9
Reading the Desktop Database	9-12
Adding to the Desktop Database	9-17
Deleting Entries From the Desktop Database	9-20
Manipulating the Desktop Database Itself	9-23
Summary of the Desktop Manager	9-27
Pascal Summary	9-27
Constants	9-27
Data Types	9-27
Routines	9-28
C Summary	9-30
Constants	9-30
Data Types	9-31
Routines	9-31
Assembly-Language Summary	9-34
Data Structures	9-34
Trap Macros	9-35
Result Codes	9-35

Desktop Manager

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 maintaining user comments associated with files and managing the icons used by applications.

S WARNING

Although there may be instances where you would like to gain access to the desktop database by using Desktop Manager routines, you should never change, add to, or remove any of this information. Manipulating the desktop database is likely to wreak havoc on your users' systems. s

In case you should 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 this chapter.

Much of the information in the desktop database comes from the bundle resources for applications and other files on the volume. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* 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 Desktop Manager provides routines that let your application retrieve this information from the desktop database.

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 one for it and stores it in the System Folder of the boot drive.

Note

If you distribute read-only media, it is generally a good idea to store on each volume both a desktop database (for users running System 7 or later) and a Desktop file (for users running older versions of system software). Create a desktop database on your master volume by pressing Command-Option when booting your system with System 7. Then create a Desktop file by pressing Command-Option and restarting your system with version 6.0. u

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 in a database.

Desktop Manager

Although the Desktop Manager provides tools for both reading and changing the desktop database, your application should not ordinarily change anything in the database. You can read the database to retrieve information, such as the icons defined by other applications.

Note

The desktop database doesn't store customized icons (that is, those with resource IDs of -16455 described in the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*), so your application can't retrieve them by using Desktop Manager routines. u

If the Translation Manager is available, the desktop database also includes information retrieved from each application's 'open' and 'kind' resources. For information about the information stored in these resources, see the chapter "Translation Manager" in this book.

About the Desktop Database

In earlier versions of system software, Finder information for each volume was stored in the volume's Desktop file, a resource file created and used by the Finder and invisible to the user. This strategy meets the needs of a single-user system with reasonably small volumes. The Desktop file is still used on ejectable volumes with a capacity less than 2 MB so that these floppy disks can be shared with Macintosh computers running earlier versions of system software. (Note, however, that resources can't be shared. Since the Finder is always running in System 7, it keeps each floppy disk's Desktop file open, so your application can't read or write to it.)

Because resources can't be shared, a different strategy has been used for AppleShare volumes, which are available to multiple users over a network. The Desktop Manager in System 7 uses the strategy for large local volumes that AppleShare file servers have previously used for shared volumes. When a volume is first mounted, the Finder collects the bundle information from all applications on the disk and builds the desktop database. Whenever an application is added to or removed from the disk, the Finder updates the desktop database. Through Desktop Manager routines, the database is also accessible to any other application running on the system.

Using the Desktop Manager

You can manipulate the desktop database with a set of low-level routines that follow the parameter-block conventions used by the File Manager. The desktop database functions use a desktop parameter block (see page 9-7 for the structure of the desktop parameter block).

Desktop Manager

Because you cannot use the Desktop Manager functions on a disk that does not have a desktop database, call `PBHGetVolParms` to verify that the target disk has a desktop database before calling any of the Desktop Manager functions. (For a description of the `PBHGetVolParms` function and the `bHasDesktopMgr` bit that you should check, see the chapter “File Manager” in *Inside Macintosh: Files*.)

Because the Finder uses the desktop database, the database is almost always open. When the Desktop Manager opens the database, it assigns the database a reference number that represents the access path. You use the `PBDTGetPath` function to get the reference number, which you must specify when calling most other Desktop Manager functions. If the desktop database is not open, `PBDTGetPath` opens it.

If you are manipulating the database in the absence of the Finder, you can open the database with `PBDTOpenInform`, which performs the same functions as `PBDTGetPath` and also sets a flag to tell your application whether the desktop database was empty when it was opened. Your application should never close the database.

The Desktop Manager provides different functions for manipulating different kinds of information in the database. Not all manipulations are possible with all kinds of data.

You can retrieve five kinds of information from the database:

- n icon definitions
- n file types and icon types supported by a known creator
- n name and location of applications with a known creator
- n user comments for a file or a directory
- n size and parent directory of the desktop database

To retrieve an icon definition, call `PBDTGetIcon`. You must specify a file creator, file type, and icon type. The database recognizes both large and small icons, with 1, 4, or 8 bits of color encoding. (For a description of these icons, see the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.)

To step through a list of all the icon types supported by an application, make repeated calls to `PBDTGetIconInfo`. Each time you call `PBDTGetIconInfo`, you specify a creator and an index value. Set the index to 1 on the first call, and increment it on each subsequent call until `PBDTGetIconInfo` returns the result code `afpItemNotFound`. For each entry in the icon list, `PBDTGetIconInfo` reports the icon type, the file type it is associated with, and the size of its icon data.

To identify the application that can open a file with a given creator, call `PBDTGetAPPL`. In each call to `PBDTGetAPPL`, you specify a creator (which is the application’s signature) and an index value. An index value of 0 retrieves the “first choice” application—that is, the one with the most recent creation date. By setting the index to 1 on the first call and incrementing it on each subsequent call until `PBDTGetAPPL` returns the result code `afpItemNotFound`, you can make multiple calls to `PBDTGetAPPL` to find information about all copies or versions of the application with this signature on the disk. `PBDTGetAPPL` returns them all in arbitrary order. `PBDTGetAPPL` returns the name, parent directory ID, and creation date of each application in the desktop database.

Desktop Manager

To retrieve the user comments for a file or directory, call `PBDTGetComment`. The user can change comments at any time by typing in the comment box of the information window for any desktop object.

Your application should not ordinarily call the functions for adding and removing data to and from the database. If your application does need to write to or delete information from the desktop database, it must call `PBDTFlush` to update the copy stored on the volume.

The following list summarizes the data manipulation functions:

Kind of data	Read	Write	Remove
Icon definitions (for a given file type and creator)	<code>PBDTGetIcon</code>	<code>PBDTAddIcon</code>	—
Icon types (associated with each file type that an application supports)	<code>PBDTGetIconInfo</code>	—	—
Applications with a given creator	<code>PBDTGetAPPL</code>	<code>PBDTAddAPPL</code>	<code>PBDTRemoveAPPL</code>
User comments	<code>PBDTGetComment</code>	<code>PBDTSetComment</code>	<code>PBDTRemoveComment</code>
Entire desktop database	<code>PBDTGetInfo</code> (returns the size and parent directory of the database)	<code>PBDTFlush</code> (updates the copy stored on the volume)	<code>PBDTDelete</code> and <code>PBDTReset</code> (neither should be called by your application)

Desktop Manager Reference

This section describes the data structure and routines that are specific to the Desktop Manager. The “Data Structure” section describes the desktop parameter block, and the “Routines” section describes the routines your application can use to retrieve information from the desktop database.

Data Structure

You can manipulate the desktop database with a set of low-level routines that follow the parameter-block conventions used by the File Manager. (For more information on parameter blocks, see the chapter “File Manager” in *Inside Macintosh: Files*.) This section describes the parameter block you pass to Desktop Manager routines.

The Desktop Parameter Block

The desktop database functions use the desktop parameter block, a data structure of type DTPBRec:

```

TYPE DTPBRec =
  RECORD
    qLink:          QElemPtr;  {next queue entry}
    qType:          Integer;   {queue type}
    ioTrap:         Integer;   {routine trap}
    ioCmdAddr:      Ptr;       {routine address}
    ioCompletion:   ProcPtr;   {completion routine}
    ioResult:       OSErr;     {result code}
    ioNamePtr:      StringPtr; {file, directory, or }
                          { volume name}
    ioVRefNum:      Integer;   {volume reference number}
    ioDTRefNum:     Integer;   {desktop database reference }
                          { number}
    ioIndex:        Integer;   {index into icon list}
    ioTagInfo:      LongInt;   {tag information}
    ioDTBuffer:     Ptr;       {data buffer}
    ioDTReqCount:   LongInt;   {requested length of data}
    ioDTActCount:   LongInt;   {actual length of data}
    filler1:        SignedByte; {unused}
    ioIconType:     SignedByte; {icon type}
    filler2:        Integer;   {unused}
    ioDirID:        LongInt;   {parent directory ID}
    ioFileCreator:  OSType;    {file creator}
    ioFileType:     OSType;    {file type}
    ioFiller3:      LongInt;   {unused}
    ioDTLgLen:      LongInt;   {logical length of desktop }
                          { database}
    ioDTPyLen:      LongInt;   {physical length of desktop }
                          { database}
    ioFiller4:      LongInt;   {unused}
    ioAPPLParID:    LongInt     {parent directory ID of }
                          { application}
  END;
DTPBPtr = ^DTPBRec;          {pointer to desktop }
                              { parameter block}

```

For a description of the standard fields of a parameter block (`qLink`, `qType`, `ioTrap`, `ioCmdAddr`, `ioCompletion`, and `ioResult`), see the chapter “File Manager” in *Inside Macintosh: Files*. For other fields of the desktop parameter block, see the relevant routine description provided in the next section.

Routines

This section describes the low-level routines for using the desktop database.

All low-level routines exchange parameters with your application through a parameter block. When calling a low-level routine, you pass a pointer to the parameter block. See the chapter “File Manager” in *Inside Macintosh: Files* for a description of the standard fields in a parameter block.

IMPORTANT

Clear all fields (other than input fields) in the parameter block that you pass to Desktop Manager routines. *s*

Three Desktop Manager functions—namely, `PBDTGetPath`, `PBDTOpenInform`, and `PBDTCloseDown`—run synchronously only. All other Desktop Manager routines can run either asynchronously or synchronously. There are three versions of each of these routines. The first version takes two parameters: a pointer to the parameter block, and a Boolean value that determines whether the routine is run asynchronously (`TRUE`) or synchronously (`FALSE`). Here, for example, is the first version of a routine that retrieves the user’s comment stored for a file or a directory:

```
FUNCTION PBDTGetComment (paramBlock: DTPBPtr;
                        async: Boolean): OSErr;
```

The second version does not take a second parameter; instead, it adds the suffix “Async” to the name of the routine.

```
FUNCTION PBDTGetCommentAsync (paramBlock: DTPBPtr): OSErr;
```

Similarly, the third version of the routine does not take a second parameter; instead, it adds the suffix “Sync” to the name of the routine.

```
FUNCTION PBDTGetCommentSync (paramBlock: DTPBPtr): OSErr;
```

All routines in this section are documented using the first version only. Note, however, that the second and third versions of these routines do not use the glue code that the first versions use and are therefore more efficient.

IMPORTANT

All of the Desktop Manager routines may move or purge memory blocks in the application heap or for some other reason should not be called from within an interrupt. Your application should not call Desktop Manager routines at interrupt time. *s*

Desktop Manager

Because you cannot use the Desktop Manager functions on a disk that does not have a desktop database, call `PBGetVolParms` to verify that the target disk has a desktop database before calling any of the Desktop Manager functions. (For a description of the `PBGetVolParms` function and the `bHasDesktopMgr` bit that you should check, see the chapter “File Manager” in *Inside Macintosh: Files*.)

S WARNING

Although routines that set information in and get information from the desktop database are described in this section, you should never use these routines to change, add to, or remove any information from the desktop database. Manipulating the desktop database is likely to wreak havoc on your users' systems. *s*

Assembly-Language Note

You can invoke each of the Desktop Manager routines with a macro that has the same name as the routine, preceded by an underscore. These macros, however, aren't really trap macros. Instead, they expand to invoke the trap macro `_HFSDispatch`. The File Manager determines which routine to execute from the routine selector, an integer placed in register D0. The routine selectors appear in “Assembly-Language Summary” beginning on page 9-34. *u*

Locating, Opening, and Closing the Desktop Database

To get the access path to a database or to create a database if one doesn't exist, use the `PBDTGetPath` or `PBDTOpenInform` function. These routines run synchronously only. System software uses the `PBDTCloseDown` function to close the desktop database; your application should never use this function, which is described in this section only for completeness.

PBDTGetPath

You can get the reference number of the desktop database using the `PBDTGetPath` function.

```
FUNCTION PBDTGetPath (paramBlock: DTPBPtr): OSErr;
```

`paramBlock` A pointer to a desktop parameter block.

Parameter block

<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioNamePtr</code>	<code>StringPtr</code>	A pointer to the volume name or full pathname of the desktop database.
<code>ioVRefNum</code>	<code>Integer</code>	The volume reference number of the desktop database.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.

DESCRIPTION

The `PBDTGetPath` function returns the desktop database reference number in the `ioDTRefNum` field, which represents the access path to the database. You specify the volume by passing a pointer to its name in the `ioNamePtr` field or a volume reference number in the `ioVRefNum` field. If the desktop database is not already open, `PBDTGetPath` opens it and then returns the reference number. If the desktop database doesn't exist, `PBDTGetPath` creates it. If `PBDTGetPath` fails, it sets the `ioDTRefNum` field to 0.

Note

You cannot use the desktop reference number as a file reference number in any File Manager routines. [u](#)

s WARNING

Do not call `PBDTGetPath` at interrupt time—it allocates memory in the system heap. [s](#)

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>extFSErr</code>	-58	External file system—file system identifier is nonzero
<code>desktopDamagedErr</code>	-1305	The desktop database has become corrupted—the Finder will fix this, but if your application is not running with the Finder, use <code>PBDTReset</code> or <code>PBDTDelete</code>

PBDTOpenInform

The `PBDTOpenInform` function performs the same function as `PBDTGetPath`, but it also reports whether the desktop database was empty when it was opened.

```
FUNCTION PBDTOpenInform (paramBlock: DTPBPtr): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

Parameter block

<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioNamePtr</code>	<code>StringPtr</code>	A pointer to the volume name or full pathname of the desktop database.
<code>ioVRefNum</code>	<code>Integer</code>	The volume reference number of the desktop database.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioTagInfo</code>	<code>LongInt</code>	The return flag (in low bit).

DESCRIPTION

If the desktop database was just created in response to `PBDTOpenInform` (and is therefore empty), `PBDTOpenInform` sets the low bit in the `ioTagInfo` field to 0. If the desktop database had been created before you called `PBDTOpenInform`, `PBDTOpenInform` sets the low bit in the `ioTagInfo` field to 1.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>paramErr</code>	-50	Parameter error; use <code>PBDTGetPath</code>
<code>extFSErr</code>	-58	External file system—file system identifier is nonzero
<code>desktopDamagedErr</code>	-1305	The desktop database has become corrupted—the Finder will fix this, but if your application is not running with the Finder, use <code>PBDTReset</code> or <code>PBDTDelete</code>

PBDTCloseDown

The `PBDTCloseDown` function is used by system software to close the desktop database, though your application should never do this itself. `PBDTCloseDown` runs synchronously only, and though it will not close down the desktop databases of remote volumes, it will invalidate all local `DTRefNum` values for remote desktop databases.

