

Menu Manager

You can use the Menu Manager to create and manage the menus in your application. Menus allow the user to view or choose from a list of choices and commands that your application provides.

All Macintosh applications should provide these standard menus: the Apple menu, the File menu, and the Edit menu. If you include an Apple menu as a menu of your application, the Menu Manager automatically adds the Help and Application menus to your application's menu bar; it adds the Keyboard menu if more than one keyboard layout or input method is installed.

Menus are typically stored as resources. This chapter describes the menu-related resources. See the chapter "Introduction to the Macintosh Toolbox" in this book for general information on resources and see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on Resource Manager routines. See *Macintosh Human Interface Guidelines* for additional examples of menus that incorporate many principles of user interface design. *Inside Macintosh: Text* contains further information on localizing your application for worldwide markets.

You can choose to provide help balloons for your application's menus. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for additional details on providing help balloons for your application's menus.

You often present a dialog box to the user as a result of the user's choice of a menu command that requires additional information before you can perform the command. See the chapter "Dialog Manager" later in this book for information on creating dialog boxes in your application.

For additional information on processing events, see the chapter "Event Manager" earlier in this book.

This chapter provides an introduction to menus and the menu bar, and it then describes

- various types of menus your application can use
- standard menus
- how to store menus as resources
- how to create menus
- how to create a menu bar
- how to change characteristics of menu items
- how to add items to a menu

Introduction to Menus

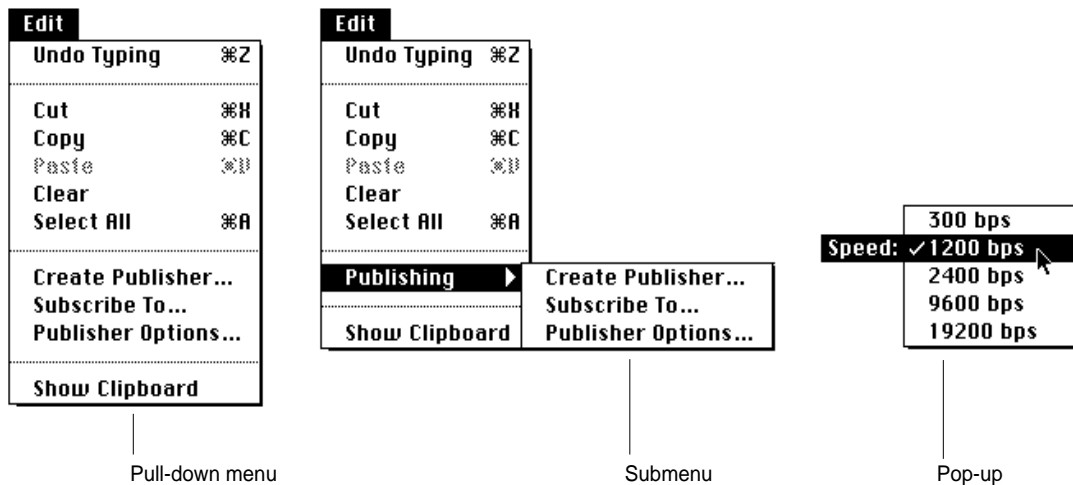
A **menu** is a user interface element you can use in your application to allow the user to view or choose an item from a list of choices and commands that your application provides. Menus can appear in several different forms: pull-down menus, hierarchical menus, and pop-up menus.

Menu Manager

A **pull-down menu** is identified by a menu title (a word or an icon) in the menu bar. Your application can use pull-down menus in the menu bar to allow users to choose a command or perform an action on a selected object. A **pop-up menu** is a menu that does not appear in the menu bar, but appears elsewhere on the screen when the user presses the mouse button while the cursor is in a particular place. Pop-up menus are most often accessed from a dialog box. Your application can use pop-up menus to let the user select one choice from a list of many or to set a specific value. A **submenu** refers to a menu that is attached to another menu. A menu to which a submenu is attached is referred to as a **hierarchical menu**.

Figure 3-1 shows examples of a pull-down menu, a submenu, and a pop-up menu.

Figure 3-1 A pull-down menu, a submenu, and a pop-up menu



The standard **menu bar** extends across the top of the startup screen and contains the title of each available pull-down menu. Your application's menu bar should always provide at least the Apple menu, the File menu, and the Edit menu. When you insert the Apple menu in your application's menu bar, the Menu Manager automatically adds the Help and Application menus to your application's menu bar. It also adds the Keyboard menu if multiple script systems are installed or if a certain bit is set in the 'itlc' resource. Your application can include as many other menus as fit on the smallest screen on which your application runs, and you should create only as many items as are essential to your application.

If your application uses a menu bar, you should make it always visible and available for use. If you do not always wish to display the menu bar (for example, if your application allows the user to view a screen presentation), you can give the user the option of viewing the presentation on the entire screen without the menu bar showing. However, you must provide a way, such as a keyboard equivalent for a command, for the user to access the menu bar or to make the menu bar reappear.

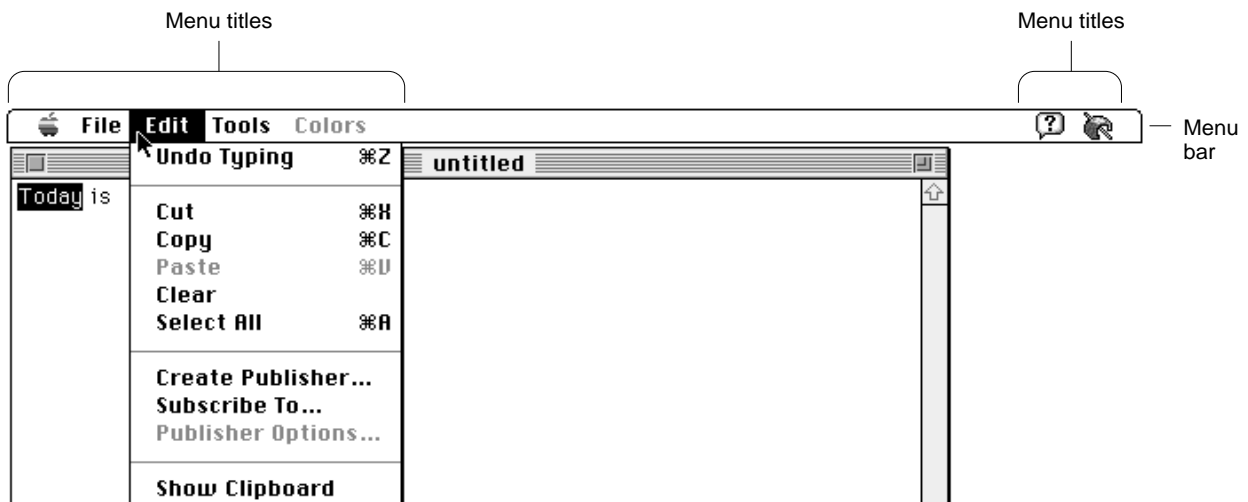
Using menus in your application allows the user to explore many possible choices and options without having to choose any particular one. By providing help balloons for

Menu Manager

your menus, you further allow users to learn about the possible actions or consequences of a particular menu choice without having to choose the menu command to find out what happens.

Figure 3-2 shows the SurfWriter application's menu bar with the Edit menu displayed. This application supports the standard Apple, File, and Edit menus; the Help and Application menus; and in addition supports two other application-specific menus.

Figure 3-2 The SurfWriter application's menu bar with the Edit menu displayed



Each menu has a menu title and one or more menu items associated with it. You should name each menu so that the title describes or relates to the actions the user can perform from that menu. For example, the Edit menu of a typical application contains commands that let the user edit the contents of a document.

Your application can disable any menu. The Menu Manager indicates that a menu is disabled by dimming its menu title. (In Figure 3-2, the Colors menu is disabled.) The Menu Manager dims all menu items of a disabled menu. The user can still pull down and examine the items in a disabled menu, but cannot choose any of the items.

Your application can also disable individual menu items. The Menu Manager dims the appearance of a disabled item and does not highlight it when the user rests the cursor on that item. If the user releases the mouse button while the cursor is over a disabled menu item, the Menu Manager reports that the user did not choose a menu command. (You can determine if this happened, however, by using the `MenuChoice` function.)

In Figure 3-2, the Paste command is disabled; the SurfWriter application disables the Paste command if the Clipboard is empty. SurfWriter also disables the Publisher Options command when the current selection does not contain a publisher or a subscriber. As explained in the chapter “Help Manager” in *Inside Macintosh: More Macintosh Toolbox*, your application should provide help balloons for disabled items that describe what the item normally does and explain why the item is not available at this time.

Menu Manager

Note

Although `enabled` and `disabled` are the constants you use in a resource file to display or to dim menus and menu items, you shouldn't use these terms in your help balloons or user documentation. Instead use the terms *menus*, *menu commands*, and *menu items* for those that are enabled, and use the terms *not available* and *dimmed* to distinguish those that have been disabled. ♦

The Menu Manager highlights an enabled menu item when the cursor is over it. Enabled items do not have a dim appearance and can be chosen by the user.

Your application specifies whether menu items are enabled or disabled when it first defines and creates a menu. You can also disable or enable menu items at any time after creating a menu. You should enable a menu item whenever your application allows the user to choose the action associated with that item, and you should disable an item whenever the user cannot choose that item. For example, if the user selects text and then presses the mouse button while the cursor is in the menu bar, you should enable the Copy command in the Edit menu. You should disable the Copy command in the Edit menu if the user has not selected anything to copy.