```
FUNCTION PBDTCloseDown (paramBlock: DTPBPtr): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

Parameter block

<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.

DESCRIPTION

The `PBDTCloseDown` function closes the database specified in `ioDTRefNum` and frees all resources allocated by `PBDTOpenInform` or `PBDTGetPath`.

S WARNING

Applications should not call `PBDTCloseDown`. The system software closes the database when the volume is unmounted. s

RESULT CODES

noErr	0	No error
ioErr	-36	I/O error
rfNumErr	-51	Reference number invalid
extFSErr	-58	External file system—file system identifier is nonzero

Reading the Desktop Database

You can get information from the desktop database, such as a specific icon that represents a file of a given type and creator or the user comments associated with a file, by using the routines described in this section.

PBDTGetIcon

To retrieve an icon definition, use the `PBDTGetIcon` function.

```
FUNCTION PBDTGetIcon (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

paramBlock

A pointer to a desktop parameter block.

async

A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

ioCompletion	ProcPtr	A pointer to a completion routine.
ioResult	OSErr	The result code of the function.
ioDTRefNum	Integer	The desktop database reference number.
ioTagInfo	LongInt	Reserved; must be set to 0.
ioDTBuffer	Ptr	A pointer to a buffer to hold the icon's data.
ioDTReqCount	LongInt	The requested size of the icon's bitmap.
ioDTActCount	LongInt	The actual size of the icon's bitmap.
ioIconType	SignedByte	The icon type.
ioFileCreator	OSType	The icon's file creator.
ioFileType	OSType	The icon's file type.

DESCRIPTION

The `PBDTGetIcon` function returns the bitmap for an icon that represents a file of a given type and creator. (For example, to get the icon for a file of file type 'SFWR' created by the application with a signature of 'WAVE', specify these two values in `ioFileType` and `ioFileCreator`.) You pass a pointer to a buffer for the icon bitmap in the `ioDTBuffer` field. The bitmap is returned in this buffer. You specify the desktop

Desktop Manager

database in `ioDTRefNum`, the file creator in `ioFileCreator`, and the file type in `ioFileType`. For the icon type in `ioIconType`, specify a constant from the following list:

Constant	Value	Corresponding resource type	Description
<code>kLargeIcon</code>	1	'ICN#'	Large black-and-white icon with mask
<code>kLarge4BitIcon</code>	2	'icl4'	Large 4-bit color icon
<code>kLarge8BitIcon</code>	3	'icl8'	Large 8-bit color icon
<code>kSmallIcon</code>	4	'ics#'	Small black-and-white icon with mask
<code>kSmall4BitIcon</code>	5	'ics4'	Small 4-bit color icon
<code>kSmall8BitIcon</code>	6	'ics8'	Small 8-bit color icon

The value you supply in `ioDTReqCount` is the size in bytes of the buffer that you've allocated for the icon's bitmap pointed to by `ioDTBuffer`; this value depends on the icon type. Be sure to allocate enough storage for the icon data; 1024 bytes is the largest amount required for any icon in System 7. You can use constants to indicate the amount of memory you have provided for the icon's data. The following list shows these constants and, for each icon type, shows the amount of bytes you should allocate.

Constant	Value (bytes)	Corresponding resource type	Description
<code>kLargeIconSize</code>	256	'ICN#'	Large black-and-white icon with mask
<code>kLarge4BitIconSize</code>	512	'icl4'	Large 4-bit color icon
<code>kLarge8BitIconSize</code>	1024	'icl8'	Large 8-bit color icon
<code>kSmallIconSize</code>	64	'ics#'	Small black-and-white icon with mask
<code>kSmall4BitIconSize</code>	128	'ics4'	Small 4-bit color icon
<code>kSmall8BitIconSize</code>	256	'ics8'	Small 8-bit color icon

The value in `ioDTActCount` reflects the size of the bitmap actually retrieved. If `ioDTActCount` is larger than `ioDTReqCount`, only the amount of data allowed by `ioDTReqCount` is valid.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSErr</code>	-58	External file system—file system identifier is nonzero
<code>afpItemNotFound</code>	-5012	Information not found

PBDTGetIconInfo

You can iteratively generate a list of icon types associated with each file type supported by an application by repeatedly calling the `PBDTGetIconInfo` function.

```
FUNCTION PBDTGetIconInfo (paramBlock: DTPBPtr;
                        async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioIndex</code>	<code>Integer</code>	An index into the icon list.
<code>ioTagInfo</code>	<code>LongInt</code>	Reserved; must be set to 0.
<code>ioDTActCount</code>	<code>LongInt</code>	The size of the icon's bitmap.
<code>ioIconType</code>	<code>SignedByte</code>	The icon type.
<code>ioFileCreator</code>	<code>OSType</code>	The icon's file creator.
<code>ioFileType</code>	<code>OSType</code>	The icon's file type.

DESCRIPTION

The `PBDTGetIconInfo` function retrieves an icon type and the associated file type supported by a given creator in the database. You use it to identify the set of icons associated with each file type that is supported by a given creator. You specify the creator by placing its signature in `ioFileCreator`, and you specify the database by placing the desktop database reference number in the `ioDTRefNum` field. The `PBDTGetIconInfo` function returns the size of the bitmap in `ioDTActCount`, the file type in `ioFileType`, and the icon size and color depth in `ioIconType`.

The `PBDTGetIconInfo` function can return in the `ioIconType` field any of the values listed in the description of the `PBDTGetIcon` function on page 9-12. Ignore any values returned in `ioIconType` that are not listed there; they represent special icons and information used only by the Finder.

To step through a list of the icon types and file types supported by an application, make repeated calls to `PBDTGetIconInfo`, specifying a creator and an index value for `ioIndex` each call. Set the index to 1 on the first call, and increment it on each subsequent call until `ioResult` returns `afpItemNotFound`.

RESULT CODES

noErr	0	No error
ioErr	-36	I/O error
rfNumErr	-51	Reference number invalid
extFSerr	-58	External file system—file system identifier is nonzero
afpItemNotFound	-5012	Information not found

SEE ALSO

To get a list of file types that an application can natively open, you can use the `GetFileTypesThatAppCanNativelyOpen` function, as described in the chapter “Translation Manager” of this book.

PBDTGetAPPL

To identify the application that can open a file with a given creator, use the `PBDTGetAPPL` function.

```
FUNCTION PBDTGetAPPL (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

paramBlock

A pointer to a desktop parameter block.

async

A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

ioCompletion	ProcPtr	A pointer to a completion routine.
ioResult	OSErr	The result code.
ioNamePtr	StringPtr	A pointer to the application's name.
ioDTRefNum	Integer	The desktop database reference number.
ioIndex	Integer	An index into the application list.
ioTagInfo	LongInt	The application's creation date.
ioFileCreator	OSType	The application's signature.
ioAPPLParID	LongInt	The application's parent directory.

DESCRIPTION

For an application in the database specified in `ioDTRefNum` with the signature specified in `ioFileCreator`, `PBDTGetAPPL` returns the filename in `ioNamePtr`, the parent directory ID in `ioAPPLParID`, and the creation date in `ioTagInfo`. A single call, with `ioIndex` set to 0, finds the application file with the most recent creation date. If you want to retrieve information about all copies of the application with the given signature, start with `ioIndex` set to 1 and increment until `ioResult` returns `afpItemNotFound`; when called multiple times in this fashion, `PBDTGetAPPL` returns information about all the application's copies, including the file with the most recent creation date, in arbitrary order.

RESULT CODES

noErr	0	No error
ioErr	-36	I/O error
rfNumErr	-51	Reference number invalid
extFSErr	-58	External file system—file system identifier is nonzero
afpItemNotFound	-5012	Information not found

PBDTGetComment

To retrieve the user comments for a file or directory, use the `PBDTGetComment` function.

```
FUNCTION PBDTGetComment (paramBlock: DTPBPtr;
                        async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioNamePtr</code>	<code>StringPtr</code>	A pointer to a file or directory name.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioDTBuffer</code>	<code>Ptr</code>	A pointer to comment text (200 bytes).
<code>ioDTActCount</code>	<code>LongInt</code>	The comment size.
<code>ioDirID</code>	<code>LongInt</code>	The parent directory of the file or directory.

DESCRIPTION

The `PBDTGetComment` function retrieves the comment stored for a file or directory in the database specified in `ioDTRefNum`. You specify the filename or directory name and its parent directory ID through `ioNamePtr` and `ioDirID`. You allocate a buffer big enough to hold the largest comment, 200 bytes, and put a pointer to it in the `ioDTBuffer` field. The `PBDTGetComment` function places the comment in the buffer as a plain text string and places the length of the comment in `ioDTActCount`.

RESULT CODES

noErr	0	No error
ioErr	-36	I/O error
fnfErr	-43	File or directory doesn't exist
rfNumErr	-51	Reference number invalid
extFSErr	-58	External file system—file system identifier is nonzero
afpItemNotFound	-5012	Information not found

Adding to the Desktop Database

Your application should not ordinarily call the functions for adding data to the database. If your application does need to write to or delete information from the desktop database, it must call `PBDTFlush` to update the copy stored on the volume.

PBDTAddIcon

To add an icon definition to the desktop database, use the `PBDTAddIcon` function.

```
FUNCTION PBDTAddIcon (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioTagInfo</code>	<code>LongInt</code>	Reserved; must be set to 0.
<code>ioDTBuffer</code>	<code>Ptr</code>	A pointer to the icon's data.
<code>ioDTReqCount</code>	<code>LongInt</code>	The size of the icon's bitmap.
<code>ioIconType</code>	<code>SignedByte</code>	The icon type.
<code>ioFileCreator</code>	<code>OSType</code>	The icon's file creator.
<code>ioFileType</code>	<code>OSType</code>	The icon's file type.

DESCRIPTION

The `PBDTAddIcon` function adds an icon definition to the desktop database specified in `ioDTRefNum`. You specify the creator and file type that the icon is associated with in the `ioFileCreator` and `ioFileType` fields. For the icon type in `ioIconType`, specify either a constant or a value from the following list.

Constant	Value	Corresponding resource type	Description
<code>kLargeIcon</code>	1	'ICN#'	Large black-and-white icon with mask
<code>kLarge4BitIcon</code>	2	'icl4'	Large 4-bit color icon
<code>kLarge8BitIcon</code>	3	'icl8'	Large 8-bit color icon
<code>kSmallIcon</code>	4	'ics#'	Small black-and-white icon with mask
<code>kSmall4BitIcon</code>	5	'ics4'	Small 4-bit color icon
<code>kSmall8BitIcon</code>	6	'ics8'	Small 8-bit color icon

Desktop Manager

The value you supply in `ioDTReqCount` is the size in bytes of the buffer that you've allocated for the icon's bitmap pointed to by `ioDTBuffer`; this value depends on the icon type. Be sure to allocate enough storage for the icon data; 1024 bytes is the largest amount required for any icon in System 7. For the number of bytes in `ioDTReqCount`, you specify either a constant or a value from the following list.

Constant	Value (bytes in bitmap)	Corresponding resource type	Description
<code>kLargeIconSize</code>	256	'ICN#'	Large black-and-white icon with mask
<code>kLarge4BitIconSize</code>	512	'icl4'	Large 4-bit color icon
<code>kLarge8BitIconSize</code>	1024	'icl8'	Large 8-bit color icon
<code>kSmallIconSize</code>	64	'ics#'	Small black-and-white icon with mask
<code>kSmall4BitIconSize</code>	128	'ics4'	Small 4-bit color icon
<code>kSmall8BitIconSize</code>	256	'ics8'	Small 8-bit color icon

You pass a pointer to the icon bitmap in the `ioDTBuffer` field. You must initialize the `ioTagInfo` field to 0.

If the database already contains an icon definition for an icon of that type, file type, and file creator, the new definition replaces the old.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>wPrErr</code>	-44	Volume is locked through hardware
<code>vLckdErr</code>	-46	Volume is locked through software
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSErr</code>	-58	External file system—file system identifier is nonzero
<code>afpIconTypeError</code>	-5030	Sizes of new icon and one it replaces don't match

PBDTAddAPPL

To add an application to the desktop database, use the `PBDTAddAPPL` function.

```
FUNCTION PBDTAddAPPL (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Desktop Manager

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioNamePtr</code>	<code>StringPtr</code>	A pointer to the application's name.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioTagInfo</code>	<code>LongInt</code>	Reserved; must be set to 0.
<code>ioDirID</code>	<code>LongInt</code>	The application's parent directory.
<code>ioFileCreator</code>	<code>OSType</code>	The application's signature.

DESCRIPTION

The `PBDTAddAPPL` function adds an entry in the desktop database specified in `ioDTRefNum` for an application with the specified signature. You pass the application's signature in `ioFileCreator`, a pointer to the application's filename in `ioNamePtr`, and the application's parent directory ID in `ioDirID`. Initialize `ioTagInfo` to 0.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>fnfErr</code>	-43	Application not present on volume
<code>wPrErr</code>	-44	Volume is locked through hardware
<code>vLckdErr</code>	-46	Volume is locked through software
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSerr</code>	-58	External file system—file system identifier is nonzero

PBDTSetComment

To add a user comment for a file or a directory to the desktop database, use the `PBDTSetComment` function.

```
FUNCTION PBDTSetComment (paramBlock: DTPBPtr;
                        async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Desktop Manager

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioNamePtr</code>	<code>StringPtr</code>	A pointer to a file or directory name.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioDTBuffer</code>	<code>Ptr</code>	A pointer to the comment text.
<code>ioDTReqCount</code>	<code>LongInt</code>	The comment length.
<code>ioDirID</code>	<code>LongInt</code>	The parent directory of the file or directory.

DESCRIPTION

The `PBDTSetComment` function establishes the user comment associated with a file or directory in the database specified in `ioDTRefNum`. You specify the object name through `ioNamePtr` and the parent directory ID in `ioDirID`. You put the comment as a plain text string in a buffer pointed to by `ioDTBuffer`, and you specify the length of the buffer (in bytes) in `ioDTReqCount`. The maximum length of a comment is 200 bytes; longer comments are truncated. Since the comment is a plain text string and not a Pascal string, the Desktop Manager relies on the value in `ioDTReqCount` for determining the length of the buffer.

If the specified object already has a comment in the database, the new comment replaces the old.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>fnfErr</code>	-43	File or directory doesn't exist
<code>wPrErr</code>	-44	Volume is locked through hardware
<code>vLckdErr</code>	-46	Volume is locked through software
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSerr</code>	-58	External file system—file system identifier is nonzero

Deleting Entries From the Desktop Database

Your application should not ordinarily call the functions for adding and removing data to and from the database. If your application does need to write to or delete information from the desktop database, it must call `PBDTFlush` to update the copy stored on the volume.

PBDTRemoveAPPL

To remove an application from the desktop database, call the `PBDTRemoveAPPL` function.

```
FUNCTION PBDTRemoveAPPL (paramBlock: DTPBPtr;
                        async: Boolean): OSerr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSerr</code>	The result code of the function.
<code>ioNamePtr</code>	<code>StringPtr</code>	A pointer to the application's name.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioDirID</code>	<code>LongInt</code>	The application's parent directory.
<code>ioFileCreator</code>	<code>OSType</code>	The application's signature.

DESCRIPTION

The `PBDTRemoveAPPL` function removes the mapping information for an application from the database specified in `ioDTRefNum`. You specify the application's name through `ioNamePtr`, its parent directory ID in `ioDirID`, and its signature in `ioFileCreator`.

You can call `PBDTRemoveAPPL` even if the application is not present on the volume.

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>ioErr</code>	<code>-36</code>	I/O error
<code>wPrErr</code>	<code>-44</code>	Volume is locked through hardware
<code>vLckdErr</code>	<code>-46</code>	Volume is locked through software
<code>rfNumErr</code>	<code>-51</code>	Reference number invalid
<code>extFSerr</code>	<code>-58</code>	External file system—file system identifier is nonzero
<code>afpItemNotFound</code>	<code>-5012</code>	Application not found

PBDTRemoveComment

To remove a user comment from the desktop database, call the `PBDTRemoveComment` function.

```
FUNCTION PBDTRemoveComment (paramBlock: DTPBPtr;
                             async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioNamePtr</code>	<code>StringPtr</code>	A pointer to the filename or directory name.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioDirID</code>	<code>LongInt</code>	The parent directory of the file or directory.

DESCRIPTION

The `PBDTRemoveComment` function removes the comment associated with a file or directory from the database specified in `ioDTRefNum`. You specify the file or directory name through `ioNamePtr` and the parent directory ID in `ioDirID`. You cannot remove a comment if the file or directory is not present on the volume. If no comment was stored for the file, `PBDTRemoveComment` returns an error.

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>ioErr</code>	<code>-36</code>	I/O error
<code>fnfErr</code>	<code>-43</code>	File or directory doesn't exist
<code>wPrErr</code>	<code>-44</code>	Volume is locked through hardware
<code>vLckdErr</code>	<code>-46</code>	Volume is locked through software
<code>rfNumErr</code>	<code>-51</code>	Reference number invalid
<code>extFSerr</code>	<code>-58</code>	External file system—file system identifier is nonzero
<code>afpItemNotFound</code>	<code>-5012</code>	Comment not found

Manipulating the Desktop Database Itself

If your application adds information to or removes information from the desktop database, use the `PBDTFlush` function to save your changes. To get information about the desktop database itself, use the `PBDTGetInfo` function. The `PBDTReset` function removes all information from the desktop database, and the `PBDTDelete` function removes the desktop database; you should not use these two functions unless absolutely necessary.

PBDTFlush

To save your changes to the desktop database, use the `PBDTFlush` function.

```
FUNCTION PBDTFlush (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.

DESCRIPTION

The `PBDTFlush` function writes the contents of the desktop database specified in `ioDTRefNum` to the volume.

Note

If your application has manipulated information in the database using any of the routines described in “Adding to the Desktop Database” or “Deleting Entries From the Desktop Database” beginning on page 9-17 and page 9-20, respectively, you must call `PBDTFlush` to update the copy stored on the volume. ^u

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>ioErr</code>	<code>-36</code>	I/O error
<code>wPrErr</code>	<code>-44</code>	Volume is locked through hardware
<code>vLckdErr</code>	<code>-46</code>	Volume is locked through software
<code>rfNumErr</code>	<code>-51</code>	Reference number invalid
<code>extFSErr</code>	<code>-58</code>	External file system—file system identifier is nonzero

PBDTGetInfo

To determine the parent directory and the amount of space used by the desktop database on a particular volume, use the `PBDTGetInfo` function.

```
FUNCTION PBDTGetInfo (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioVRefNum</code>	<code>Integer</code>	The volume reference number where the database files are actually stored.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioIndex</code>	<code>Integer</code>	The number of files comprising the desktop database on the volume.
<code>ioDirID</code>	<code>LongInt</code>	The parent directory of the desktop database.
<code>ioDTLgLen</code>	<code>LongInt</code>	The logical length of the database files.
<code>ioDTPyLen</code>	<code>LongInt</code>	The physical length of the database files.

DESCRIPTION

The `PBDTGetInfo` function returns information about the desktop database. You specify the volume of the desktop database in `ioDTRefNum`. The parent directory of the desktop database for the volume is returned in `ioDirID`. The sum of the logical lengths of the files that constitute the desktop database for a given volume is returned in `ioDTLgLen`; the sum of the physical lengths of the files that constitute the desktop database for a given volume is returned in `ioDTPyLen`. The number of files comprising the desktop database is returned in `ioIndex`.

RESULT CODES

<code>noErr</code>	0	No err
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSErr</code>	-58	External file system—file system identifier is nonzero

PBDTReset

The `PBDTReset` function removes information from the desktop database. Unless you are manipulating the desktop database in the absence of the Finder, you should never use this function.

```
FUNCTION PBDTReset (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioDTRefNum</code>	<code>Integer</code>	The desktop database reference number.
<code>ioIndex</code>	<code>Integer</code>	Reserved; must be set to 0.

DESCRIPTION

The `PBDTReset` function removes all icons, application mappings, and comments from the desktop database specified in `ioDTRefNum`. You can call `PBDTReset` only when the database is open. It remains open after the data is cleared.

IMPORTANT

Your application should never call `PBDTReset`.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>wPrErr</code>	-44	Volume is locked through hardware
<code>vLckdErr</code>	-46	Volume is locked through software
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSerr</code>	-58	External file system—file system identifier is nonzero

PBDTDelete

The `PBDTDelete` function removes the desktop database. Unless you are manipulating the desktop database in the absence of the Finder, you should never use this function.

```
FUNCTION PBDTDelete (paramBlock: DTPBPtr; async: Boolean): OSErr;
```

`paramBlock`

A pointer to a desktop parameter block.

`async`

A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

<code>ioCompletion</code>	<code>ProcPtr</code>	A pointer to a completion routine.
<code>ioResult</code>	<code>OSErr</code>	The result code of the function.
<code>ioVRefNum</code>	<code>Integer</code>	The volume reference number of the desktop database.
<code>ioIndex</code>	<code>Integer</code>	Reserved; must be set to 0.

DESCRIPTION

The `PBDTDelete` function removes the desktop database from a local volume. You specify the volume by passing a volume reference number in `ioVRefNum`. You can call `PBDTDelete` only when the database is closed.

IMPORTANT

Your application should never call `PBDTDelete`.

RESULT CODES

<code>noErr</code>	0	No error
<code>ioErr</code>	-36	I/O error
<code>wPrErr</code>	-44	Volume is locked through hardware
<code>vLckdErr</code>	-46	Volume is locked through software
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSErr</code>	-58	External file system—file system identifier is nonzero

Summary of the Desktop Manager

Pascal Summary

Constants

CONST

```

    {for mapping icons to ioIconType in the desktop database}
    kLargeIcon           = 1;           {'ICN#'}
    kLarge4BitIcon       = 2;           {'icl4'}
    kLarge8BitIcon       = 3;           {'icl8'}
    kSmallIcon           = 4;           {'ics#'}
    kSmall4BitIcon       = 5;           {'ics4'}
    kSmall8BitIcon       = 6;           {'ics8'}

    {for allocating storage for icon data in the desktop database}
    kLargeIconSize       = 256;         {'ICN#'}
    kLarge4BitIconSize   = 512;         {'icl4'}
    kLarge8BitIconSize   = 1024;        {'icl8'}
    kSmallIconSize       = 64;          {'ics#'}
    kSmall4BitIconSize   = 128;         {'ics4'}
    kSmall8BitIconSize   = 256;         {'ics8'}
  
```

Data Types

```

TYPE  DTPBPtr = ^DTPBRec;
      DTPBRec =                                     {parameter block for desktop database}
      RECORD
        qLink:           QElemPtr;                 {next queue entry}
        qType:           Integer;                  {queue type}
        ioTrap:          Integer;                  {routine trap}
        ioCmdAddr:       Ptr;                      {routine address}
        ioCompletion:    ProcPtr;                  {completion routine}
        ioResult:        OSErr;                    {result code}
        ioNamePtr:       StringPtr;                {file, directory, or volume name}
        ioVRefNum:       Integer;                  {volume reference number}
        ioDTRefNum:      Integer;                  {desktop database reference number}
        ioIndex:         Integer;                  {index into icon list}
      
```

Desktop Manager

```

ioTagInfo:      LongInt;      {tag information}
ioDTBuffer:     Ptr;          {data buffer}
ioDTReqCount:   LongInt;      {requested length of data}
ioDTActCount:   LongInt;      {actual length of data}
filler1:        SignedByte;   {unused}
ioIconType:     SignedByte;    {icon type}
filler2:        Integer;       {unused}
ioDirID:        LongInt;       {parent directory ID}
ioFileCreator: OSType;         {file creator}
ioFileType:     OSType;         {file type}
ioFiller3:      LongInt;       {unused}
ioDTLgLen:      LongInt;       {logical length of desktop }
                                     { database}
ioDTPyLen:      LongInt;       {physical length of desktop }
                                     { database}
ioFiller4:      Integer;       {unused}
ioAPPLParID:    LongInt        {parent directory ID of }
                                     { application}

END;
DTPBPtr = ^DTPBRec;              {pointer to desktop }
                                     { parameter block}

```

Routines
Locating, Opening, and Closing the Desktop Database

```

FUNCTION PBDTGetPath      (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTOpenInform   (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTCloseDown    (paramBlock: DTPBPtr): OSErr;

```

Reading the Desktop Database

```

FUNCTION PBDTGetIcon      (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTGetIconAsync (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetIconSync  (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetIconInfo  (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTGetIconInfoAsync
    (paramBlock: DTPBPtr): OSErr;

```

Desktop Manager

```

FUNCTION PBDTGetIconInfoSync
                                (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetAPPL             (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTGetAPPLAsync       (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetAPPLSync        (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetComment         (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTGetCommentAsync    (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetCommentSync     (paramBlock: DTPBPtr): OSErr;

```

Adding to the Desktop Database

```

FUNCTION PBDTAddIcon            (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTAddIconAsync      (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTAddIconSync       (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTAddAPPL           (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTAddAPPLAsync      (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTAddAPPLSync       (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTSetComment        (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTSetCommentAsync   (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTSetCommentSync    (paramBlock: DTPBPtr): OSErr;

```

Deleting Entries From the Desktop Database

```

FUNCTION PBDTRemoveAPPL        (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTRemoveAPPLAsync   (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTRemoveAPPLSync    (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTRemoveComment     (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTRemoveCommentAsync (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTRemoveCommentSync (paramBlock: DTPBPtr): OSErr;

```

Manipulating the Desktop Database Itself

```

FUNCTION PBDTFlush          (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTFlushAsync    (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTFlushSync     (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetInfo       (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTGetInfoAsync  (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTGetInfoSync   (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTReset        (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTResetAsync    (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTResetSync     (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTDelete       (paramBlock: DTPBPtr; async: Boolean): OSErr;
FUNCTION PBDTDeleteAsync   (paramBlock: DTPBPtr): OSErr;
FUNCTION PBDTDeleteSync    (paramBlock: DTPBPtr): OSErr;

```

C Summary

Constants

```

enum {
    /*for mapping icons to ioIconType in the desktop database*/
    kLargeIcon          = 1,      /*'ICN#'*/
    kLarge4BitIcon      = 2,      /*'icl4'*/
    kLarge8BitIcon      = 3,      /*'icl8'*/
    kSmallIcon          = 4,      /*'ics#'*/
    kSmall4BitIcon      = 5,      /*'ics4'*/
    kSmall8BitIcon      = 6,      /*'ics8'*/

    /*for allocating storage for icon data in the desktop database*/
    kLargeIconSize      = 256,    /*'ICN#'*/
    kLarge4BitIconSize  = 512,    /*'icl4'*/
    kLarge8BitIconSize  = 1024,   /*'icl8'*/
    kSmallIconSize      = 64,     /*'ics#'*/
    kSmall4BitIconSize  = 128,    /*'ics4'*/
    kSmall8BitIconSize  = 256     /*'ics8'*/
};

```

Data Types

```

struct DTPBRec {          /*parameter block for desktop database*/
    ParamBlockHeader
    short    ioDTRefNum;      /*desktop refnum*/
    short    ioIndex;        /*index into icon list*/
    long     ioTagInfo;      /*tag information*/
    Ptr      ioDTBuffer;     /*data buffer*/
    long     ioDTReqCount;   /*requested length of data*/
    long     ioDTActCount;   /*actual length of data*/
    char     ioFiller1;     /*unused*/
    char     ioIconType;    /*icon type*/
    short    ioFiller2;     /*unused*/
    long     ioDirID;       /*parent directory ID*/
    OSType   ioFileCreator; /*file creator*/
    OSType   ioFileType;   /*file type*/
    long     ioFiller3;     /*unused*/
    long     ioDTLgLen;     /*logical length of */
                                /* desktop database*/
    long     ioDTPyLen;     /*physical length of desktop */
                                /* database*/
    short    ioFiller4[14]; /*unused*/
    long     ioAPPLParID;   /*parent directory ID of */
                                /* application*/
};
typedef struct DTPBRec DTPBRec;
typedef DTPBRec *DTPBPtr;

```

Routines

Locating, Opening, and Closing the Desktop Database

```

pascal OSErr PBDTGetPath      (DTPBPtr paramBlock);
pascal OSErr PBDTOpenInform
                                (DTPBPtr paramBlock);
pascal OSErr PBDTCloseDown   (DTPBPtr paramBlock);

```

Reading the Desktop Database

```

pascal OSErr PBDTGetIcon      (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTGetIconAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetIconSync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetIconInfo
                                (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTGetIconInfoAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetIconInfoSync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetAPPL      (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTGetAPPLAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetAPPLSync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetComment
                                (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTGetCommentAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetCommentSync
                                (DTPBPtr paramBlock);

```

Adding to the Desktop Database

```

pascal OSErr PBDTAddIcon      (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTAddIconAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTAddIconSync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTAddAPPL      (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTAddAPPLAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTAddAPPLSync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTSetComment
                                (DTPBPtr paramBlock, Boolean async);

```

```
pascal OSErr PBDTSetCommentAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTSetCommentSync
                                (DTPBPtr paramBlock);
```

Deleting Entries From the Desktop Database

```
pascal OSErr PBDTRemoveAPPL
                                (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTRemoveAPPLAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTRemoveAPPLSync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTRemoveComment
                                (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTRemoveCommentAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTRemoveCommentSync
                                (DTPBPtr paramBlock);
```

Manipulating the Desktop Database Itself

```
pascal OSErr PBDTFlush          (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTFlushAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTFlushSync      (DTPBPtr paramBlock);
pascal OSErr PBDTGetInfo        (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTGetInfoAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTGetInfoSync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTReset          (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTResetAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTResetSync      (DTPBPtr paramBlock);
pascal OSErr PBDTDelete         (DTPBPtr paramBlock, Boolean async);
pascal OSErr PBDTDeleteAsync
                                (DTPBPtr paramBlock);
pascal OSErr PBDTDeleteSync
                                (DTPBPtr paramBlock);
```