Your application can also specify other characteristics of menu items, such as whether the item has a marking character next to its text (for example, a checkmark) or whether the item has a keyboard equivalent (for example, Command-C for the Copy command). “Menu Items” beginning on page 3-12 describes the characteristics of individual menu items in more detail.

The user typically chooses commands by moving the cursor to the menu bar and pressing the mouse button while the cursor is over a menu title. When the user presses the mouse button while the cursor is in the menu bar, your application should call the `MenuSelect` function. The `MenuSelect` function tracks the mouse, displays and removes menus as the user drags the cursor through the menu bar, highlights menu titles as the user drags the cursor over them, displays the menu items associated with a selected menu, highlights enabled menu items as the user drags through a menu, and handles all user activity until the user releases the mouse button.

The user chooses a menu item by releasing the mouse button while the cursor is over a particular enabled menu item. When the user chooses a menu item, the Menu Manager briefly blinks the chosen menu item (to confirm the choice) and then removes the menu from the display. The Menu Manager leaves the title of the chosen menu highlighted to provide feedback to the user.

The `MenuSelect` function returns information that allows your application to determine which menu item was chosen. Your application then typically responds by performing the desired command. When your application completes the requested action, your application should unhighlight the menu title, indicating to the user that the action is complete.

The user can move the cursor out of the menu (or menu bar) at any time; the Menu Manager displays any currently visible menu as long as the mouse button is pressed. (If the cursor is outside of the menu, the Menu Manager removes any highlighting of the menu item.) If the user releases the mouse button outside of a menu, the `MenuSelect`

Menu Manager

function reports that the user did not choose a menu item, and the Menu Manager removes any currently visible menu. Your application should not take any action if the user does not choose a menu item.

Menu and Menu Bar Definition Routines

The menu definition procedure and menu bar definition function define the general appearance and behavior of menus. The Menu Manager uses these routines to display and perform basic operations on menus and the menu bar.

A **menu definition procedure** performs all the drawing of menu items within a menu. When you define a menu, you specify its menu definition procedure. The Menu Manager uses the specified menu definition procedure to draw the menu items in a menu, determine which item the user chose from a menu, insert scrolling indicators as items in a menu, and calculate the menu's dimensions.

A **menu bar definition function** draws the menu bar and performs most of the drawing activities related to the display of menus when the user moves the cursor between them. Unless you specify otherwise, the Menu Manager uses the standard menu bar definition function to manage your application's menu bar. The Menu Manager uses the standard menu bar definition function to draw the menu bar, clear the menu bar, determine whether the cursor is in the menu bar or any currently displayed menu, calculate the left edges of menu titles, highlight a menu title, invert the entire menu bar, erase the background color of a menu and draw the menu's structure (shadow), and save or restore the bits behind a menu.

Apple provides a standard menu definition procedure and standard menu bar definition function. These definition routines are stored as resources in the System file. The standard menu definition procedure is the 'MDEF' resource with resource ID 0. The standard menu bar definition function is the 'MBDF' resource with resource ID 0.

When you define your menus and menu bar, you specify the definition routines that the Menu Manager should use when managing them. You'll usually want to use the standard definition routines for your application. However, if you need a feature not provided by the standard menu definition procedure (for example, if you want to include more graphics in your menus), you can choose to write your own menu definition procedure. See "Writing Your Own Menu Definition Procedure" beginning on page 3-87 for more information. While the Menu Manager does allow you to specify your own menu bar definition function, Apple recommends that you use the standard menu bar definition function.

The Menu Bar

Each application has its own menu bar. The menu bar of an application applies to only that application. You usually define a menu bar for your application by providing a **menu bar** ('MBAR') **resource** that lists the order and resource ID of each menu that appears in your menu bar. You define the menu title and the individual characteristics of menu items that appear in a menu by providing a **menu** ('MENU') **resource** for each

Menu Manager

menu that appears in your menu bar. You use Menu Manager routines to create the menus and menu bar based on these resource definitions.

Your application can change the enabled state of a menu, add menus to or remove menus from its menu bar, or change the characteristics of any menu items. Whenever your application changes the enabled state of a menu or the number of menus in its menu bar, your application must call the `DrawMenuBar` procedure to update the menu bar's appearance.

The menu bar (as defined by the standard menu bar definition function) is white, with a height that is tall enough to display menu titles in the height of the system font and system font size, and with a black lower border that is one pixel tall. The menu bar is as wide as the screen and always appears on the monitor designated by the user as the startup screen. (The user selects a startup screen using the Monitors control panel.) The menu bar appears at the top of the screen, and nothing except the cursor can appear in front of it. Figure 3-3 shows the menu bar of the SurfWriter application.

Figure 3-3 The menu bar of the SurfWriter application



The menu bar helps to indicate the active application. The active application is the one whose menu bar is currently showing and whose icon appears as the menu title of the Application menu.

The titles of menus appear in the menu bar. A menu title is a text string (except for the Apple, Help, Keyboard, and Application menus, the titles of which contain a small icon). Menu titles always appear in the system font and system font size (for Roman scripts, the system font is Chicago and the system font size is 12).

You can insert any number of menu titles in the menu bar; however, less than 10 is usually optimum. Keep in mind that not all users have the same size monitor. Design your menu bar so that all titles can fit in the menu bar of the smallest screen on which your application can run. You should also consider localization issues when designing the number of menus that fit in your menu bar—not all menu titles might fit in the menu bar once the menu titles are translated. For example, English text often grows 50 percent larger when translated to other languages.

Figure 3-4 shows the SurfWriter application's menu bar with menu titles that have been localized for another script system.

Figure 3-4 The SurfWriter application's menu bar localized for another script system



Menus

A menu (as defined by the standard menu definition procedure) is a list of menu items arranged vertically and contained in a rectangle. The rectangle is shaded and can extend vertically for the length of the screen. If a menu has more items than will fit on the screen, the standard menu definition procedure adds a downward-pointing triangular indicator to the last item on the screen, and it automatically scrolls through the additional items when the user moves the cursor past the last menu item currently showing on the screen. When the user begins to scroll through the menu, the standard menu definition procedure adds an upward-pointing triangular indicator to the top item on the screen to indicate that the user can scroll the menu back to its original position.

Each menu can have color information associated with it. If you do not define the colors for your menus in your application's menu color information table, the Menu Manager uses the default colors for your menus and menu bar. The default colors are black text on a white background. In most cases the default colors should meet the needs of your application. "The Menu Color Information Table Record" on page 3-98 and "The Menu Color Information Table Resource" on page 3-155 give information on how you can define colors for your application's menus.

Your application's menus can contain any number of menu items. "Menu Items" (the next section) describes the visual variations that you can use when defining your menu items.

You typically define the order and resource IDs of the menus in your application's menu bar in an 'MBAR' resource. You should define your 'MBAR' resource such that the Apple menu is the first menu in the menu bar. You should define the next two menus as the File and Edit menus, followed by any other menus that your application uses. You do not need to define the Keyboard, Help, or Application menus in your 'MBAR' resource; the Menu Manager automatically adds them to your application's menu bar if your application calls the `GetNewMBar` function and your menu bar includes an Apple menu or if your application inserts the Apple menu into the current menu list using the `InsertMenu` procedure.

You define the menu title and characteristics of each individual menu item in a 'MENU' resource. "Creating a Menu Resource" on page 3-43 describes the 'MENU' resource in more detail.

Pop-up menus do not appear in the menu bar but appear elsewhere on the screen. You often use pop-up menus in a dialog box when you want the user to be able to make a selection from a large list of choices. For example, rather than displaying the choices as a number of radio buttons, you can use a pop-up menu to display the choices at the user's convenience.

A hierarchical menu refers to either a pull-down or pop-up menu that has a submenu attached to it. (However, you should avoid attaching a submenu to a pop-up menu whenever possible, as this can make the interface more complex and less intuitive to the user.)

Menu Manager

“Creating a Pop-Up Menu” on page 3-56 gives additional information about pop-up menus, and “Creating a Hierarchical Menu” on page 3-53 describes hierarchical menus in more detail.

Menu Items

A **menu item** can contain text or can be a line (a **divider**) separating groups of choices. A divider is always dimmed, and it has no other characteristics associated with it.

Each menu item (other than dividers) can have a number of visual characteristics:

- An icon to the left of the menu item’s text. If you define an icon for a menu item, use an icon that gives a symbolic representation of the menu item’s meaning or effect. You can specify an icon, a small icon, a reduced icon, or a color icon as the icon for a menu item; however, items with small or reduced icons cannot have submenus and cannot be drawn in a script other than the current system script.
- A checkmark or other marking character to the left of the menu item’s text (and to the left of the item’s icon, if any). Use such a mark if you need to denote the status of the menu item or the mode it controls. A menu item can have a mark or a submenu, but not both.
- The symbol for the Command key (⌘) and another 1-byte character to the right of the menu item’s text (referred to as the *keyboard equivalent* of a command). Use this if your application allows the user to invoke the menu command from the keyboard by pressing the Command key and one or more other keys in combination, just as if the user had chosen the command from the menu. An item that has a keyboard equivalent cannot have a submenu, a small icon, or a reduced icon and cannot be drawn in a script other than the current system script.
- A triangular indicator to the right of the menu item’s text to indicate that the item has a submenu. A menu item that has a submenu cannot have a keyboard equivalent, a marking character, a small icon, or a reduced icon and cannot be drawn in a script other than the current system script.
- A font style—either plain or one of various other styles—for the menu item’s text. You can set the menu item’s style to bold, italic, underline, outline, shadow, or any combination of these.
- The text of the menu item. Choose words for menu items that declare the action that occurs when the user chooses the command (usually verbs, such as Print or Save). You can also use adjectives if the command changes the attribute of a selected object (for example, Bold or Italic). Unless you specify otherwise, the text of menu items appears in the script of the system font and system font size (for Roman scripts, the system font is Chicago and the system font size is 12 points). If you want a menu item’s text to appear in a script other than the current system script, you can specify a script code for the text. The Menu Manager draws the item’s text in the script identified by the script code if the script for the specified script system is installed. A menu item that is drawn in another script cannot have a submenu, small icon, or reduced icon.
- Three ellipsis points (...) as the last character in the text of the menu item. Use ellipses in the text of menu items to indicate that your application displays a dialog box that requests more information from the user before executing the command. Do not use