Assembly-Language Summary

Data Structures

Desktop Parameter Block

12	ioCompletion	long	completion routine
16	ioResult	word	result code
18	ioNamePtr	long	pointer to file, directory, or volume name
22	ioVRefNum	word	volume reference number
24	ioDTRefNum	word	desktop database reference number
26	ioIndex	word	index into icon list; or number of files in database
28	ioTagInfo	long	tag information
32	ioDTBuffer	long	pointer to icon data
36	ioDTReqCount	long	requested size of icon data buffer
40	ioDTActCount	long	actual size of icon definition
44	ioIconType	byte	icon's type
48	ioDirID	long	parent directory
52	ioFileCreator	long	file creator
56	ioFileType	long	file type
64	ioDTLgLen	long	logical length of database files
68	ioDTPyLen	long	physical length of database files
100	ioAPPLParID	long	application's parent directory

Trap Macros

Trap Macros Requiring Routine Selectors`_HFSDispatch`

Selector	Routine
\$0020	PBDTGetPath
\$0021	PBDTCloseDown
\$0022	PBDTAddIcon
\$0023	PBDTGetIcon
\$0024	PBDTGetIconInfo
\$0025	PBDTAddAPPL
\$0026	PBDTRemoveAPPL
\$0027	PBDTGetAPPL
\$0028	PBDTSetComment
\$0029	PBDTRemoveComment
\$002A	PBDTGetComment
\$002B	PBDTFlush
\$002C	PBDTReset
\$002D	PBDTGetInfo
\$002E	PBDTOpenInform
\$002F	PBDTDelete

Result Codes

<code>noErr</code>	0	No error
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>fnfErr</code>	-43	File or directory doesn't exist
<code>wPrErr</code>	-44	Volume is locked through hardware
<code>vLckdErr</code>	-46	Volume is locked through software
<code>paramErr</code>	-50	Parameter error; use <code>PBDTGetPath</code>
<code>rfNumErr</code>	-51	Reference number invalid
<code>extFSerr</code>	-58	External file system—file system identifier is nonzero
<code>desktopDamagedErr</code>	-1305	The desktop database has become corrupted—the Finder will fix this, but if your application is not running with the Finder, use <code>PBDTReset</code> or <code>PBDTDelete</code>
<code>afpItemNotFound</code>	-5012	Information not found
<code>afpIconTypeError</code>	-5030	Sizes of new icon and one it replaces don't match

Glossary

alternate rectangle A rectangle used by the Help Manager (under some circumstances) for transposing a help balloon's tip when trying to fit the balloon onscreen. For all help resources except the 'hdlg' resource, the Help Manager moves the tip to different sides of the hot rectangle. For 'hdlg' resources, however, the Help Manager allows you to specify alternate rectangles for transposing balloon tips. You can also specify alternate rectangles when you use the `HMShowBalloon` and `HMShowMenuBalloon` functions. Compare **hot rectangle**. See also **tip**.

application translation extension A translation extension that can create a list of file types and identify files but that performs no actual file translation.

balloon definition function An implementation of a window definition function that defines the general appearance of a help balloon. See also **help balloon**.

catalog type The type of a file as maintained in a volume's hierarchical file system catalog file. See also **translation file type**.

cell A rectangular part of a list displaying information about one item from the list.

color icon record A data structure of type `CICon` used for information obtained from a color icon ('cicn') resource.

color icon resource A resource of type 'cicn' used for color icon resource data. A color icon resource can define a color icon of any size without a mask or a 32-by-32 pixel color icon with a mask. You can define the bit depth for a color icon resource and you can use resources of type 'cicn' in menus and dialog boxes. Note that the Finder does *not* use or display any resources that you create of type 'cicn'. To create an icon for display by the Finder, create one or more of the icons in an icon family. See also **icon family**, **icon resource**, **small icon resource**.

component A piece of code that provides a defined set of services to one or more clients. Applications, system extensions, as well as other components can use the services of a component.

component connection An access path to a component. A single component can serve multiple client applications at the same time by supporting multiple connections.

component identifier A value that identifies a particular component.

component instance A value that identifies a component connection. Each instance of a component can maintain separate storage and error information, and manage its A5 world.

Component Manager A collection of routines that allows your application or other clients to access components. The Component Manager manages components and also provides services to components.

component subtype A value that identifies variations on the basic interface that a component supports. As with component types, a component subtype is a sequence of four characters. The value of the component subtype is meaningful only in the context of a given component type.

component type A value that identifies the type of service provided by a component. As with resource types, a component type is a sequence of four characters.

control device function A function that interacts and communicates with the Finder, responding to requests from the Finder to handle events and perform actions. Every implementation of a control panel must contain a control device function in the control device code ('cdev') resource.

control panel A modeless dialog box that contains controls that let users specify basic settings and preferences for a systemwide feature, such as the speaker volume, desktop pattern, or picture displayed by a screen saver.

control panel file A file of type 'cdev' that contains the required and optional resources to implement a control panel. These resources also define the look of a control panel, including its icon. One of the required resources is a code resource containing a control device function.

convertor See **translator**.

current resource file The file whose resource fork the Resource Manager searches first when searching for a resource; usually the file whose resource fork was opened most recently.

data fork The fork of a file that contains the file's data.

desktop database A database of icons, file types, applications, and comments maintained by the Finder for all volumes over 2 MB.

Desktop Manager A collection of routines that manages the desktop database.

dialog-item component The portion of an 'hdlg' resource in which you specify the help messages for a particular item in a dialog or alert box.

drop-launch To drag a document's icon onto an application's icon, thereby opening the document.

dynamic window A window that can change its title or reposition any of the objects within its content area.

explicit translation The conversion of a file or scrap requiring direct intervention from an application. See also **implicit translation**.

file A named, ordered sequence of bytes stored on a Macintosh volume and divided into two forks, the data fork and the resource fork.

file reference number A number (greater than 0) returned to your application when it opens a fork of a file. Each file reference number corresponds to a unique access path.

file translation list A list of source and destination file types among which a file translation system can translate. Defined by the FileTranslationList data type.

file translation system A translation system that can recognize and translate files from one format to another.

file type specification A way of specifying the catalog type and translation file type of a file, as well as other information about translating the file. Defined by the FileTypeSpec data type.

filter See **translator**.

header component The portion of a help resource in which you supply information that applies to all help balloons specified in the resource—information such as the version number of the Help Manager, the balloon definition function, and the variation code.

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 the 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 the balloon's hot rectangle. See also **alternate rectangle**, **hot rectangle**.

Help Manager A collection of routines that your application can use to provide Balloon Help assistance to your application's users.

help messages Descriptive text or pictures that appear inside help balloons.

help resources Application-supplied resources that describe help messages, balloon definition functions, variation codes, and, when necessary, the tips and the hot rectangles or alternate rectangles for the Help Manager to use in drawing help balloons. These help resources are the menu help ('hmnu') resource, the dialog item help ('hdlg') resource, the rectangle help ('hret') resource, the window help ('hwin') resource, the Finder icon help ('hfdr') resource, and the default help override ('hovr') resource.

hot rectangle An area defined to display a help balloon. When the user moves the cursor over this area, the Help Manager displays the help balloon associated with the hot rectangle. Compare **alternate rectangle**.

hot-rectangle component The portion of an 'hrct' resource in which you specify hot rectangles and the help messages associated with each hot rectangle.

icon An image on a Macintosh screen that graphically represents some object, such as a file, a folder, or the Trash. See also **icon family**.

icon cache An icon suite that includes a pointer to an icon getter function and a pointer to data that can be used as a reference constant. See also **icon getter function**, **icon suite**.

icon component The portion of an 'hfdrr' resource in which you specify a help message for your application's Finder icon.

icon family A set of icons that represent a single object and share the same resource ID. The resource types and names of each member of an icon family are 'ICN#'—a large (32-by-32 pixel) black-and-white icon and mask; 'ics#'—a small (16-by-16 pixel) black-and-white icon and mask; 'icl4'—a large (32-by-32 pixel) color icon with 4 bits of color data per pixel; 'ics4'—a small (16-by-16 pixel) color icon with 4 bits of color data per pixel; 'icl8'—a large (32-by-32 pixel) color icon with 8 bits of color data per pixel; and 'ics8'—a small (16-by-16 pixel) color icon with 8 bits of color data per pixel.

icon getter function An application-defined function that returns a handle to icon data for a specified icon type. You can associate an icon getter function with an icon cache. Subsequent calls to Icon Utilities routines that use icons not present in the icon cache use the icon getter function to read the icon data into memory.

icon resource A resource of type 'ICON' that contains a bitmap for a 32-by-32 pixel black-and-white icon. You can use resources of type 'ICON' in menus and dialog boxes. Note that the Finder does *not* use or display any resources of type 'ICON' that you create. To create an icon for display by the Finder, create

one or more of the icons in an icon family. See also **color icon resource**, **icon family**, **small icon resource**.

icon suite One or more handles to icon data that represents icons from a single icon family. Some Icon Utilities routines accept a handle to an icon suite and draw the appropriate icon from that suite for the destination rectangle and the bit depth of the display device.

Icon Utilities A collection of routines that your application can use to display icons in graphics ports (such as windows or dialog boxes) created by your application.

implicit translation The automatic conversion of a file or scrap without direct intervention from an application. See also **explicit translation**.

kind resource A resource that contains kind strings for document types. Defined by the 'kind' resource type.

kind string The string displayed in the "Kind" column in a Finder window when a folder's contents are viewed by name, size, kind, label, or date (that is, by any method other than by icon or small icon).

list A series of items displayed within a rectangle. Lists may have zero, one, or two scroll bars.

list definition procedure A code resource of type 'LDEF' that defines the appearance of a list.

List Manager A collection of routines that your application can use to create and display lists in your application's windows or dialog boxes.

Macintosh Easy Open The part of the Macintosh system software that provides translation services for users of Macintosh computers. Macintosh Easy Open uses the Translation Manager to provide these services.

menu-item component The portion of an 'hmenu' resource in which you specify the help messages for a particular menu item.

menu-title component The portion of an 'hmenu' resource in which you specify help messages for the menu title.

mini icons Icons of resource types `'icm#'`, `'icm4'`, and `'icm8'` that measure 12 by 16 pixels. Like the icons in an icon family, the three resource types for mini icons identify the icon list, 4-bit color icons, and 8-bit color icons, respectively. Compare **small icon resource**.

missing-items component The portion of a help resource in which you specify help messages for any items missing from or unspecified in the rest of the resource.

monitors extension An extension to the Monitors control panel that a video card manufacturer can develop and provide to give users a simple way to control features of the video card. A monitors extension is limited to the video card; it cannot be used to control the settings of systemwide features. A user can open an extension only through the Monitors control panel.

monitors extension file A file of type `'cdev'` that contains required and optional resources that implement an extension to the Monitors control panel for a specific video card. One of the required resources is a code resource containing a monitors extension function.

monitors extension function A function that interacts and communicates with the Monitors control panel, responding to requests from the Monitors control panel to handle events and perform actions. Every implementation of an extension to the Monitors control panel must contain a monitors extension function in the monitors code (`'mnr'`) resource.

mouse location The location of the cursor at the time an event occurs.

open resource A resource of type `'open'` that declares which file types your application can open as documents.

point-to-point translation A translation group with one source type and one destination type.

resource Data of any kind stored in a defined format in a file's resource fork and managed by the Resource Manager.

resource attributes Flags in the resource map that tell the Resource Manager how to handle the corresponding resource.

resource file Synonym for resource fork.

resource fork The fork of a file that contains the file's resources.

resource ID An integer that identifies a specific resource of a given type.

resource map Data read into memory when the Resource Manager opens a resource fork. A resource map contains information about the resources in the resource fork.

resource type The type of a resource in a resource fork, designated by a sequence of four characters (such as `'MENU'` for a menu).

scrap A storage area (either in memory or on disk) that is available to applications to hold the last data cut or copied by the user.

Scrap Manager A collection of routines that your application can use to support copy-and-paste operations.

scrap translation list A list of source and destination scrap types among which a scrap translation system can translate. Defined by the `ScrapTranslationList` data type.

scrap translation system A translation system that can recognize and translate scraps from one format to another.

scrap type specification A way of specifying information about translating a scrap. Defined by the `ScrapTypeSpec` data type.

small icon resource A resource of type `'SICN'` that describes 12-by-16 pixel icons, even though the icons are stored in the resource as 16-by-16 pixel bitmaps. An `'SICN'` resource consists of a list of 16-by-16 pixel bitmaps for black-and-white icons; by convention, the list includes only two bitmaps, and the second bitmap is considered a mask. You can use resources of type `'SICN'` in menus. Note that the Finder does *not* use or display any resources that you create of type `'SICN'`. To create an icon for display by the Finder, create one or more of the icons in an icon family. See also **color icon resource**, **icon family**, **icon resource**. Compare **mini icons**.

static window A window that doesn't change its title or reposition any of the objects within its content area.

superuser A user who is considered to be very knowledgeable. A monitors extension can define controls that it displays for superusers only.

tip At the side of a help balloon, the point that indicates what object or area is explained in the help balloon. See also **help balloon**, **variation code**.

transform A mode you can specify with some Icon Utilities routines that draw icons. Specifying transforms with these routines alters the appearance of the icons in standard ways that are analogous to Finder states for icons. For example, you can specify the transform `ttSelected` to draw an icon so that it is highlighted as if it were selected in the Finder.

translation extension A component called by Macintosh Easy Open to identify and translate files or scraps. See also **application translation extension**.

translation file type The type of a file relevant for translation purposes. See also **catalog type**.

translation group A collection of source and destination file types; within each translation group, each source file type can be translated into any destination file type.

Translation Manager A collection of routines that provide data conversion services (such as implicit translation) for applications on Macintosh computers. You can use the Translation Manager to implement explicit translation.

translation system A translation extension, with or without external translators, that is able to recognize and translate files or scraps.

translator A piece of software called by translation extensions or by applications to convert documents or scraps from one format to another.

type selection A feature that allows a user to type the name of an item in a list to select it.

variation code In the header component of a help resource, an integer that specifies the preferred position of a help balloon relative to its hot rectangle. The balloon definition function

draws the frame of the help balloon based on its variation code. See also **balloon definition function**.

window component The portion of an 'hwin' resource in which you associate an 'hrct' or 'hdlg' resource to a particular window.

Index

Numerals

- 4-bit color icons 5-4
- 8-bit color icons 5-4, 5-5
- 12-by-16 pixel icons (mini) 5-7
- 16-by-16 pixel icons (small) 5-4, 5-6
- 32-by-32 pixel icons (large) 5-4

A

- About This Macintosh dialog box 4-8
- action functions 5-57 to 5-58
- activate events, in lists 4-34
- AddIconToSuite function 5-33
- AddResource procedure 1-90 to 1-91
- A5 register, and code resources 4-98
- A5 world, and component connections 6-68 to 6-69
- alert boxes
 - help balloons for 3-51 to 3-74
 - help balloons for areas outside 3-87 to 3-89
 - and Help menu 3-92
 - and help messages for menus 3-38
- aligning icons 5-36
- alternate rectangles. *See also* hot rectangles
 - specifying in 'hdlg' resources 3-55 to 3-57
 - specifying in HMShowBalloon function 3-79 to 3-81
- anchor algorithm for extending list selections 4-18 to 4-20
- Apple menu, help balloons for 3-13 to 3-18
- Apple Menu Items folder, icon for 1-132
- application-defined routines. *See also* sample routines
 - DoGetFileTranslationList 7-54
 - DoGetScrapTranslationList 7-59
 - DoIdentifyFile 7-56
 - DoIdentifyScrap 7-60
 - DoTranslateFile 7-57
 - DoTranslateScrap 7-61
 - MyBalloonDef 3-129 to 3-130
 - MyCdev 8-74 to 8-78
 - MyClickLoop 4-101
 - MyComponent 6-79
 - MyIconAction 5-58
 - MyIconGetter 5-59
 - MyLDEF 4-97 to 4-99
 - MyMatchFunction 4-99 to 4-100
 - MyMntrExt 8-78 to 8-82
 - MyTip 3-130 to 3-131

- application icons, help balloons for 3-84 to 3-86
- Application menu, help balloons for 3-13 to 3-16
- applications
 - adding to the desktop database 9-18
 - default icon for 1-130
 - removing from desktop database 9-21
- application startup 1-50
- application translation extensions 7-14, 7-35
- arrow keys
 - extending list selections with 4-16 to 4-20
 - moving list selections with 4-15 to 4-16
 - processing for the current list 4-52
 - supporting navigation of lists with 4-48 to 4-53

B

- balloon definition functions
 - creating 3-93 to 3-94
 - standard 3-8
- Balloon Help assistance. *See also* help balloons
 - defined 3-5
 - determining whether enabled 3-98
 - enabling and disabling 3-5, 3-7, 3-107 to 3-108
 - user interface guidelines 3-18 to 3-23, 3-37 to 3-38, 3-39 to 3-40, 3-57 to 3-58, 3-70 to 3-71
- BalloonWriter tool 3-17
- batch translation 7-9
- BeginUpdate procedure, and updating lists 4-34
- 'BNDL' resource type
 - and control panel files 8-7, 8-22, 8-83
 - and monitors extensions files 8-12, 8-57, 8-59, 8-60
 - used in desktop database 9-3
- bundle bit 7-23
- bundles resources. *See* 'BNDL' resource type

C

- CallComponentFunction function 6-63 to 6-64
- CallComponentFunctionWithStorage function 6-64 to 6-65
- CanDocBeOpened function 7-17, 7-40 to 7-42
- can do request 6-22
- CaptureComponent function 6-25, 6-75 to 6-76
- 'card' resource type 8-11, 8-50 to 8-51, 8-87
- catalog types 7-19

I N D E X

- 'cdev' file type 8-4, 8-48
- 'cdev' resource type 8-7, 8-25 to 8-48, 8-76 to 8-77
- cdev functions. *See* control device functions
- Cell data type 4-22, 4-65
- cell record 4-65
- cells. *See* list cells
- ChangedResource procedure 1-21, 1-88 to 1-90
- 'cicn' resource type. *See also* icon resources
 - and color icon record 5-17
 - and Dialog Manager 5-6
 - drawing 5-13 to 5-17
 - and Menu Manager 5-6
- CIcon data type 5-17
- Clear command (Edit menu) 2-6
- click-loop procedures 4-25, 4-100 to 4-101
- Clipboard file 2-33
- Clipboard window 2-10
 - hiding 2-20
 - showing 2-25
- cloning, components 6-35
- close boxes, help balloon for 3-14 to 3-16, 3-87 to 3-89
- CloseComponent function 6-47
- CloseComponentResFile function 6-73
- close request 6-21
- CloseResFile procedure 1-110 to 1-111
- color icon record 5-17
- color icon resources. *See* 'cicn' resource type
- color icons 5-4 to 5-6
- Command key, using to create discontinuous selections in lists 4-11
- Command-key equivalents. *See* keyboard equivalents
- component connections 6-6, 6-65 to 6-69
- Component data type 6-41
- ComponentDescription data type 6-37 to 6-40, 6-52 to 6-54
- component description record 6-37 to 6-40, 6-52 to 6-54
- component file 6-32 to 6-33, 6-71 to 6-73, 6-84
- ComponentFunctionImplemented function 6-50 to 6-51
- component identifiers 6-9, 6-40 to 6-41, 6-42 to 6-43, 6-46
- ComponentInstance data type 6-41
- component instances 6-6, 6-40 to 6-41, 6-45 to 6-46
- Component Manager 6-3 to 6-99
 - data structures in
 - for applications 6-37 to 6-40
 - for components 6-52 to 6-55
 - requests to components 6-18 to 6-28
 - resources in 6-80 to 6-85
 - routines in
 - for applications 6-41 to 6-52
 - for components 6-56 to 6-76
 - testing for availability 6-6
- ComponentParameters data type 6-55
- component parameters record 6-54 to 6-55
- component requests 6-18 to 6-28
 - can do 6-22
 - close 6-21
 - open 6-19 to 6-20
 - register 6-23 to 6-24
 - target 6-25 to 6-26
 - unregister 6-24 to 6-25
 - version 6-22 to 6-23
- ComponentResource data type 6-81 to 6-85
- component resources 6-32 to 6-33, 6-80 to 6-85
- components
 - calling 6-73 to 6-74
 - capturing 6-25 to 6-26, 6-75 to 6-76
 - cloning 6-35
 - closing connections to 6-12, 6-47
 - defined 6-3
 - finding 6-8 to 6-9, 6-42 to 6-44
 - getting information about 6-10 to 6-11, 6-47 to 6-51
 - hiding 6-75 to 6-76
 - interfaces of, defining 6-28 to 6-30
 - levels of service 6-3, 6-35, 6-73
 - manufacturer code for 6-4, 6-39, 6-53
 - opening connections to 6-7 to 6-10, 6-44 to 6-46
 - registering 6-30 to 6-32, 6-57 to 6-62, 6-80 to 6-81
 - requesting services from 6-18 to 6-27
 - structure of 6-13 to 6-18
 - targeting 6-25 to 6-26
 - unregistering 6-24 to 6-25
 - using services of 6-11 to 6-12
- ComponentSetTarget function 6-25, 6-77
- component subtypes 6-4, 6-38, 6-53
- component types 6-4, 6-38, 6-53
- content region of help balloons 3-8, 3-93
- control device code resources 8-7, 8-25, 8-74
- control device functions 8-4, 8-25 to 8-47, 8-74 to 8-78
 - and activate events 8-33
 - and keyboard-related events 8-37
 - and mouse-related events 8-39
 - performing initialization 8-29
 - preserving a handle to private storage 8-30
 - and update events 8-43
- Control Panel desk accessory 8-15, 8-23, 8-37, 8-39
- control panel files
 - creating 8-12 to 8-48
 - defined 8-4
 - and monitors extensions 8-73
 - and system extensions 8-8
 - user documentation for 8-8
- control panels 8-3 to 8-92
 - and Command-key equivalents 8-37
 - compatibility with the Control Panel desk accessory 8-15, 8-23, 8-37, 8-39
 - creating 8-12 to 8-48
 - creating resources for 8-14 to 8-24
 - and error reporting 8-47

- initializing 8-30
- required resources 8-6, 8-82 to 8-87
- shutting down 8-45
- specifying a font for 8-23
- and text defined as user items 8-43
- user interface guidelines for 8-12
- valid resource IDs for 8-14
- where to store 8-8
- Control Panels folder, icon for 1-132
- controls, help balloons for 3-55
- conversion of file formats. *See* translation of file formats
- copy and paste, user interface guidelines for 2-10 to 2-11
- Copy command (Edit menu) 2-6, 2-19
- CountComponentInstances function 6-67 to 6-68
- CountComponents function 6-43 to 6-44
- CountlResources function 1-98 to 1-99
- CountlTypes function 1-102
- CountResources function 1-98
- CountTypes function 1-102
- CreateResFile procedure 1-57 to 1-58
- current resource file
 - defined 1-10
 - getting and setting 1-28 to 1-30, 1-68 to 1-71
- CurResFile function 1-68 to 1-69
- cursors, tracking location by Help Manager 3-25
- cut and paste, intelligent 2-10 to 2-11
- Cut command (Edit menu) 2-6, 2-15 to 2-19

D

- DataArray data type 4-25
- data forks 1-4 to 1-6
- DataHandle data type 4-25, 4-66
- 'dctb' resource type, and Standard File Package dialog boxes 7-11
- default help override resources. *See* 'hovr' resource type
- DelegateComponentCall function 6-35, 6-36, 6-74
- desk accessories, default icon for 1-130
- desktop, default icons used on 1-133
- desktop database 9-3 to 9-26
 - adding data to 9-17 to 9-20
 - closing 9-11
 - contents of 9-5
 - deleting data from 9-20 to 9-22
 - determining parent directory of 9-24
 - determining reference number of 9-5, 9-9 to 9-11
 - determining space used by 9-24
 - locating 9-9 to 9-11
 - opening 9-11 to 9-12
 - removing data from 9-25 to 9-26

- retrieving data from 9-12 to 9-16
- saving to disk 9-23
- Desktop file 9-3, 9-4
- Desktop Manager 9-3 to 9-26
 - data structures in 9-6 to 9-8
 - routines in 9-8 to 9-26
- desktop parameter block 9-7 to 9-8
- DetachResource procedure 1-22 to 1-24, 1-108
- dialog boxes
 - creating lists in 4-29
 - handling editing operations in 2-31
 - help balloons for 3-51 to 3-74
 - help balloons for areas outside 3-87 to 3-89
 - and Help menu 3-92
 - and help messages for menus 3-38, 3-47 to 3-51
- DialogCopy procedure 2-31
- DialogCut procedure 2-31
- dialog-item help resources. *See* 'hdlg' resource type
- Dialog Manager
 - and Scrap Manager 2-31
 - and TextEdit 2-31
- DialogPaste procedure 2-31
- discontiguous selections, in lists 4-11
- DisposeCIcon procedure 5-30
- DisposeIconSuite function 5-42 to 5-43
- 'DITL' resource type
 - for a control panel 8-6, 8-17 to 8-19
 - help items in 3-51 to 3-52, 3-59 to 3-63
 - for a monitors extension 8-11, 8-50, 8-54 to 8-55
- DocOpenMethod data type 7-41
- Document Converter 7-9 to 7-10, 7-47
- document opening methods 7-17, 7-41
- documents
 - batch translation of 7-9
 - default icon for 1-130
 - drop launching 7-7
 - help balloons for icons 3-84 to 3-86
 - identifying the type of 7-32 to 7-33
 - opening from the Finder 7-5 to 7-7
 - opening in an application 7-8 to 7-9
 - opening with explicit translation 7-17 to 7-18
 - translating. *See* translation of file formats 7-33 to 7-35
 - translating on the desktop 7-9 to 7-10
 - of type 'TEXT' 7-11
- Down Arrow key 4-48
- DrawGrowIcon procedure, using to create resizable lists 4-28
- DrawText procedure, using to draw text in a cell of a list 4-24
- drop launching 7-7, 7-13
- DTPBRec data type 9-7 to 9-8
- dynamic file-type lists. *See* application translation extensions
- dynamic windows, help balloons for 3-74 to 3-84

E

Edition Manager, and Macintosh Easy Open 7-4, 7-10
 editions, translating format of 7-10
 Edit menu
 Clear command 2-6
 and control panels 8-5, 8-46
 Copy command 2-6
 Cut command 2-6
 Paste command 2-6
 Show Clipboard/Hide Clipboard command 2-10
 8-bit color icons 5-4, 5-5
 EndUpdate procedure, and updating lists 4-34
 events
 in control panels 8-25 to 8-26
 handling resume events 2-25 to 2-26
 handling suspend events 2-19 to 2-20
 in lists 4-32 to 4-34
 explicit translation 7-17
 extend algorithm for extending list selections 4-16 to 4-17
 ExtendFileTypeList function 7-17, 7-38 to 7-39
 extensions, default icon for 1-130
 Extensions folder, icon for 1-132

F

file filter functions. *See* Standard File Package, file filter functions
 file reference numbers
 defined 1-10
 for System file's resource fork 1-50
 used with Resource Manager routines 1-24
 file reference resources. *See* 'FREF' resource type
 files
 data fork of 1-4 to 1-6
 resource fork of 1-4 to 1-6. *See also* applications; documents
 user comments associated with 9-3
 file system specification (FSSpec) records, and Resource Manager routines 1-13
 FileTranslationList data type 7-28, 7-48 to 7-49
 file translation lists
 creating 7-28 to 7-32
 defined 7-27
 structure of 7-48 to 7-49
 FileTranslationSpec data type 7-41
 file translation systems 7-5
 FileType data type 7-18
 FileTypePtr data type 7-39
 file types
 declaring those an application can open 7-13 to 7-14, 7-44

 finding those supported by applications 9-14
 identifying 7-32 to 7-33
 and Macintosh Easy Open 7-19
 FileTypeSpec data type 7-30, 7-46 to 7-47
 file type specifications 7-30, 7-46 to 7-47
 filters for translating file formats. *See* translation systems
 Finder
 and control panels 8-7
 database for a volume 9-3 to 9-26
 and Macintosh Easy Open 7-4, 7-5 to 7-7
 Finder icon help resources. *See* 'hfdrr' resource type
 Finder icons. *See* standard icons
 FindNextComponent function 6-8, 6-42 to 6-43
 'finf' resource type 8-7, 8-23, 8-86
 'FKEY' resource type 1-129
 folders
 default icons for 1-131
 icons for those in the System Folder 1-132
 font information resource 8-7, 8-23, 8-86
 fonts
 in control panels 8-7, 8-23, 8-86
 in help balloons 3-110 to 3-113
 Fonts folder, icon for 1-132
 ForEachIconDo function 5-38 to 5-39
 4-bit color icons 5-4
 'FREF' resource type
 and control panel files 8-7, 8-22, 8-82
 and monitors extensions files 8-12, 8-57, 8-59
 FSpCreateResFile procedure 1-25 to 1-26, 1-53 to 1-55
 FSpOpenResFile function 1-26 to 1-28, 1-58 to 1-62
 FSSpec data type 1-13, 1-25, 1-54, 1-59
 function key resource IDs 1-129

G

'gama' resource type, and monitor extensions 8-12, 8-59
 gamma tables, and monitor extensions 8-59
 Gestalt function
 and Help Manager 3-18
 and Icon Utilities 5-8
 and Resource Manager 1-13 to 1-14
 and Scrap Manager 2-14
 and Translation Manager 7-12
 GetCIcon function 5-29
 GetComponentIconSuite function 6-49 to 6-50
 GetComponentInfo function 6-10 to 6-11, 6-48 to 6-49
 GetComponentInstanceA5 function 6-68 to 6-69
 GetComponentInstanceError function 6-51 to 6-52
 GetComponentInstanceStorage function 6-67
 GetComponentListModSeed function 6-44