Menu Manager

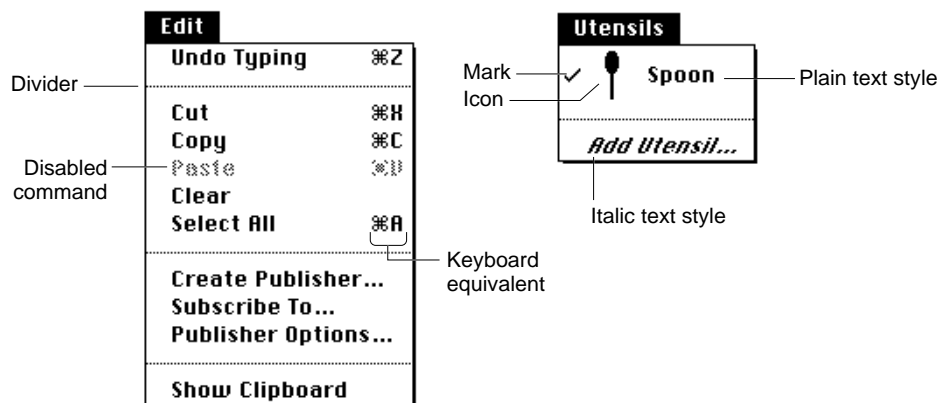
ellipses in menu items that display informational dialog boxes that do not require additional information from the user. In addition, you should not use ellipses if your application displays a confirmation alert after the user chooses a menu command. For example, if the user makes changes to a document, then chooses the Close command, your application can display a confirmation alert box, asking the user whether the document should be saved before closing. This type of command should not contain ellipses in its text.

If your application displays a dialog box requesting more information in response to the choice of a menu command, do include ellipses in the menu item's text. For example, the Open command includes ellipses in its text because the user must provide additional information: the name of the file to open. When you request more information from the user in a dialog box, you should provide an OK button or its equivalent in the dialog box that the user can select to perform the command. The dialog box should also include a Cancel button or its equivalent so that the user can cancel the command. See the chapter "Dialog Manager" in this book for information on creating dialog boxes.

- A dimmed appearance. When your application disables a menu item, the Menu Manager dims the menu item to indicate that the user can't choose it. Note that the Menu Manager dims the entire menu item, including any mark or icon, the menu text, and any keyboard equivalent symbol. Divider lines always have a dimmed appearance, regardless of whether your application enables them or not. When your application disables an entire menu, the Menu Manager dims the menu title and all menu items in that menu.

Figure 3-5 shows two menus with menu items that illustrate many of the characteristics that you can use when defining your menu items.

Figure 3-5 Two menus with various characteristics



When the primary line direction is right to left (as is the case for non-Roman script systems such as Arabic) the Menu Manager reverses the order of elements in menu items. For example, any marking character appears to the far right and any keyboard equivalent appears to the far left of the menu item's text.

Menu Manager

On a monitor that is set to display only black and white, the Menu Manager displays dividers as dotted lines. In all other cases, the Menu Manager displays dividers as appropriate, based on the current color table. For example, on a monitor set to display 4-bit color or greater, the Menu Manager typically displays dividers as gray lines.

Your menu can contain as many menu items as you wish. However, only the first 31 menu items can be individually disabled (all menu items past 31 are always enabled if the menu is enabled and always disabled if the menu is disabled). If your menu items exceed the length of the screen, the user must scroll to view the additional items. Keep in mind that the fewer the menu items in a menu, the simpler and clearer the menu is for the user.

Groups of Menu Items

The menu items in a particular menu should be logically related to the title of the menu and grouped to provide greater ease of use and understanding to the user. You should separate groups with dividers.

A menu can contain both commands that perform actions and commands that set attributes. You should use a verb or verb phrase to name commands that perform actions (for example, Cut, Copy, Paste). You should use an adjective to name commands that set attributes of a selected object (for example, Bold, Italic, Underline). You should group menu items by their type: verbs (actions) or adjectives (attributes). Create groups within each type according to the guidelines described here.

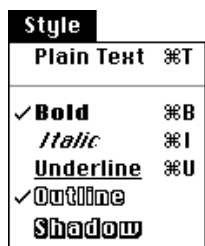
Group action commands that are logically related but independent; this makes your menus easier to read. For example, the Cut, Copy, Paste, Clear, and Select All commands in the Edit menu are grouped together; the Create Publisher, Subscribe To, and Publisher Options commands are grouped together; and the Show Clipboard command is set off by itself. (Figure 3-5 on page 3-13 shows these commands in the Edit menu of a typical application.)

Group attribute commands that are interdependent. You typically group a set of commands that set attributes into either a mutually exclusive group or an accumulating group.

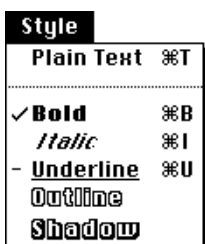
Group a set of attribute commands together if only one attribute in the group can be in effect at any one time (a *mutually exclusive* group). Place a checkmark next to the item that is currently in effect. If the user chooses a different attribute in the group, move the checkmark to the newly chosen attribute. For example, Figure 3-6 shows a Colors menu from the SurfWriter application. The colors listed in the Colors menu form a mutually exclusive group because only one color can be in effect at any one time. In this example, green is the color currently in effect. If the user chooses a different color, such as blue, the SurfWriter application uses the `SetItemMark` procedure to remove the checkmark from the Green command and to place a checkmark next to the Blue command.

Figure 3-6 Menu items in a mutually exclusive group

You can also group a set of attribute commands together if a number of the attributes in the group can be in effect at any one time (an *accumulating* group). In an accumulating group, use checkmarks to indicate that multiple attributes are in effect. In this type of group, you also need to provide a command that cancels all the other attributes. For example, a Style menu that lets the user choose any combination of font styles should also include a Plain Text command that cancels all the other attributes. Figure 3-7 shows a Style menu; in this example, the Bold and Outline attributes are both in effect.

Figure 3-7 Menu items in an accumulating group

You can also use a combination of checkmarks and dashes to help indicate the state of the user's content. For example, in a menu that reflects the state of a selection, place a checkmark next to an item if the attribute applies to the entire selection; place a dash next to an item if the attribute applies to only part of the selection. Figure 3-8 shows a Style menu that indicates that the selection contains more than one style. In this figure, the Bold attribute applies to the entire selection; the Underline attribute applies to only part of the selection.

Figure 3-8 Use of a checkmark and dash in an accumulating group

Menu Manager

Your application should adjust its menus appropriately before displaying its menus. For example, you should add checkmarks or dashes to items that are attributes as necessary, based on the state of the user's document and according to the type of window that is in the front. See "Adjusting the Menus of an Application" on page 3-73 for more information.

Another way to show the presence or absence of an attribute is to use a toggled command. Use a toggled command if the attribute has two states and you want to allow the user to move between the two states using a single menu command. For example, your application could provide a Show Borders command when the borders surrounding publishers and subscribers are not showing in a document. When the user chooses the Show Borders command, your application should show the borders and change the menu item to Hide Borders. When the user chooses the Hide Borders command, your application should hide the borders surrounding any publishers or subscribers and change the menu item to Show Borders. Use a toggled command only when the wording of the two versions of the command is not confusing to the user. Choose a verb phrase as the text of a toggled command; the text should clearly indicate the action your application performs when the user chooses the item. See "Changing the Text of an Item" on page 3-59 for further information on providing a toggled command.

Keyboard Equivalents for Menu Commands

A menu command can have a keyboard equivalent. The term **keyboard equivalent** refers to a keyboard combination, such as Command-C (⌘-C) or any other combination of the Command key, another key, and one or more modifier keys, that invokes a corresponding menu command when pressed by the user. For example, if your application supports the New command in the File menu, your application should perform the same action when the user presses Command-N as when the user chooses New from the File menu.

The term **Command-key equivalent** refers specifically to a keyboard equivalent that the user invokes by holding down the Command key and pressing another key (other than a modifier key) at the same time. This generates a keyboard event that specifies a 1-byte character that your application should pass as a parameter to the `MenuKey` function. The `MenuKey` function maps the given 1-byte character to the menu item (if any) with that Command-key equivalent.

The Menu Manager provides support for Command-key equivalents. If you define a Command-key equivalent for a menu item, the standard menu definition procedure draws the Command symbol and the specified 1-byte character to the right of the menu item's text (or to the left of the item's text if the primary line direction is right to left).

You detect a Command-key equivalent of a command by examining the `modifiers` field of the event record for a keyboard event. This allows you to determine whether the Command key was pressed at the same time as the keyboard event. If so, your application typically calls the `MenuKey` function, passing as a parameter the character code that represents the key pressed by the user. The `MenuKey` function determines if the 1-byte character matches any of the keyboard equivalents defined for your menu items; if so, `MenuKey` returns this information to your application. Your application can then

Menu Manager

perform the associated menu command, if any. See the chapter “Event Manager” in this book for additional information about the `modifiers` field of the event record.