GetComponentRefcon function 6-35, 6-71
 GetComponentVersion function 6-50
 GetControlValue function 8-40
 GetDialogItem procedure 8-30, 8-72
 GetFileTypesThatAppCanNativelyOpen
 function 7-17, 7-37 to 7-38
 GetIconCacheData function 5-55
 GetIconCacheProc function 5-56
 GetIconFromSuite function 5-13, 5-34 to 5-35
 GetIcon function 5-14, 5-28 to 5-29
 GetIconSuite function 5-11, 5-13, 5-31 to 5-32
 GetIndResource function 1-99 to 1-100
 GetIndType procedure 1-103
 GetLabel function 5-41 to 5-42
 GetMaxResourceSize function 1-105 to 1-106
 GetNamedResource function 1-75 to 1-76
 Get1IndResource function 1-100 to 1-101
 Get1IndType procedure 1-104
 Get1NamedResource function 1-77 to 1-78
 Get1Resource function 1-74 to 1-75
 GetResAttrs function 1-84 to 1-85
 GetResFileAttrs function 1-116 to 1-118
 GetResInfo procedure 1-81 to 1-82
 GetResource function 1-18, 1-73 to 1-74
 GetResourceSizeOnDisk function 1-105
 GetScrap function 2-38 to 2-40, 7-11
 GetSuiteLabel function 5-40

H

handles

NIL, returned by Resource Manager routines 1-14, 1-51
 type-casting, for use by Resource Manager 1-33, 1-39
 HCreateResFile procedure 1-56 to 1-57
 'hdlg' resource type
 compared with 'hrct' resource type 3-57, 3-65, 3-70
 example of 3-59 to 3-61, 3-72 to 3-74
 options for 3-54
 Rez input format for 3-53
 Rez output format for 3-140 to 3-148
 used with 'hwin' resources 3-68, 3-69 to 3-70

help balloons

for alert boxes 3-51 to 3-74
 for close boxes 3-14 to 3-16, 3-87 to 3-89
 content region of 3-93
 for controls 3-55
 default 3-13 to 3-17, 3-84 to 3-89
 defined 3-5
 determining whether enabled 3-98
 determining whether showing 3-99
 for dialog boxes 3-51 to 3-74
 displaying 3-8 to 3-13, 3-100 to 3-103

enabling and disabling 3-5, 3-7, 3-107 to 3-108
 help messages in
 extracting 3-122 to 3-128
 font name and size for 3-110 to 3-113
 formats 3-23 to 3-24
 specifying in a help message record 3-75 to 3-80
 user interface guidelines 3-18 to 3-23, 3-37 to 3-38, 3-39 to 3-40, 3-57 to 3-58, 3-70 to 3-71
 for icons 3-14, 3-17, 3-84 to 3-86
 for menus 3-14 to 3-17, 3-27 to 3-51, 3-103 to 3-105
 methods for displaying help balloons 3-8 to 3-13, 3-99 to 3-105
 overriding default
 for application icons 3-84 to 3-86
 for inactive windows 3-87 to 3-89
 for window frames 3-87
 positioning 3-9 to 3-11, 3-23
 removing 3-75, 3-81, 3-105, 3-106
 routines for 3-97 to 3-128
 sizes of, getting 3-119 to 3-122
 structure region of 3-93
 tips of. *See* tips of help balloons
 user interface guidelines 3-18 to 3-23, 3-37 to 3-38, 3-39 to 3-40, 3-57 to 3-58, 3-70 to 3-71
 variation codes for 3-9 to 3-11
 for windows 3-14 to 3-16, 3-63 to 3-84, 3-87 to 3-89
 for zoom boxes 3-14 to 3-16, 3-87 to 3-89

HelpItem item type 3-62 to 3-63

Help Manager. *See also* help balloons 3-5 to 3-177

application-defined routines for 3-128 to 3-131
 data structures in 3-95 to 3-97
 resources in 3-132 to 3-165
 routines in 3-97 to 3-128
 testing for availability 3-18

Help Manager string list record 3-78 to 3-79, 3-97

Help menu

adding items to 3-90 to 3-93
 help balloons for 3-13 to 3-16
 place in menu bar 3-18

Show/Hide Balloons command 3-7

help message record 3-76 to 3-80, 3-95 to 3-97

help messages. *See* help balloons

help resources

getting and setting 3-114 to 3-119
 options for 3-25 to 3-27. *See also* 'hdlg' resource type; 'hfdr' resource type; 'hmnu' resource type; 'hovr' resource type; 'hrct' resource type; 'hwin' resource type

'hfdr' resource type

and control panel files 8-23

example of 3-86

options for 3-85

Rez input format for 3-84

Rez output format for 3-156 to 3-160

Hide/Show Balloons command (Help menu) 3-7

Hide/Show Clipboard command (Edit menu) 2-10

HMBalloonPict function 3-120 to 3-121

HMBalloonRect function 3-119 to 3-120

HMCompareItem identifier 3-124, 3-127

HMExtractHelpMsg function 3-122 to 3-125

HMGetBalloons function 3-75, 3-98

HMGetBalloonWindow function 3-121 to 3-122

HMGetDialogResID function 3-118 to 3-119

HMGetFont function 3-110

HMGetFontSize function 3-111

HMGetHelpMenuHandle function 3-90 to 3-91, 3-108 to 3-109

HMGetIndHelpMsg function 3-125 to 3-128

HMGetMenuResID function 3-115 to 3-116

HMIIsBalloon function 3-99

HMMessageRecord data type 3-76 to 3-80, 3-95 to 3-97

'hmnu' resource type

- example of 3-30 to 3-32, 3-41 to 3-42, 3-44 to 3-45, 3-46, 3-49 to 3-50
- options for 3-32
- Rez input format for 3-30, 3-43 to 3-44, 3-45 to 3-46
- Rez output format for 3-132 to 3-140

HMRemoveBalloon function 3-75, 3-81, 3-106

HMScanTemplateItems function 3-116 to 3-117

HMSetBalloons function 3-107 to 3-108

HMSetDialogResID function 3-117 to 3-118

HMSetFont function 3-112

HMSetFontSize function 3-113

HMSetMenuResID function 3-50, 3-114 to 3-115

HMShowBalloon function 3-74 to 3-84, 3-100 to 3-103

HMShowMenuBalloon function 3-103 to 3-105

HMStringResType data type 3-78 to 3-79, 3-97

HomeResFile function 1-71

HOpenResFile function 1-62 to 1-64

hot rectangles. *See also* alternate rectangles

- defined 3-9
- in dialog and alert boxes 3-56 to 3-57
- in menus 3-29
- specifying in dynamic windows 3-74, 3-80
- specifying in static windows 3-66

'hovr' resource type 3-160 to 3-165

- example of 3-89
- options for 3-88
- Rez input format for 3-87
- Rez output format for 3-160 to 3-165

'hrct' resource type

- compared with 'hdlg' resource type 3-57, 3-65, 3-70
- example of 3-71 to 3-72
- options for 3-67
- Rez input format for 3-66
- Rez output format for 3-148 to 3-153
- used with 'hwin' resources 3-68, 3-69 to 3-70

'hwin' resource type

- compared with using HelpItem item 3-63
- example of 3-71 to 3-72, 3-72 to 3-74

- options for 3-69
- Rez input format for 3-68
- Rez output format for 3-154 to 3-156
- used with 'hdlg' and 'hrct' resources 3-68, 3-69 to 3-74

I, J

'icl4' resource type, as part of an icon family 5-4

'icl8' resource type

- as part of an icon family 5-5

'icm#' resource type 5-7

'icm4' resource type 5-7

'icm8' resource type 5-7

'ICN#' resource type, as part of an icon family 5-4

icon caches

- defined 5-9
- working with 5-53 to 5-57

icon families

- defined 5-4 to 5-5
- drawing an icon from a resource 5-10
- drawing icons in 5-8 to 5-13
- drawing specific icons from 5-12 to 5-13

icon getter function 5-9, 5-58 to 5-59

IconIDToRgn function 5-44 to 5-45

icon list resources, as part of an icon family 5-4

icon masks 5-4

- converting to a region 5-43 to 5-46

IconMethodToRgn function 5-45 to 5-46

icon resources

- 'cicn' 5-6
- 'icl4' 5-4
- 'icl8' 5-4
- 'icm#' 5-7
- 'icm4' 5-7
- 'icm8' 5-7
- 'ICN#' 5-4
- 'ICON' 5-6
- 'ics#' 5-4
- 'ics4' 5-4
- 'ics8' 5-4
- 'SICN' 5-6

'ICON' resource type

- and Dialog Manager 5-6
- drawing 5-13 to 5-17
- and Menu Manager 5-6

icons

- alignment of 5-8, 5-36
- black and white 5-4
- color 5-4
- for components 6-48, 6-59, 6-81
- for control panel files 8-14
- default 1-129 to 1-134

defined 5-3
 designing and creating 5-4
 in desktop database 9-26
 adding to 9-17 to 9-18
 getting icon definitions from 9-12 to 9-13
 removing from 9-25
 in dialog boxes 5-3
 disposing of 5-30
 drawing
 members of an icon family 5-10
 from an icon suite 5-11
 from resources 5-19 to 5-28
 those that are not part of an icon family 5-13 to 5-17, 5-28 to 5-29
 8-bit color 5-4, 5-5
 4-bit color 5-4
 help balloons for 3-13, 3-18, 3-84 to 3-86
 hit testing 5-46 to 5-53
 in icon suite, constants for specifying 5-31
 label information, getting 5-40, 5-41
 large (32-by-32 pixel) 5-4
 manipulating icon data 5-13
 in menus 5-3
 mini 5-7. *See also* icon families; icon resources
 for monitors extension files 8-50
 small (16-by-16 pixel) 5-4, 5-6
 in the System file 1-129 to 1-134
 transform constants 5-37
 icon suites
 creating 5-30 to 5-33
 defined 5-9
 disposing of 5-42 to 5-43
 drawing icons from 5-11, 5-35 to 5-38
 getting icons from 5-34 to 5-35
 specifying icons in, constants for 5-31
 IconSuiteToRgn function 5-43 to 5-44
 Icon Utilities 5-3 to 5-73
 application-defined routines for 5-57 to 5-59
 data structure in 5-17 to 5-18
 routines in 5-18 to 5-57
 testing for availability 5-7
 'ics#' resource type, as part of an icon family 5-4
 'ics4' resource type, as part of an icon family 5-5
 'ics8' resource type, as part of an icon family 5-5
 'ictb' resource type
 and control panels 8-23
 and monitors extensions 8-56
 implicit translation
 defined 7-4
 of editions 7-10
 while opening documents 7-6, 7-8
 while pasting data 2-21, 2-39, 7-10
 inactive windows, help balloons for 3-87 to 3-89
 InfoScrap function 2-34 to 2-35
 InitResources function 1-50

'INIT' resource type, and monitors extensions 8-12, 8-61
 InvalRect procedure, using to stimulate redrawing of a list 4-32
 item color table resources
 and control panels 8-23
 and monitors extensions 8-56
 item list resources. *See* 'DITL' resource type 8-50
 IUMagIDString function, used by LSearch
 function 4-43
 IUMagString function, using to customize a search of a list 4-43

K

Keyboard control panel, and type selection 4-45
 keyboard equivalents, and control panels 8-37
 keyboard events, handling in control panels 8-37 to 8-39
 keyboards, using to navigate lists 4-15 to 4-20, 4-45 to 4-53
 key-down events, scrolling lists in response to 4-45 to 4-53
 'kind' resource type 7-11, 7-14 to 7-15, 7-45 to 7-46
 kind strings 7-14 to 7-15

L

labels, icon 5-22, 5-37, 5-41
 LActivate procedure 4-85 to 4-86
 LAddColumn function 4-73 to 4-74
 LAddRow function 4-74 to 4-75
 LAddToCell procedure 4-80 to 4-81
 large 4-bit color icon resources, as part of an icon family 5-4
 large 8-bit color icon resources, as part of an icon family 5-5
 large (32-by-32 pixel) icons 5-4
 LAutoScroll procedure 4-88 to 4-89
 LCellSize procedure 4-92 to 4-93
 LClick function 4-25, 4-33, 4-84 to 4-85
 LCloseMsg message 4-62 to 4-63
 LClrCell procedure 4-40, 4-81
 'LDEF' resource type 4-7, 4-58, 4-98
 LDelColumn procedure 4-75 to 4-76
 LDelRow procedure 4-76 to 4-77
 LDispose procedure 4-72 to 4-73
 LDrawMsg message 4-60 to 4-62
 LDraw procedure 4-88
 Left Arrow key 4-48
 LGetCellDataLocation procedure 4-82 to 4-83

LGetCell procedure 4-41, 4-83
 LGetSelect function 4-35, 4-77 to 4-78
 LHiliteMsg message 4-62
 lInitMsg message 4-60
 list cells
 condensed text in 4-5
 containing several types of information 4-8
 customizing selection algorithm 4-38 to 4-39
 defined 4-4
 selection of 4-9 to 4-15, 4-34 to 4-39
 size of 4-4, 4-23, 4-28
 list definition procedure resources 4-7, 4-58, 4-98
 list definition procedures 4-58 to 4-64, 4-96 to 4-99
 changing fields of the list record 4-98
 compiling 4-98
 entry point of 4-99
 processing messages in 4-59
 responding to lCloseMsg message 4-62 to 4-63
 responding to lDrawMsg message 4-60 to 4-62
 responding to LHiliteMsg message 4-62
 responding to lInitMsg message 4-60
 using global variables in 4-98
 using to create graphical lists 4-7
 ListHandle data type 4-23
 List Manager 4-3 to 4-111
 application-defined routines for 4-96 to 4-101
 data structures in 4-65 to 4-69
 routines in 4-70 to 4-96
 ListRec data type 4-22 to 4-25, 4-66 to 4-69
 list record 4-22 to 4-25, 4-66 to 4-69
 setting selFlags field 4-39
 lists 4-4. *See also* list cells
 activating 4-34, 4-85 to 4-86
 adding items alphabetically 4-41 to 4-42
 appearance of 4-4 to 4-8
 arrow-key navigation in 4-48 to 4-53
 automatic drawing mode 4-32, 4-87 to 4-88
 borders around, drawing 4-30
 cell data, accessing 4-25, 4-79 to 4-83
 creating 4-27 to 4-30, 4-70 to 4-72
 creating a list of pictures 4-63 to 4-64
 data bounds of 4-28
 discontiguous selections in 4-11
 disposing of 4-30, 4-72 to 4-73
 double click in 4-33
 events in, responding to 4-32 to 4-34, 4-84 to 4-86
 graphical items in 4-7, 4-63 to 4-64
 introduced 4-4 to 4-8
 keyboard navigation of 4-15 to 4-20, 4-45 to 4-53
 location of last click, determining 4-24
 multiple in a window 4-20 to 4-21
 outline around current 4-20 to 4-21, 4-53 to 4-57
 outline of 4-20
 redrawing 4-33 to 4-34, 4-86
 scroll bars in 4-5 to 4-6, 4-8

scrolling 4-24, 4-25, 4-36 to 4-37, 4-88 to 4-90
 searching in 4-43 to 4-44, 4-90 to 4-91
 selection algorithm, customizing 4-14 to 4-15, 4-38 to 4-39
 selection of items in 4-9 to 4-15, 4-34 to 4-39
 size box in 4-8
 type selection in 4-20, 4-45 to 4-48
 visible cells of 4-23
 LLastClick function 4-24, 4-96
 LNew function 4-70 to 4-72
 LNextCell function 4-35, 4-93 to 4-94
 LoadIconCache function 5-54 to 5-55
 LoadResource procedure 1-80 to 1-81
 LoadScrap function 2-41
 localization guidelines, for Help Manager 3-20
 LRect procedure 4-95
 LScroll procedure 4-89 to 4-90
 LSearch function 4-90 to 4-91
 LSetCell procedure 4-79 to 4-80
 LSetDrawingMode procedure 4-32, 4-87
 LSetSelect procedure 4-36, 4-78 to 4-79
 LSize procedure 4-91 to 4-92
 LUpdate procedure 4-86