The keyboard layout (‘KCHR’) resource of some keyboards masks or cancels the effect of the Shift key when the Command key is also pressed. For example, with a U.S. keyboard layout, when a user presses Command-S, the character code in the message field of the event record is \$73 (the character code for “s”); when a user presses Command-Shift-S, the character code in the message field of the event record is also \$73. However, not all ‘KCHR’ resources mask the Shift key in this way.


Furthermore, when your application uses the `MenuKey` function to process Command-key equivalents, `MenuKey` does not distinguish between uppercase and lowercase letters. The `MenuKey` function takes the 1-byte character passed to it and calls the `UpperText` procedure (which provides localizable uppercase conversion of the character). Thus, `MenuKey` translates any lowercase character to uppercase when comparing a keyboard event to keyboard equivalents. If your application must distinguish between lowercase and uppercase characters for keyboard equivalents, you need to provide your own method for handling such keyboard equivalents.

The key you specify for a Command-key equivalent must be a 1-byte character and is usually a letter (although you can specify 1-byte characters other than letters). For consistency and to provide greater support for localizing your application, you should always specify any letters for keyboard equivalents in uppercase when you define your application’s menu commands.

If you wish to provide other types of keyboard equivalents in addition to Command-key equivalents, your application must take additional steps to support them. If your application allows the user to hold down more than one modifier key to invoke a keyboard equivalent, your application must provide in the menu item a visual indication that represents this keyboard combination. In most cases your application must use its own method (other than `MenuKey`) for mapping the keyboard equivalent to the corresponding menu item.

If you specify a key other than a letter for a Command-key equivalent or use more than one modifier key for a keyboard equivalent, you should choose keys and keyboard combinations that can be easily localized for other regions.

If your application uses other keyboard equivalents, you can examine the state of the modifier keys and use the `KeyTranslate` function, if necessary, to help map the keyboard equivalent to a particular menu item. See the chapter “Event Manager” in this book for information on the `KeyTranslate` function, and see the discussion of ‘KCHR’ resources in *Inside Macintosh: Text* for information on how various keyboard combinations map to specific character codes.

One command that isn’t listed in a menu but can be invoked from the keyboard is the Command-period () or Cancel command. You detect a Command-period command in a method similar to the method for detecting other keyboard equivalents—you examine the `modifiers` field of a keyboard event to determine whether the Command key was pressed. In this case, however, if the user pressed the period key in addition to the Command key, rather than invoking a menu command your application should cancel the current operation.

Menu Manager

You typically define the Command-key equivalents for your application's menu commands when you define the menu commands in a 'MENU' resource. The Menu Manager displays the Command-key equivalent for a menu command (if it has one) to the right of the menu item's text (or to the left of the item's text for right-to-left script systems).

Apple reserves several keyboard equivalents for common commands. You should use these keyboard equivalents for commands in the File and Edit menus of your application.

Table 3-1 show the keyboard equivalents for standard commands.

Table 3-1 Reserved keyboard equivalents for all systems

Keys	Command	Menu
⌘-A	Select All	Edit
⌘-C	Copy	Edit
⌘-N	New	File
⌘-O	Open...	File
⌘-P	Print...	File
⌘-Q	Quit	File
⌘-S	Save	File
⌘-V	Paste	Edit
⌘-W	Close	File
⌘-X	Cut	Edit
⌘-Z	Undo	Edit

Note

You should use the keyboard equivalents Z, X, C, and V for the editing commands Undo, Cut, Copy, and Paste in order to provide support for editing in desk accessories and dialog boxes. ♦

Apple also reserves several keyboard equivalents for use with worldwide versions of system software, localized keyboards, and keyboard layouts. Table 3-2 shows these keyboard equivalents. Your application should not use the keyboard equivalents listed in Table 3-2 for its own menu commands.

See *Inside Macintosh: Text* for more discussion of handling keyboard equivalents in other script systems.

The key combinations listed in Table 3-1 and Table 3-2 are reserved across all applications. Even if your application doesn't support one of these menu commands, it shouldn't use these keyboard equivalents for another command. This guideline is for the user's benefit. Reserving these key combinations provides guaranteed, predictable behavior across all applications.

Table 3-2 Reserved keyboard equivalents for worldwide systems

Keys	Action
⌘-Space bar	Rotate through enabled script systems
⌘-Option-Space bar	Rotate through keyboard layouts or input methods within the active script system
⌘-modifier key-Space bar	Reserved
⌘-Right arrow	Change keyboard layout to the current keyboard layout of the Roman script
⌘-Left arrow	Change keyboard layout to the current keyboard layout of the system script

Table 3-3 shows other common keyboard equivalents. These keyboard equivalents are secondary to the standard keyboard equivalents listed in Table 3-1 and Table 3-2. If your application doesn't support one of the functions in Table 3-3, then you can use the equivalent as you wish.

Table 3-3 Other common keyboard equivalents

Keys	Command	Menu
⌘-B	Bold	Style
⌘-F	Find	File
⌘-G	Find Again	File
⌘-I	Italic	Style
⌘-T	Plain Text	Style
⌘-U	Underline	Style

You shouldn't assign keyboard equivalents to infrequently used menu commands. Only add keyboard equivalents for the commands that your users employ most frequently.

Menus Added Automatically by the Menu Manager

In System 7, the Menu Manager may add as many as three additional menus to your application's menu bar: the Help menu, the Keyboard menu, and the Application menu. These menus provide access to system features such as Balloon Help, keyboard layouts, and application switching. All three of these menus have icons as titles and are positioned at the right side of the menu bar. (These menus are sometimes referred to as the *system-handled menus*.)

The Menu Manager automatically inserts these additional menus in your application's current menu list when your application inserts an Apple menu into its menu bar. In this case, the Menu Manager always displays the Application menu, displays the Help menu if space is available, and displays the Keyboard menu if multiple script systems are installed and space is available. The Menu Manager also displays the Keyboard menu if the `smfShowIcon` bit is set in the flags byte of the `'itlc'` resource.

Menu Manager

The Help menu icon or both the Help menu icon and the Keyboard menu icon disappear from the menu bar if your application inserts a menu whose title extends into the space occupied by one or both of those icons. This allows your application to reclaim any space in the menu bar that would have been occupied by one or both of those two menu icons, if necessary. However, if your application inserts a menu whose title is long enough to overlap space occupied by the Application menu icon, the overlapping portion of that title disappears behind the Application menu icon. The Application menu icon is always displayed in the menu bar.

Because the Menu Manager inserts the Help, Keyboard, and Application menus into your application's current menu bar, you should not make any assumptions about the last menu (or menus) in your menu bar. Apple also reserves the right to add other system-handled menus to your application's menu bar; for compatibility you should define your menu bar such that there is room for the Help, Keyboard, and Application menus and at least one additional system-handled menu.

Your application does not need to take any action if the user chooses an item from the Keyboard or Application menu; the Menu Manager performs any appropriate actions for these two menus. If the user chooses an item that your application added to the Help menu, your application should perform the corresponding action.

The following sections describe the Help, Keyboard, and Application menus in more detail, and they also describe other menus in a typical application, including the Apple, File, and Edit menus.

The Apple Menu

You should define the Apple menu as the first menu in your application. The title of the Apple menu is the Apple icon. The Apple menu of an application typically provides an About command as the first menu item, followed by a divider, which is followed by a list of all desktop objects contained in the Apple Menu Items folder. (The phrase *desktop objects* refers to applications, desk accessories, documents, folders, and any other item that can reside in the Apple Menu Items folder.) The items following the divider in the Apple menu are listed in alphabetical order. Each item below the divider lists a desktop object and the small icon for that object.

Figure 3-9 shows the Apple menu for the SurfWriter application as it might appear on a particular user's system.

To create the items in your application's Apple menu, define the Apple menu title, the characteristics of your application's About command, and the divider following it in a 'MENU' resource.

To insert the items contained in the Apple Menu Items folder into your application's Apple menu, use the `AppendResMenu` or `InsertResMenu` procedure and specify 'DRV' as the resource type to add in the parameter `theType`. If you do this, these procedures automatically add all items in the Apple Menu Items folder in alphabetical order to the specified menu.

Figure 3-9 The Apple menu for the SurfWriter application**Note**

The Apple Menu Items folder is available in System 7 and later. In System 6, the `AppendResMenu` and `InsertResMenu` procedures add only the desk accessories in the System file to the specified menu when you specify 'DRVr' as the resource type to add in the parameter `theType`. ♦

The user can place any desktop object in the Apple Menu Items folder. When the user places an item in this folder, the system software automatically adds it to the list of items in the Apple menu of all open applications.

When the user chooses an item other than your application's About command from the Apple menu, your application should call the `OpenDeskAcc` function. The `OpenDeskAcc` function prepares to open the desktop object chosen by the user; for example, if the user chooses the Alarm Clock desk accessory, the `OpenDeskAcc` function prepares to open the Alarm Clock. The `OpenDeskAcc` function schedules the Alarm Clock desk accessory for execution and returns to your application. On your application's next call to `WaitNextEvent`, it receives a suspend event, and then the Alarm Clock desk accessory becomes the foreground process.