M

'mach' resource type 8-6, 8-20 to 8-21, 8-29, 8-84 to 8-85
 machine resources 8-6, 8-20 to 8-21, 8-29, 8-84 to 8-85
 Macintosh Easy Open 7-3 to 7-75
 application guidelines 7-10 to 7-11
 capabilities 7-4 to 7-10
 defined 7-4
 and Edition Manager 7-4, 7-10
 and Finder 7-4, 7-5 to 7-7
 and Scrap Manager 7-4, 7-10
 and Standard File Package 7-4, 7-8 to 7-9
 MakeIconCache function 5-53 to 5-54
 manufacturer code for components 6-4, 6-39, 6-53
 match functions 4-99 to 4-100
 MaxSizeRsrc function. *See* GetMaxResourceSize function
 menu commands
 Clear (Edit menu) 2-6
 Copy (Edit menu) 2-6
 handling 2-19
 Cut (Edit menu) 2-6, 2-10 to 2-11
 handling 2-15 to 2-19
 Paste (Edit menu) 2-6, 2-10 to 2-11
 handling 2-20 to 2-25, 2-28 to 2-31
 Show/Hide Balloons (Help menu) 3-7
 Show/Hide Clipboard (Edit menu) 2-10, 2-25
 menu help resources. *See* 'hmenu' resource type

menu items. *See also* menu commands
 adding to Help menu 3-90 to 3-93
 help balloons for 3-39 to 3-51
 menus. *See also* menu commands
 help balloons for 3-14 to 3-16, 3-27 to 3-51, 3-103 to 3-105
 menu titles, help balloons for 3-36 to 3-38
 mini 4-bit color icon resources 5-7
 mini 8-bit color icon resources 5-7
 mini icon list resources 5-7
 mini icons, defined 5-7
 missing items, help balloons for
 in 'hdlg' resources 3-54 to 3-56
 in 'hmenu' resources 3-33 to 3-36
 in 'hovr' resources 3-88
 'mnr' resource type 8-11, 8-56, 8-88
 monitor code resources 8-11, 8-56
 Monitors control panel 8-3, 8-9
 features 8-10
 and the Options dialog box 8-10, 8-52
 monitors extension functions 8-61 to 8-73, 8-78 to 8-82
 allocating memory for 8-66
 error handling 8-63, 8-82
 and keyboard-related events 8-73
 modifying the rectangle resource for 8-66
 and mouse-related events 8-71
 performing initialization 8-68
 monitors extensions 8-9 to 8-12
 optional resources 8-11, 8-56
 required resources 8-11, 8-51 to 8-56, 8-83
 user interface guidelines for 8-49 to 8-51, 8-52 to 8-56
 mouse-down events, in lists 4-33
 'movv' scrap format type 2-33
 multicolumn lists, containing fewer items than columns 4-7

N

Network control panel 4-7
 NewIconSuite function 5-32 to 5-33
 NIL handle
 in a resource map 1-18
 returned by Resource Manager routines 1-14, 1-51
 'nrct' resource type 8-6, 8-13, 8-15 to 8-17, 8-82, 8-85 to 8-86

O

OpenComponent function 6-46
 OpenComponentResFile function 6-72

OpenDefaultComponent function 6-7 to 6-8, 6-45 to 6-46
 open request 6-19 to 6-20
 OpenResFile function 1-66 to 1-68
 'open' resource type 7-10, 7-13 to 7-14, 7-44
 OpenRFPPerm function 1-64 to 1-66
 Options dialog box 8-10
 and controls for superusers 8-51
 defining the display area for controls 8-50
 and monitors extension controls 8-54
 standard controls 8-10
 supplying the icon for 8-57
 outlining the current list 4-53 to 4-57

P

package resource IDs 1-128
 package resources 1-128
 packages 1-128
 'PACK' resource type 1-128
 partial resources 1-40 to 1-41
 Paste command (Edit menu) 2-6, 2-10 to 2-11
 PBDTAddAPPL function 9-18 to 9-19
 PBDTAddIcon function 9-17 to 9-18
 PBDTCloseDown function 9-11
 PBDTDelete function 9-26
 PBDTFlush function 9-23
 PBDTGetAPPL function 9-15 to 9-16
 PBDTGetComment function 9-16
 PBDTGetIcon function 9-12 to 9-14
 PBDTGetIconInfo function 9-14 to 9-15
 PBDTGetInfo function 9-24
 PBDTGetPath function 9-9 to 9-10
 PBDTOpenInform function 9-10 to 9-11
 PBDTRemoveAPPL function 9-21
 PBDTRemoveComment function 9-22
 PBDTReset function 9-25
 PBDTSetComment function 9-19 to 9-20
 'PICT' resource type, and help messages 3-24
 'PICT' scrap format type 2-33
 picture resources, and help messages 3-24
 PlotCIconHandle function 5-16, 5-26 to 5-27
 PlotCIcon procedure 5-15, 5-25 to 5-26
 PlotIconHandle function 5-15, 5-24 to 5-25
 PlotIconID function 5-10, 5-20 to 5-22
 PlotIconMethod function 5-22 to 5-23
 PlotIcon procedure 5-14, 5-23 to 5-24
 PlotIconSuite function 5-11, 5-13, 5-35 to 5-38
 PlotSICNHandle function 5-16, 5-27 to 5-28
 point-to-point translation 7-30
 preferences files
 and control panels 8-30
 default icon for 1-130

- and monitors extensions 8-71
- resources in 1-13
- Preferences folder, icon for 1-132
- printer, determining type in use 1-127
- PrintMonitor Documents folder, icon for 1-132
- private scrap
 - reading data from 2-24 to 2-25, 2-25 to 2-26
 - writing data to 2-18 to 2-19
- progress dialog box. *See* translation progress dialog box
- PScrapStuff data type 2-32
- PtInIconID function 5-48
- PtInIconMethod function 5-49 to 5-50
- PtInIconSuite function 5-47
- PtInRect function
 - using to determine if a cell is in a list 4-24
 - using to determine if a list cell is visible 4-23
- PutScrap function 2-36 to 2-37

Q

query documents, default icon for 1-130

R

- ReadPartialResource procedure 1-111 to 1-113
- 'RECT' resource type 8-11, 8-52
- rectangle help resources. *See* 'hrct' resource type
- rectangle positions resource 8-6, 8-15, 8-85 to 8-86
- rectangle resources 8-11, 8-52 to 8-53
- RectInIconID function 5-51
- RectInIconMethod function 5-52 to 5-53
- RectInIconSuite function 5-50 to 5-51
- reference number 9-5. *See also* file reference numbers
 - determining for desktop database 9-9 to 9-11
 - of desktop database 9-5
- RegisterComponent function 6-31, 6-57 to 6-59
- RegisterComponentResourceFile function 6-61 to 6-62
- RegisterComponentResource function 6-31, 6-59 to 6-61
- register request 6-23 to 6-24
- ReleaseResource procedure 1-22, 1-107
- RemoveResource procedure 1-109 to 1-110
- request codes, for components 6-19, 6-29
- ResEdit resource editor 1-15 to 1-17
- ResErr global variable 1-51
- ResError function 1-51 to 1-52
- resource attributes
 - defined 1-8
 - getting and setting 1-81 to 1-87
- resource files. *See* resource forks

- resource forks 1-4 to 1-6. *See also* current resource file
 - closing 1-110 to 1-111
 - creating 1-25 to 1-28, 1-53 to 1-58
 - file format for 1-121 to 1-125
 - getting and setting attributes of 1-116 to 1-119
 - opening 1-24 to 1-30, 1-58 to 1-68
 - reading resources from 1-30 to 1-35, 1-71 to 1-81
 - resource data, format of 1-122
 - resource header, format of 1-122
 - resource name list, format of 1-124
 - resource type list, format of 1-123
 - writing resources to 1-36 to 1-40, 1-92 to 1-95
- resource IDs 1-46 to 1-49
 - defined 1-6
 - for function key resources 1-129
 - getting unique 1-95 to 1-97
 - for owned resources 1-47
 - for packages 1-128 to 1-129
 - restrictions on 1-46 to 1-47
- Resource Manager 1-3 to 1-148
 - data structure, types, and IDs 1-42 to 1-49
 - initializing 1-50
 - routines in 1-49 to 1-120
 - testing for features of 1-13 to 1-14
- resource maps
 - accessing entries in 1-119 to 1-120
 - defined 1-9
 - format of 1-123
 - ROM, inserting in resource search path 1-134 to 1-135
- resources. *See also* resource types
 - bundle. *See* 'BNDL' resource type
 - card 8-11, 8-52, 8-87
 - changing 1-87 to 1-91
 - color icon 5-6
 - component 6-33, 6-61 to 6-62, 6-80 to 6-85, 7-18, 7-20 to 7-21
 - control device code 8-7, 8-25 to 8-48, 8-74 to 8-77
 - copying 1-24
 - counting and indexing 1-34
 - counting and listing resource types 1-97 to 1-101
 - creating 1-15 to 1-18
 - default help override. *See* 'hovr' resource type
 - defined 1-3
 - dialog color table. *See* 'dctb' resource type
 - dialog-item help. *See* 'hdlg' resource type
 - disposing of 1-106 to 1-110
 - file reference. *See* 'FREF' resource type
 - Finder icon help. *See* 'hfdr' resource type
 - font information 8-7, 8-23, 8-86
 - function keys 1-129
 - gamma tables 8-59. *See* 'gama' resource type
 - getting and setting information about 1-81 to 1-87
 - getting a unique ID 1-95 to 1-97
 - getting handles to 1-18 to 1-21

- icon. *See* icon resources
- icon family. *See* icon families
- item color table 8-23. *See* 'ictb' resource type
- item list. *See* 'DITL' resource type
- kind 7-11, 7-14 to 7-15, 7-45 to 7-46
- list definition procedure 4-58, 4-98
- locations of, typical 1-12
- machine 8-6, 8-20 to 8-21, 8-29, 8-84 to 8-85
- menu help. *See* 'hmnu' resource type
- modifying 1-87 to 1-91
- monitor code 8-11, 8-56, 8-88
- open 7-10, 7-13 to 7-14, 7-44
- owned 1-47
- partial 1-40 to 1-41, 1-111 to 1-116
- in preferences files 1-13
- reading 1-30 to 1-35, 1-40 to 1-41, 1-71 to 1-81
- rectangle 8-11, 8-52 to 8-53
- rectangle help. *See* 'hrct' resource type
- rectangle positions 8-6, 8-15, 8-85 to 8-86
- releasing and detaching 1-22 to 1-24
 - and ResEdit 1-15 to 1-17
 - and Rez resource compiler 1-15 to 1-17
- ROM 1-134 to 1-136
- ROM override 1-135 to 1-136
- search path for 1-10 to 1-12
- size of, getting 1-104 to 1-106
- small icon 5-6, 5-13, 5-16 to 5-17
- for standard icons 1-129 to 1-134
- standard types 1-43 to 1-45
- system extension. *See* 'INIT' resource type
- in System file 1-7, 1-126 to 1-134
- user information 1-127 to 1-129
- window help. *See* 'hwin' resource type
- writing 1-36 to 1-40, 1-40 to 1-41, 1-92 to 1-95
- ResourceSpec data type 6-82, 7-20
- resource types. *See also* resources
 - available for application's use 1-43 to 1-45
 - 'BNDL' 8-7, 8-22, 8-57
 - 'card' 8-11, 8-51 to 8-52, 8-87
 - 'cdev' 8-7, 8-25 to 8-47, 8-74 to 8-77
 - 'cicn' 5-6
 - 'dctb'. *See* 'dctb' resource type
 - defined 1-6
 - 'DITL'. *See* 'DITL' resource type 8-6
 - 'finf' 8-7, 8-23, 8-86
 - 'FKEY' 1-129
 - 'gama' 8-12, 8-59
 - 'hdlg'. *See* 'hdlg' resource type
 - 'hfdr'. *See* 'hfdr' resource type
 - 'hmnu'. *See* 'hmnu' resource type
 - 'hovr'. *See* 'hovr' resource type
 - 'hrct'. *See* 'hrct' resource type
 - 'hwin'. *See* 'hwin' resource type
 - 'icl4' 5-4
 - 'icl8' 5-5
 - 'icm#' 5-7
 - 'icm4' 5-7
 - 'icm8' 5-7
 - 'ICN#' 5-4
 - 'ICON' 5-6, 5-13 to 5-15
 - 'ics#' 5-4
 - 'ics4' 5-5
 - 'ics8' 5-5
 - 'ictb'. *See* 'ictb' resource type
 - 'INIT'. *See* 'INIT' resource type
 - 'kind' 7-11, 7-14 to 7-15, 7-45 to 7-46
 - 'LDEF' 4-7, 4-58, 4-98
 - list of standard 1-43 to 1-45
 - 'mach' 8-6, 8-20 to 8-21, 8-29, 8-84 to 8-85
 - 'mntr' 8-11, 8-56, 8-88
 - 'nrct' 8-6, 8-13, 8-15 to 8-17, 8-82, 8-85 to 8-86
 - 'open' 7-10, 7-13 to 7-14, 7-44
 - 'PACK' 1-128
 - 'RECT' 8-11, 8-50, 8-52
 - reserved for Operating System's use 1-46
 - ResType data type 1-42
 - 'ROv#' 1-135 to 1-136
 - 'SICN' 5-6
 - 'STR#'. *See* 'STR#' resource type
 - 'STR '. *See* 'STR ' resource type
 - 'thng'. *See* 'thng' resource type
 - ResType data type 1-42
 - resume events
 - handling 2-25 to 2-26
 - updating type-selection threshold after 4-46
 - Rez resource compiler 1-15 to 1-17
 - RGetResource function 1-78 to 1-79
 - Right Arrow key 4-48
 - RmveResource procedure. *See* RemoveResource procedure
 - ROM override resource 1-135 to 1-136
 - ROM-resident resources 1-70, 1-134 to 1-136
 - overriding 1-135 to 1-136
 - ROM resource map 1-70, 1-134
 - 'ROv#' resource type 1-135 to 1-136
 - RsrcMapEntry function 1-120
 - RsrcZoneInit procedure 1-50 to 1-51