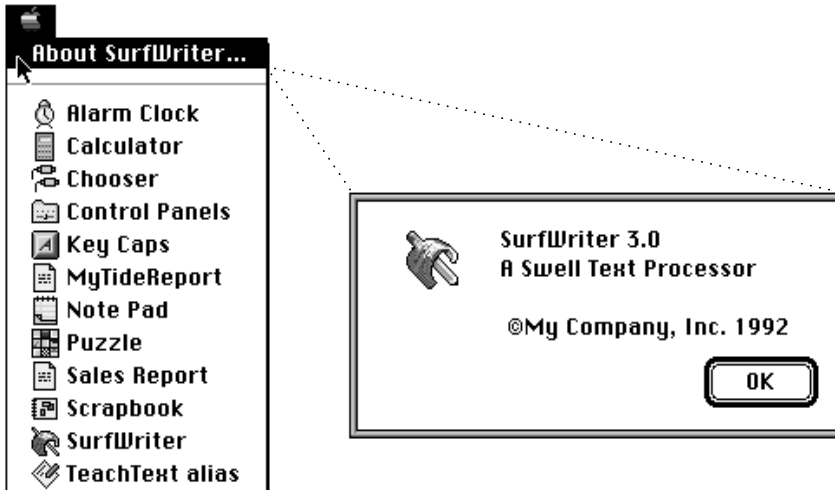
If the user chooses a desktop object other than a desk accessory or an application, the `OpenDeskAcc` function also takes the appropriate action. For example, as shown in Figure 3-9, if the user chooses a document called MyTideReport created by the SurfWriter application, the `OpenDeskAcc` function prepares to open the SurfWriter application (if it isn't already open) and schedules the SurfWriter application for execution. The SurfWriter application is instructed to open the MyTideReport document when it becomes the foreground process.

When the user chooses your application's About command, your application can display a dialog box or an alert box that contains your application's name, version number, copyright information, or other information as necessary. Your application should provide an OK button in the dialog box; the user clicks the OK button to close the dialog box.

Menu Manager

Figure 3-10 shows the alert box that the SurfWriter application displays when the user chooses the About command from the application's Apple menu.

Figure 3-10 Choosing the About command of the SurfWriter application



If your application provides any application-specific Help commands, place these in the Help menu, not the Apple menu.

The File Menu

The standard File menu contains commands related to managing documents. For example, the user can open, close, save, or print documents from this menu. The user should also be able to quit your application by choosing Quit from the File menu.

Your application should support the menu commands of the standard File menu. If you add other commands to your application's File menu, they should pertain to managing a document.

Figure 3-11 shows the standard File menu for applications.

Figure 3-11 The standard File menu for an application

File		
New		⌘N
Open...		⌘O
Close		⌘W
Save		⌘S
Save As...		
Page Setup...		
Print...		⌘P
Quit		⌘Q

Menu Manager

Table 3-4 describes the standard commands in the File menu and the actions your application should take when a user chooses them.

Table 3-4 Actions for standard File menu commands

Command	Action
New	Open a new, untitled document.
Open...	Display the Open dialog box using the Standard File Package.
Close	Close the active window (which may be a document window, modeless dialog box, or other type of window). If the active window is a document and the document has been changed since the last save, display a dialog box asking the user if the document should be saved before closing.
Save	Save the active document to a disk, including any changes made to that document since the last time it was saved. If the user chooses Save for a new untitled document (one that the user hasn't named yet), display the Save dialog box using the Standard File Package.
Save As...	Save a copy of the active document under a new name provided by the user. Display the Save dialog box using the Standard File Package. After your application saves the document, the document should remain open and active.
Page Setup...	Display the Page Setup dialog box to let the user specify printing parameters such as the paper size and printing orientation. Your application can provide other printing options as appropriate. Your application should save the user's Page Setup printing preferences for the document when the user saves the document.
Print...	Display the Print job dialog box to let the user specify various parameters, such as print quality and number of copies. Print the document if the user clicks the Print button. The options specified in the Print dialog box apply to only the current printing operation, and your application should not save these settings with the document or restore the settings when the user chooses Print again.
Quit	Quit your application after performing any necessary cleanup operations. If any open documents have been changed since the user last saved them, display the Save dialog box once for each open document that requires saving. If any background or lengthy operation is still in progress, notify the user, giving the user the option to continue and not quit the application.

See *Macintosh Human Interface Guidelines* for additional commands that you can provide in the File menu. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for information on how to perform the actions associated with the commands in the File menu. See the chapter "Standard File Package" in *Inside Macintosh: Files* for information on the standard file dialog boxes. See the chapter "Printing Manager" in *Inside Macintosh: Imaging* for information on displaying the Page Setup and Print job dialog boxes.

Menu Manager

The New, Open, Close, Save, Print, and Quit commands have the keyboard equivalents shown in Figure 3-11 on page 3-22. These keyboard equivalents are reserved for these menu commands; do not assign these keyboard equivalents to any menu command other than the ones shown in Figure 3-11.

The Edit Menu

The standard Edit menu provides commands that let users change or edit the contents of their documents. It also provides commands that allow users to share data within and between documents created by different applications using editions or the Clipboard. All Macintosh applications should support the Undo, Cut, Copy, Paste, and Clear commands. Use these commands to provide standard text-editing abilities in your application.

Figure 3-12 shows the standard Edit menu supported by Macintosh applications.

Figure 3-12 The standard Edit menu for an application

Edit	
Undo Typing	⌘Z
Cut	⌘H
Copy	⌘C
Paste	⌘V
Clear	
Select All	⌘A
Create Publisher...	
Subscribe To...	
Publisher Options...	
Show Clipboard	

The standard editing commands (Undo, Cut, Copy, Paste, and Clear) in your application's Edit menu should appear in the order shown in Figure 3-12. Whenever possible, you should add an additional word or phrase to clarify what action your application will reverse when the user chooses the Undo command. For example, Figure 3-12 shows an application's Edit menu that uses the phrase Undo Typing when typing was the last action performed by the user. If your application can't undo the last operation, you should change the text of the Undo command to Can't Undo and disable the menu item. See "Changing the Text of an Item" on page 3-59 for an example of how to change the text of a menu item.

You can include other commands in your application's Edit menu if they're related to editing or changing the content of your application's documents. If you add commands to the Edit menu, add them after the standard menu commands. For example, if appropriate, your application should support a Select All command. If your application supports both the Clear and Select All commands, they should appear in the order shown in Figure 3-12.

Table 3-5 describes the standard commands in the Edit menu and the actions your application should take when a user chooses them.

Table 3-5 Actions for standard Edit menu commands

Command	Action
Undo	Reverse the effect of the previous operation. You should add the name of the last operation to the Undo command. For example, change the item to read Undo Typing if the user just finished entering some text in a document. If your application cannot undo the previous operation, disable this menu item and change the phrase to Can't Undo.
Cut	Remove the data in the current selection, if any. Store the cut selection in the scrap (on the Clipboard). This replaces the previous contents of the scrap.
Copy	Copy the data in the current selection, if any. Copy the selection to the scrap (the Clipboard). This replaces the previous contents of the scrap.
Paste	Paste the data from the scrap at the insertion point; this replaces any current selection.
Clear	Remove the data in the highlighted selection; do not copy the data to the scrap (Clipboard).
Select All	Highlight all data in the document.
Create Publisher...	Display the Create Publisher dialog box (using the Edition Manager). Create an edition based on the selected data if the user clicks the Publish button.
Subscribe To...	Display the Subscribe To dialog box (using the Edition Manager). Allow the user to insert data from an edition if the user clicks the Subscribe button.
Publisher Options...	Display the Publisher Options dialog box (using the Edition Manager) and allow the user to set or change options associated with the publisher. Change this menu item to Subscriber Options if the current selection includes a subscriber. When the user chooses the Subscriber Options command, display the Subscriber Options dialog box.
Show Clipboard	Display the contents of the Clipboard in a window. Change this item to Hide Clipboard when the Clipboard window is showing. When the user chooses Hide Clipboard, hide the window displaying the Clipboard contents and change the menu item to Show Clipboard.

The Undo, Cut, Copy, Paste, and Select All commands have the keyboard equivalents shown in Figure 3-12 on page 3-24. These keyboard equivalents are reserved for these menu commands; do not assign these keyboard equivalents to any menu command other than the ones shown in Figure 3-12. See the chapter “Scrap Manager” in *Inside Macintosh: More Macintosh Toolbox* for information on copying data to and from the scrap.

Menu Manager

See the chapter “Edition Manager” in *Inside Macintosh: Interapplication Communication* for information on supporting the Create Publisher, Subscribe To, and Publisher Options commands in your application.

The Font Menu

You can provide a Font menu to allow the user to choose text fonts. A font is a complete set of characters created in one typeface and font style. The characters in a font can appear in many different point sizes, but all have the same design elements.

You should list the names of all currently available fonts in your application’s Font menu. The currently available fonts are those fonts residing in the Fonts folder of the user’s System Folder (or in earlier versions of system software, in the user’s System file).

You add fonts to the Font menu using the `AppendResMenu` or `InsertResMenu` procedure. These two procedures add items to the specified menu in alphabetical order.

The user can install a large number of fonts and thereby create a very large Font menu. Therefore, you should never include other items in the Font menu. Use separate menus to accommodate lists of attributes such as style and size choices. You can also provide a Size menu to allow the user to choose a specific point size of a font; the next section describes the Size menu.

Figure 3-13 shows a typical Font menu. Your application should indicate which typeface is in use by adding a checkmark to the left of the name of the current font. In Figure 3-13, the application has placed a checkmark next to Palatino to indicate that Palatino® is the current font. When the user starts entering text at the insertion point, your application should display text in the current font.

Figure 3-13 A typical Font menu



In the Font menu, you can use dashes to indicate that the selection contains more than one font. (Place a checkmark next to an item if the entire selection contains only one font.) If the current selection contains more than one font, place a dash next to the name of each font that the selection contains. See “Changing the Mark of Menu Items” on page 3-61 for information on adding dashes and checkmarks to a menu item.

Menu Manager

Figure 3-14 shows the use of dashes to indicate that a selection contains more than one font. In this figure, part of the selection contains a Helvetica® font and part of the selection contains a Palatino font.

Figure 3-14 A Font menu showing a selection containing more than one font



The `AppendResMenu` and `InsertResMenu` procedures can recognize when an added font resource is associated with a script other than the current system script (non-Roman fonts have font numbers greater than \$4000). The Menu Manager displays a font name in its corresponding script if the script system for that font is installed.

You can choose to provide a Size menu and a Style menu in addition to a Font menu. If you do so, these three menus typically appear in the order Font, Size, Style in most applications.

The Size Menu

Your application can provide a Size menu to allow the user to choose sizes for fonts. Font sizes are measured in points. A point is a typographical unit of measure equivalent (on Macintosh computers) to 1/72 of an inch.

Your application should indicate the current point size by adding a checkmark to the menu item of the current size. You can use dashes if the selection contains more than one point size.

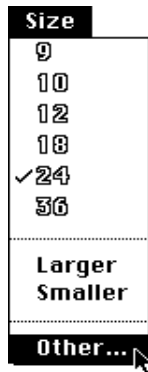
System 7 supports both bitmapped and TrueType fonts. TrueType fonts can be displayed in a wider range of point sizes, for example, 12 points, 51 points, 156 points, 578 points, or greater. Your application should not provide an upper limit for font sizes.

In the Size menu, your application should outline font sizes to indicate which sizes are directly provided by the current font. If the user chooses a TrueType font, outline all sizes of that font in the Size menu. If the user chooses a bitmapped font, outline only those sizes that appear in the Fonts folder. Use plain type for all other font sizes. See the chapter “Font Manager” in *Inside Macintosh: Text* for additional information on supporting fonts in your application.

Menu Manager

Figure 3-15 shows a typical Size menu of an application.

Figure 3-15 A typical Size menu



Your application should also provide a method that allows users to choose any point size. You can add an Other command to the end of the Size menu for this purpose. When the user chooses this command, display a dialog box that allows the user to choose any available font size. You can include an editable text item in which the user can type the desired font size. Figure 3-16 shows a dialog box an application might display when the user chooses the Other command from the Size menu.

Figure 3-16 A dialog box to select a new point size for a font

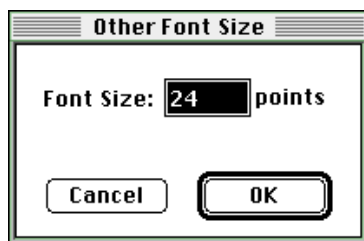
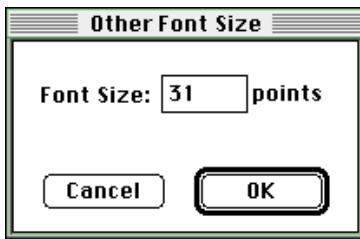
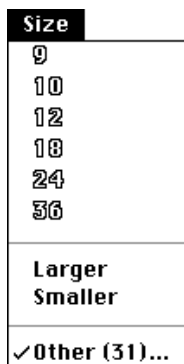


Figure 3-17 shows the Other dialog box after the user has entered a new font size of 31.

Figure 3-17 Entering a new point size for a font

If the user enters a font size not currently in the menu, your application should add a checkmark to the Other menu command and include the font size as part of the text of the Other command. You should show the font size in parentheses after the text Other, as shown in Figure 3-18.

Figure 3-18 The Other command with a font size added to it

If a selection contains more than one nonstandard size, you should include the text Mixed in parentheses following the word Other. In this case leave the editable text field of the Other dialog box blank when the user chooses the Other (Mixed) command.

See “Handling a Size Menu” on page 3-82 for more information on how to respond to the user’s choice of a command from the Size menu. See the chapter “Dialog Manager” for information on creating a dialog box.

The Help Menu

The Help menu is specific to each application, just as the Apple, File, and Edit menus are. The Help menu items defined by the Help Manager are common to all applications and give the user access to Balloon Help.

You can add menu items to your application’s Help menu to give your users access to any online help that your application supplies in addition to help balloons. If you currently provide your users with help information when they choose the About

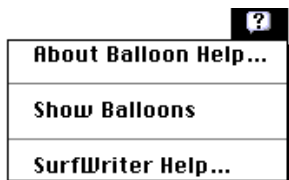
Menu Manager

command from the Apple menu, you should instead append a command for your own help to the Help menu. This gives users one consistent place to obtain help information.

When adding your own items to the Help menu, include the name of your application in the command so that users can easily determine which application the help relates to.

Figure 3-19 shows the Help menu for the SurfWriter application. This application appends one item to the end of the standard Help menu: SurfWriter Help. When the user chooses this item, the application provides access to any application-specific help information.

Figure 3-19 The Help menu of the SurfWriter application



You add items to the Help menu by using the `HMGetHelpMenuHandle` function and the `AppendMenu` procedure. Apple reserves the right to change the number of standard items in the Help menu. You should always append any additional items to the end. See “Adding Items to the Help Menu” on page 3-67 for specific examples.

The user turns Balloon Help on or off by choosing Show Balloons or Hide Balloons from the Help menu. The Help Manager automatically enables or disables Balloon Help when the user chooses Show Balloons or Hide Balloons from the Help menu. The setting of help is global and affects all applications.

When the user turns on Balloon Help, the Help Manager displays small help balloons as the user moves the cursor over areas such as scroll bars, buttons, menus, or rectangular areas in windows or dialog boxes that have help information associated with them.

Help balloons are rounded-rectangle windows that contain explanatory information for the user.

The Help Manager provides help balloons for the menu titles of the Apple, Help, Application, and Keyboard menus. The Help Manager also provides help balloons for menu items in the Application and Keyboard menus, for any item from the Apple Menu Items folder in the Apple menu, and for the standard items in the Help menu. The Help Manager provides these help balloons only if your application uses the standard menu definition procedure.

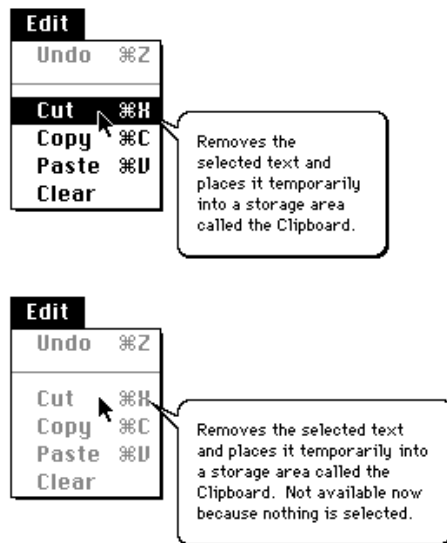
Your application should provide the content of help balloons for all other menu items and menus in your application.

Figure 3-20 shows the default help balloons for the Apple menu title and Application menu title.

Menu Manager

Figure 3-20 Default help balloons for the Apple menu and Application menu

Figure 3-21 shows help balloons for an application's Cut command when it is enabled and when it is disabled.

Figure 3-21 Help balloons for different states of the Cut command

Your application can provide the content for help balloons for your menus and menu items. You define the help balloons for your application using 'hmmu' resources.

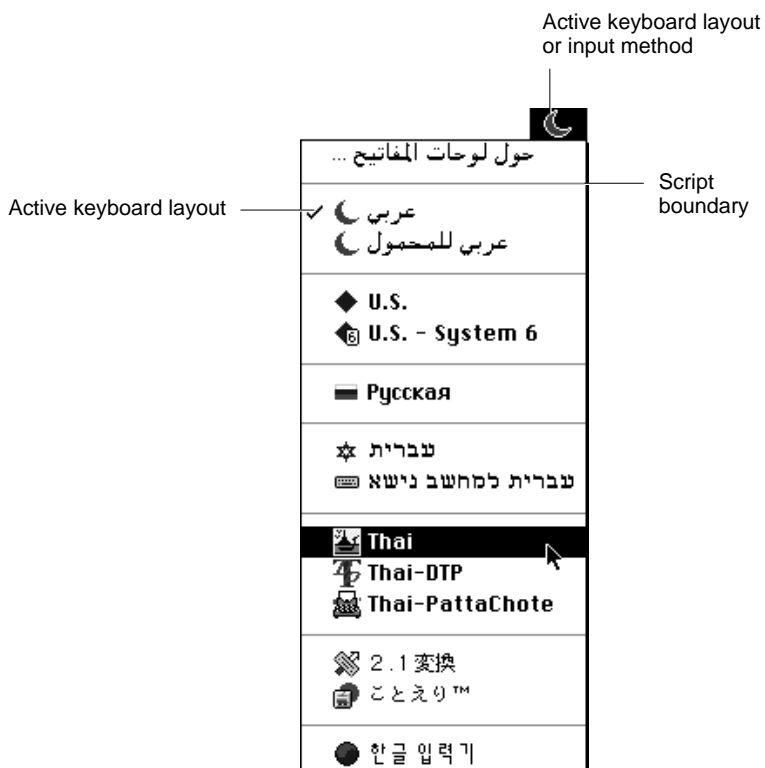
For information on how to define the help balloons for your application's menus in 'hmmu' resources, see the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox*.

The Keyboard Menu

The Keyboard menu displays a list of all the keyboard layouts and input methods that are available for each enabled script system. Each script system has at least one keyboard layout or input method associated with it. If only the Roman script system and the U.S. keyboard layout are available, the Menu Manager does not add the Keyboard menu (unless the `smfShowIcon` bit is set in the flags byte of the `'itlc'` resource). If the user's system includes an additional script system or includes additional keyboard layouts for the Roman script system and the `smfShowIcon` bit is set in the `'itlc'` resource, the Menu Manager adds the Keyboard menu to your application's menu bar as long as your application's menu bar includes an Apple menu. The Menu Manager adds the Keyboard menu to the right of the Help menu and to the left of the Application menu.