S

- sample routines
 - DoCutOrCopyCommand 2-16, 2-18, 2-29
 - DoGetFileTranslationList 7-30
 - DoIdentifyFile 7-33
- sample routines (*continued*)
 - DoMenuCommand 3-92
 - DoPasteCommand 2-21, 2-24, 2-30
 - DoPictBalloon 3-77

INDEX

- DoPictBalloon2 3-78
- DoStringListBalloon 3-79
- DoStyledTextBalloon 3-80
- DoSuspendResumeEvent 2-19, 2-25
- DoTextStringBalloon 3-77
- DoTranslateFile 7-34
- DrawerSetup 6-28
- FindAndShowBalloon 3-82
- MyActivateControlPanel 8-35
- MyAddIconToList 4-64
- MyAddItemAlphabetically 4-42
- MyAddItemsFromStringList 4-31
- MyAdjustMenusForDialogs 3-50
- MyArrowKeyExtendSelection 4-51
- MyArrowKeyInList 4-52
- MyArrowKeyMoveSelection 4-50
- MyClearAllCellData 4-40
- MyCloseControlPanel 8-45
- MyConvertScrap 2-27
- MyCopyAResource 1-24
- MyCreateAndOpenResourceFork 1-27
- MyCreateResourceFork 1-26
- MyCreateTextListInDialog 4-29
- MyCreateVerticallyScrollingList 4-27
- MyDoOpenSoundResources 1-34
- MyDrawIconFromFamily 5-10
- MyDrawIconInSuite 5-11
- MyDrawListBorder 4-30
- MyDrawOutline 4-54
- MyDrawRect 8-70
- MyDrawThisIcon 5-12
- MyFindNewCellLoc 4-49
- MyFindVideoComponent 6-9
- MyGetAndPlayRewardSoundResource 1-29
- MyGetAndPlaySoundResource 1-22
- MyGetCellData 4-41
- MyGetCompInfo 6-10
- MyGetComponent 6-10
- MyGetDirectAccessToCellData 4-41
- MyGetFirstSelectedCell 4-34
- MyGetIconData 5-13
- MyGetLastSelectedCell 4-35
- MyHandleEditCommand 8-46
- MyHandleHitInDialogItem 8-41
- MyHandleHits 8-72
- MyHandleInitMsg 8-69
- MyHandleKeyEvent 8-38
- MyHandleMouseDownInList 4-33
- MyInitialize 1-25
- MyInitializeCP 8-31
- MyKeySearchInList 4-47
- MyLDEF 4-59
- MyLDEFClose 4-63
- MyLDEFDraw 4-61
- MyLDEFHighlight 4-62
- MyLDEFInit 4-60
- MyMakeCellVisible 4-37
- MyMatchNextAlphabetically 4-44
- MyMonExtend 8-64
- MyOutlineNextList 4-57
- MyOutlinePreviousList 4-57
- MyPlotAcicn 5-15
- MyPlotAcicnWithAlignAndTransform 5-16
- MyPlotAnICON 5-14
- MyPlotAnICONWithAlignAndTranform 5-15
- MyPlotAnSICNWithAlignAndTranform 5-16
- MyReadAPartial 1-41
- MyResetTypeSelection 4-46
- MySaveWindowPosition 1-38 to 1-39
- MySearchPartialMatch 4-43
- MySelectOneCell 4-36
- MySetCellSizeForIconList 4-63
- MySetupData 8-67
- MySetWindowPosition 1-32
- MyShowBalloonForDimMenuTitle 3-124
- MyTrackList 4-55
- MyUpdateControlPanel 8-43
- MyUpdateList 4-33
- MyUpdateListOutlines 4-56
- OvalCanDo 6-22
- OvalClick 6-27
- OvalClose 6-21
- OvalDraw 6-27
- OvalDrawer 6-16 to 6-18
- OvalErase 6-27
- OvalMoveTo 6-28
- OvalOpen 6-20
- OvalSetUp 6-26
- RectangleDrawer 6-36
- RiverCP 8-27
- TranslateEntry 7-25
- scrap
 - converting data between a private scrap and 2-9 to 2-10, 2-26 to 2-28
 - converting data between the TextEdit scrap and 2-28 to 2-30
 - defined 2-4
 - location of 2-12 to 2-14
 - reading data from 2-20 to 2-24, 2-25 to 2-26
 - translating format of 7-10, 7-19, 7-21
 - using a private 2-4, 2-9
 - writing data to 2-8 to 2-10, 2-15 to 2-17, 2-19 to 2-20
- scrap file 2-33, 2-40
- scrap format types 2-33
 - 'movv' 2-33
 - 'PICT' 2-33
 - 'snd ' 2-33
 - standard 2-7, 2-33
 - 'styl' 2-33
 - 'TEXT' 2-33

- scrap information record 2-32 to 2-33
- Scrap Manager
 - data types in 2-32 to 2-33
 - routines in 2-34 to 2-41
 - and Standard File Package 2-31
 - testing for features 2-14
 - and TextEdit 2-28 to 2-30
 - and Translation Manager 2-7, 2-10, 7-4, 7-10, 7-11
- ScrapStuff data type 2-32 to 2-33
- ScrapTranslationList data type 7-49 to 7-50
- scrap translation lists 7-49 to 7-50
- scrap translation systems 7-5
- ScrapType data type 7-18
- scrap types. *See also* scrap format types
 - 'stxt' 7-19
 - 'styl' 7-19
 - 'TEXT' 7-19
- ScrapTypeSpec data type 7-49
- scrap type specifications 7-49
- screen shots 1-129
- scroll bars
 - help balloon for 3-16
 - in lists 4-5 to 4-6, 4-8
 - width of 4-23
- search path, for resources 1-10 to 1-12
- SetComponentInstanceA5 procedure 6-68
- SetComponentInstanceError procedure 6-28, 6-69 to 6-70
- SetComponentInstanceStorage procedure 6-19, 6-66 to 6-67
- SetComponentRefcon procedure 6-35, 6-70 to 6-71
- SetControlValue procedure 8-30, 8-72
- SetDefaultComponent function 6-78 to 6-79
- SetIconCacheData function 5-56
- SetIconCacheProc function 5-57
- SetResAttrs procedure 1-85 to 1-87
- SetResFileAttrs procedure 1-118 to 1-119
- SetResInfo procedure 1-82 to 1-83
- SetResLoad procedure 1-79 to 1-80
- SetResourceSize procedure 1-115 to 1-116
- SetResPurge procedure 1-94 to 1-95
- SetSuiteLabel function 5-40 to 5-41
- SetTranslationAdvertisement function 7-35, 7-51 to 7-52
- Shift key, use of in lists 4-10 to 4-11
- Show/Hide Balloons command (Help menu) 3-7
- Show/Hide Clipboard command (Edit menu) 2-10, 2-25
- 'SICN' resource type 5-6
 - drawing 5-13 to 5-17
- signatures, finding applications with specific 9-15 to 9-16
- 16-by-16 pixel (small) icons 5-4, 5-6
- size boxes
 - help balloon for 3-16
 - using in lists 4-8
- SizeResource function. *See* GetResourceSizeOnDisk function
- small 4-bit color icon resources, as part of an icon family 5-5
- small 8-bit color icon resources, as part of an icon family 5-5
- small icon list resources, as part of an icon family 5-4
- small icons 5-6. *See also* icon resources
- 'snd' scrap format type 2-33
- standard file dialog boxes
 - help balloons for 3-15
 - icons in 1-133 to 1-134
- Standard File Package
 - default icons used by 1-133
 - file filter functions 7-11, 7-16
 - icons used by 1-133 to 1-134
 - and Macintosh Easy Open 7-4, 7-8 to 7-9
 - small color icons in dialog boxes 7-8
 - and Scrap Manager 2-31
- standard icons 1-129 to 1-134
 - desktop 1-133 to 1-134
 - documents and applications 1-130
 - folders 1-131 to 1-133
- StandardOpenDialog function 7-14, 7-16
- Startup Items folder, icon for 1-132
- startup process, and Resource Manager 1-50 to 1-51
- static type lists. *See* 'open' resource type
- static windows, help balloons for 3-63 to 3-74
- stationery documents
 - default icon for 1-130
 - and Macintosh Easy Open 7-10
- 'STR#' resource type, and help messages 3-24
- string list resources, and help messages 3-24
- string resources, and help messages 3-24
- strings, putting into list cells 4-31
- 'STR' resource type, and help messages 3-24
- structure regions of help balloons 3-93
- style resources, and help messages 3-24
- styles, of text in lists 4-7
- 'styl' resource type, and help messages 3-24
- 'styl' scrap format type 2-33
- suspend events, handling 2-19 to 2-20
- system extensions
 - and control panels 8-8
 - default icon for 1-130
 - and monitors extension files 8-61
 - where to install 8-8
- System file
 - file reference number for resource fork 1-50
 - icon resources in 1-129 to 1-134
 - resources in 1-126 to 1-134
 - application icons 1-129 to 1-130
 - desktop icons 1-133
 - document icons 1-129 to 1-130

- folder icons 1-131
- Standard File Package icons 1-133 to 1-134
- System Folder icons 1-132
- user information 1-127 to 1-128
- System Folder, icon for 1-132
- system resource map 1-70, 1-134
- system startup 1-50

T

- Tab key, using to change active list 4-21
- target request 6-25 to 6-26
- TECopy procedure 2-28, 2-31
- TECut procedure 2-28, 2-31
- TEFromScrap procedure 2-28, 2-30
- TEPaste procedure 2-28, 2-31
- TEToScrap procedure 2-28, 2-30
- TextEdit, and Scrap Manager 2-28 to 2-30
- 'TEXT' file type, proper use of 7-11
- text resources, and help messages 3-24
- 'TEXT' resource type, and help messages 3-24
- 'TEXT' scrap format type 2-33
- 32-by-32 pixel (large) icons 5-4
- 'thng' resource type
 - format of 6-80 to 6-85
 - Rez input for 6-33
 - and translation extensions 7-19 to 7-21
- tip function, creating 3-128, 3-130 to 3-131
- tips of help balloons
 - defined 3-9
 - for help balloons in menus 3-29
 - specifying in dynamic windows 3-80
 - specifying in 'hdlg' resources 3-56
 - specifying in 'hrcr' resources 3-67
- transforms, for Icon Utilities routines
 - constants for 5-37
 - defined 5-8
- TranslateEntry function 7-24 to 7-26
- TranslateFile function 7-18, 7-42 to 7-43
- translation extensions 7-18 to 7-35, 7-46 to 7-62
 - data types used in 7-46 to 7-50
 - defined 7-4
 - opening resource files 7-27
 - resources in 7-22 to 7-23
 - routines defined in 7-27 to 7-35, 7-54 to 7-62
 - routines used in 7-50 to 7-54
 - runtime environment 7-26
- translation file types 7-19
- translation groups 7-28
- Translation Manager 7-3 to 7-18, 7-36 to 7-46
 - and Edition Manager 7-12
 - relation to Macintosh Easy Open 7-4
 - resources in 7-43 to 7-46

- routines in 7-36 to 7-43
 - and Scrap Manager 2-7, 2-10, 7-11
 - and Standard File Package 7-11, 7-12
 - testing for availability 7-12, 7-36
- translation of file formats 7-3 to 7-58
 - explicit 7-17 to 7-19, 7-36 to 7-43
 - implicit 7-6 to 7-8
- translation progress dialog box
 - advertisement in 7-22
 - displaying 7-35
 - and implicit translation 7-10
 - routines for displaying 7-51 to 7-54
 - shown 7-7
 - source of types 7-47
 - updating 7-35
- translation systems 7-4
- 12-by-16 pixel (mini) icons 5-7
- TypesBlock data type 7-37
- type selection
 - introduced 4-20
 - supporting 4-45 to 4-48
- type-selection threshold 4-45
 - formula for computing 4-46

U

- UncaptureComponent function 6-76
- UniqueID function 1-96
- Unique1ID function 1-96 to 1-97
- UnloadScrap function 2-40
- UnregisterComponent function 6-62
- unregister request 6-24 to 6-25
- Up Arrow key 4-48
- update events
 - handled by the Help Manager 3-26, 3-81 to 3-82
 - in lists 4-33 to 4-34
- UpdateResFile procedure 1-92 to 1-93
- UpdateTranslationProgress function 7-35, 7-52 to 7-54
- user comments
 - removing 9-22
 - retrieving from desktop database 9-16
 - setting 9-19 to 9-20
- UseResFile procedure 1-69 to 1-71
- user information resources 1-127
- user interface guidelines
 - for control panels 8-12 to 8-14
 - for handling copy and paste 2-6 to 2-8, 2-10 to 2-11
 - for Help Manager 3-18 to 3-23, 3-37 to 3-38, 3-39 to 3-40, 3-57 to 3-58, 3-70 to 3-71
 - for lists 4-4 to 4-21
 - for monitors extensions 8-49 to 8-51
- user name 1-127

V

variation codes for help balloons 3-9 to 3-11
version request 6-22 to 6-23
video cards
 icons for 8-57
 and Monitors control panel 8-10
 and sResource data structure 8-57
volumes, Finder's desktop database for 9-3 to 9-26

W, X, Y

window frames, help balloons for 3-13 to 3-16, 3-87 to 3-89
window help resources. See 'hwin' resource type
windows
 containing multiple lists 4-20 to 4-21
 help balloons for 3-13 to 3-16, 3-63 to 3-84, 3-87 to 3-89
 position of lists in 4-8
WritePartialResource procedure 1-113 to 1-115
WriteResource procedure 1-93 to 1-94

Z

ZeroScrap function 2-35
 using TextEdit with 2-28, 2-35
zoom boxes, help balloons for 3-14 to 3-16, 3-87 to 3-89

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 Pro printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe[™] Illustrator and Adobe Photoshop. 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

Michael Abramowicz, Patria Brown,
Sean Cotter, Sharon Everson,
Tony Francis, Judy Melanson,
Tim Monroe

DEVELOPMENTAL EDITORS

Antonio Padial, Jeanne Woodward,
Beverly Zegarski

INDEX SPECIALIST

Laurel Rezeau

ILLUSTRATORS

Barbara Carey, Bruce Lee

COVER DESIGNER

Barbara Smyth

PRODUCTION EDITORS

Patricia Christenson, Alan Morgenegg

PROJECT MANAGER

Patricia Eastman

Special thanks to Elizabeth Moller,
Dean Yu, Beverly Zegarski.

Acknowledgments to Dave Collins,
Chris DeRossi, Bill Guschwan,
Peter Hoddie, Nick Kledzik,
Wendy Krafft, Guillermo Ortiz,
Frank Stanbach, John Wang, and the
entire *Inside Macintosh* team.