Figure 3-22 shows a Keyboard menu as it might appear on a particular user's system. System software groups the items in the Keyboard menu by their script systems. For example, in Figure 3-22 seven script systems are shown: Arabic, Roman, Cyrillic, Hebrew, Thai, Japanese, and Korean. Two keyboard layouts are available in the user's system for the Arabic script system, two keyboard layouts for the Roman script system, one keyboard layout for the Cyrillic script system, two keyboard layouts for the Hebrew script system, three keyboard layouts for the Thai script system, two input methods for the Japanese script system, and one input method for the Korean script system.

Figure 3-22 Accessing the Keyboard menu from an application



Menu Manager

When the user chooses an item from the Keyboard menu, the Menu Manager handles it appropriately. For example, if the user chooses a different keyboard layout in a different script, the Menu Manager changes the current keyboard layout and script system to the item chosen by the user. See *Inside Macintosh: Text* for further information on supporting text and handling text in multiple scripts in your application.

The Application Menu

The Application menu is the menu farthest to the right in the menu bar; the Application menu contains the icon of the active application or desk accessory for its menu title.

The Menu Manager automatically appends the Application menu to your application's menu bar if your menu bar includes an Apple menu.

When the user chooses an item from the Application menu, the Menu Manager handles the event as appropriate. For example, if the user chooses the Hide Others command, the Menu Manager hides the windows of all other open applications. If the user chooses another application from the Application menu, the Menu Manager sends your application a suspend event. Your application receives the suspend event the next time it calls `WaitNextEvent`, and your application is switched out after handling the suspend event. (See the chapter "Event Manager" in this book for information about responding to suspend and resume events.)

Figure 3-23 shows the Application menu for the SurfWriter application as it appears when both SurfWriter and TeachText are open and the user is currently interacting with SurfWriter. The checkmark next to the menu item showing SurfWriter's icon indicates that SurfWriter is the active application.

Figure 3-23 SurfWriter's Application menu



Pop-Up Menus

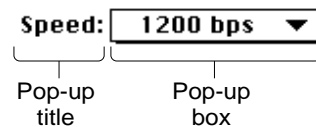
You can use pop-up menus to present the user with a list of choices in a dialog box or window. Pop-up menus are especially useful in dialog boxes that require the user to select one choice from a list of many or to set a specific value.

In System 7, the standard pop-up menu is implemented by a control definition function. This section explains how the standard pop-up control definition function provides support for pop-up menus. The chapter "Control Manager" in this book explains controls in detail.

Menu Manager

A pop-up menu appears as a rectangle with a one-pixel border and a one-pixel drop shadow. Pop-up menus are identified by a downward-pointing triangle that appears in the pop-up box. The title of the pop-up menu appears next to the pop-up box. Figure 3-24 shows a pop-up menu.

Figure 3-24 A pop-up menu



To display a pop-up menu, the user presses the mouse button while the cursor is over the pop-up title or pop-up box. If the pop-up menu is in a dialog box and your application uses the Dialog Manager, the Dialog Manager uses the pop-up control definition function to display the pop-up menu and to handle all user interaction in the pop-up menu. If the pop-up menu is in one of your application's windows, your application needs to determine which control the cursor was in when the user pressed the mouse button. Your application can then use the Control Manager routines to display the pop-up menu and to handle user interaction in the control.

Just like `MenuSelect`, the pop-up control definition function highlights the pop-up menu title and highlights menu items appropriately as the user drags the cursor through the menu items. The pop-up control definition function also highlights the default (current) menu item when the pop-up menu is first displayed and adds the checkmark to the menu item. Once the user releases the mouse button, the pop-up control definition function causes the chosen item (if any) to blink, unhighlights the menu title, changes the text in the pop-up box, and stores the item number of the chosen item as the value of the control. Your application can use the Control Manager function `GetControlValue` to get the menu item chosen by the user.

Figure 3-25 shows a pop-up menu in its closed state (as it appears initially to the user) and its open state (as it appears when the user presses the mouse button while the cursor is in the pop-up menu).

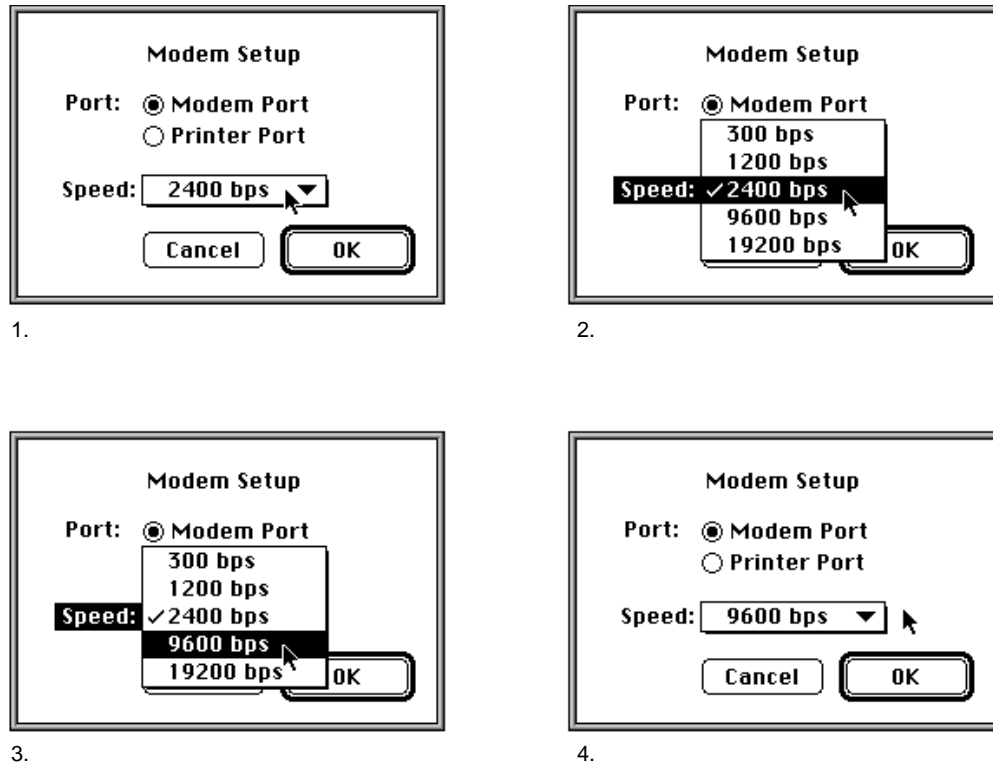
Figure 3-25 A pop-up menu in its closed and open states



If you don't provide a title for a pop-up menu, the current menu item serves as the title. In most cases you should create pop-up menus that have titles. Choose a title that reflects the contents of the menu or indicates the purpose of the menu.

Figure 3-26 shows the process of a user making a selection from a pop-up menu.

Figure 3-26 Making a selection from a pop-up menu



In step 1 in Figure 3-26, the user presses the mouse button while the cursor is over the pop-up box. When this occurs, your application can use the Dialog Manager or Control Manager to call the pop-up control definition function. In step 2, the pop-up control definition function highlights the title of the pop-up menu, removes the downward-pointing triangle from the pop-up box, adds a checkmark to the current item, highlights the current item, and displays the contents of the pop-up menu. In step 3, the pop-up control definition function handles all user interaction, highlighting and unhighlighting menu items, until the user releases the mouse button. When the user releases the mouse button, the pop-up control definition function closes the pop-up menu, unhighlights the pop-up menu title, sets the text of the pop-up box to the item chosen by the user, and stores the item number of the chosen item as the value of the control. Step 4 shows the appearance of the closed pop-up menu after the pop-up control definition function performs these actions.

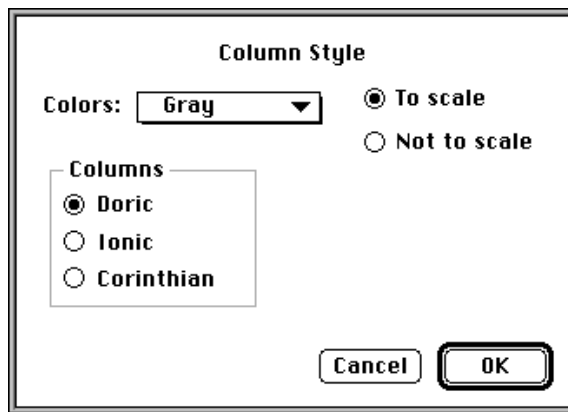
If your application does not use the standard pop-up control definition function, you can create your own control definition function and you can choose to use the `PopupMenuSelect` function to help your application handle pop-up menus. In this case, when the user presses the mouse button when the cursor is in a pop-up menu, your application should call the `PopupMenuSelect` function. Your application must

Menu Manager

highlight the pop-up title before calling `PopUpMenuSelect` and unhighlight it afterward. The `PopUpMenuSelect` function displays the pop-up menu and highlights menu items appropriately as the user drags the cursor through the menu items. Once the user releases the mouse button, `PopUpMenuSelect` flashes the chosen item, if any, and returns information indicating which menu item was chosen to your application. Your application is responsible for highlighting and unhighlighting the menu title, updating the text in the pop-up box, and storing any changes to the settings of the menu items if you use the `PopUpMenuSelect` function.

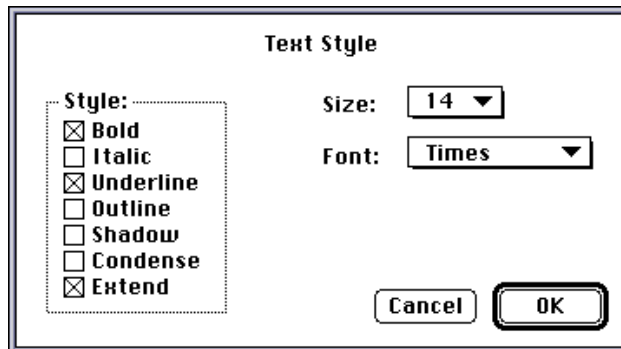
Pop-up menus work well when your application needs to present several choices to the user. Note that pop-up menus hide these choices from the user until the user chooses to display the pop-up menu. Use pop-up menus when the user doesn't need to see all the choices all the time. For example, Figure 3-27 shows a dialog box that uses a pop-up menu to allow the user to choose one color from a list of many.

Figure 3-27 Choosing one attribute from a list of many



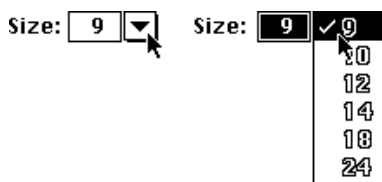
If you need to show only a few choices, you may find that using checkboxes or radio buttons is more appropriate for your application. For example, in Figure 3-27 the selection of columns is implemented with radio buttons rather than a pop-up menu. Whenever possible, you should show all available choices to the user. Note that in this example the amount of space occupied by the radio buttons is about the same as the amount of space required for a corresponding pop-up menu.

Use pop-up menus to allow the user to choose one option from a set of many choices. Don't use a pop-up menu for multiple-choice lists where the user can make more than one selection. If you do, the text in the menu box will not fully describe the selections in effect. For example, don't use a pop-up menu for font style selections. In a dialog box, font style selections are more appropriately implemented as checkboxes. Figure 3-28 shows a dialog box that uses checkboxes instead of a pop-up menu to allow the user to select more than one font style. The Size and Font choices are implemented as pop-up menus in this example, since the user can choose only one size and one font from a list of many.

Figure 3-28 A dialog box with checkboxes and pop-up menus

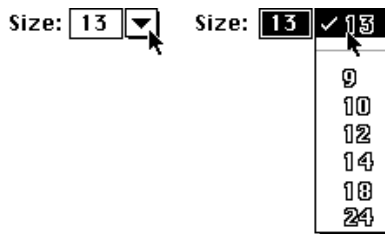
Never use a pop-up menu as a way to provide more commands. Pop-up menus should not contain actions (verbs) but can contain attributes (adjectives) or settings that allow the user to choose one from many. For these reasons, you should not use Command-key equivalents for pop-up menu items.

Your application can also use type-in pop-up menus when appropriate. Use a type-in pop-up menu to give the user a list of choices and to allow the user to type in an additional choice. The standard pop-up control definition function that implements pop-up menus does not provide specific support for type-in menus. You can create your own control definition function to handle type-in pop-up menus. If you do so, your type-in pop-up menu should adhere to the guidelines described here. Figure 3-29 shows a typical type-in pop-up menu in its closed and open states.

Figure 3-29 A type-in pop-up menu in its closed and open states

Your application is responsible for drawing and highlighting the type-in field of the pop-up menu. Your application does not need to highlight the title of a type-in pop-up menu; your application should highlight the type-in field instead.

If the user types in a value that is already in the menu, make that item the current item. If the user types a value that does not match any of the items in the pop-up menu, add the item to the top of the menu and add a divider below the item to separate it from the rest of the standard items. Figure 3-30 on the next page shows a type-in pop-up menu with a user's choice added to it.

Figure 3-30 A type-in pop-up menu with a user's choice added

A type-in pop-up menu should allow the user to type in a single additional choice. That is, a standard type-in pop-up menu does not accumulate the user's choices in the menu. For example, if the user types in a value of 13, then types in a new choice, such as 43, the menu should appear as shown in Figure 3-30, except that the type-in field and menu item that previously contained 13 is replaced by 43.

A type-in pop-up menu should also allow the user to type in any of the standard values in the menu or choose any of the standard items in the pop-up menu. If the user types in or chooses any of the standard items, you should remove any user-specified item previously added to the menu. For example, as shown in Figure 3-30, the user specified a nonstandard size of 13. If the user then types in or selects 9, your application should return the pop-up menu to its standard state, as shown in Figure 3-29 on page 3-37.

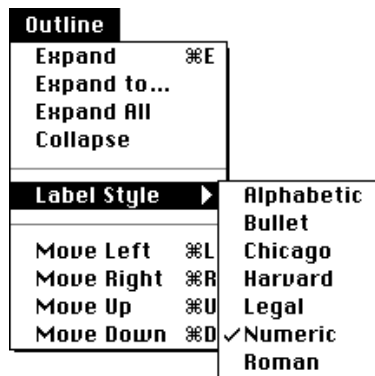
Hierarchical Menus

A hierarchical menu is a menu that has a submenu attached to it. Hierarchical menus can be useful when your application needs to offer additional choices to the user without taking up extra space in the menu bar. If you use a hierarchical menu in your application, use it to give the user additional choices or to choose attributes, not to choose additional commands.

In a hierarchical menu, a menu item serves as the title of a submenu; this menu item contains a triangle to identify that the item has a submenu. The triangle appears in the location of the keyboard equivalent. The title of a submenu should represent the choices it contains. Figure 3-31 shows a menu with a submenu whose menu title is Label Style.

When a user drags the cursor through a hierarchical menu and rests the cursor on a menu item that has a submenu, the Menu Manager displays the submenu after a brief delay. The title of the submenu remains highlighted while the user browses through the submenu; the Menu Manager unhighlights the menu title of the submenu when the user releases the mouse button.

Hierarchical menus are useful for providing lists of related items, such as font sizes and font styles. Never use more than one level of hierarchical menus (in other words, don't attach a submenu to another submenu). You can assign keyboard equivalents to the menu items of a submenu; however, if you do so, you make it harder for the user to quickly scan all menus for their keyboard equivalents.

Figure 3-31 A hierarchical menu item and its submenu

About the Menu Manager

The Menu Manager, together with the menu definition procedure and menu bar definition function, provides your application with a convenient way to manage the menus in your application. The Menu Manager uses two data structures, menu records and menu lists, to manage menus. The next two sections describe how the Menu Manager uses these two data structures. “Using the Menu Manager,” which begins on page 3-41, shows how you can use the Menu Manager to

- define a menu using a 'MENU' resource
- define a menu bar using an 'MBAR' resource
- install your application's menu bar
- change the appearance of menu items
- add menu items to a menu
- respond to the user when the user chooses a menu item
- handle the Apple and Help menus
- create a pop-up menu
- create a hierarchical menu
- handle access to menus when your application displays a dialog box
- write your own menu definition procedure

How the Menu Manager Maintains Information About Menus

The Menu Manager maintains information about menus in menu records. Each menu record includes certain information about a specific menu, including

- the menu ID of the menu
- the horizontal and vertical dimensions of the menu (in pixels)
- a handle to the menu definition procedure of the menu
- flags indicating whether each item (for the first 31 items) is enabled or disabled and whether the menu title is enabled or disabled
- the contents of the menu, including the menu title and other data that defines the menu items

You typically specify most of this information in a menu resource, that is, a resource of type 'MENU'. When you create a menu, the Menu Manager stores this information in a menu record. A **menu record** is a data structure of type `MenuInfo`. You usually never need to access the information in the menu record directly; the Menu Manager automatically updates the menu record when you make any changes to the menu, such as adding a menu item. See “The Menu Record” beginning on page 3-95 if you need to access the fields of the menu record directly.

The Menu Manager identifies every menu by a number referred to as a **menu ID**. You must assign a menu ID to each menu in your application. Each menu in your application must have a menu ID that is unique from that of any other menu in your application. You can use any number greater than 0 for a menu ID of a pull-down or pop-up menu; submenus of an application can use only menu IDs from 1 through 235; submenus of a desk accessory must use menu IDs from 236 through 255.

When you create a menu, the Menu Manager creates a menu record for the menu and returns a handle to that menu record. To refer to a menu, you usually use either the menu’s menu ID or a handle to the menu’s menu record.

To refer to a menu item, use the menu item’s **item number**. Item numbers identify items in menus; items are assigned item numbers starting with 1 for the first menu item in the menu, 2 for the second menu item in the menu, and so on, up to the number of the last menu item in the menu.

How the Menu Manager Maintains Information About an Application’s Menu Bar

A **menu list** contains handles to the menu records of one or more menus (although a menu list can be empty). The end of a menu list can contain handles to the menu records of submenus and pop-up menus; the phrase *submenu portion of the menu list* refers to this portion of the menu list, which contains information about submenus and pop-up menus.

When your application initializes the Menu Manager, the Menu Manager allocates the current menu list, which is initially empty. The contents of the current menu list change as your application adds menus to or removes menus from it.

Menu Manager

The **current menu list** contains handles to the menu records of all menus in the current menu bar and the menu records of any submenus or pop-up menus that you have inserted into the current menu list. Your application typically creates a menu list using `GetNewMBar`, and it then sets the current menu list to its newly created menu list using `SetMenuBar`. You can insert other menus in the current menu list using the `GetMenu` function and `InsertMenu` procedure.

The Menu Manager displays the menu bar and the titles of all pull-down menus that are defined in the current menu list when your application calls the `DrawMenuBar` procedure. The Menu Manager displays the menus in the menu bar in the same order that they appear in the current menu list.

The Menu Manager provides routines for adding menus to and removing menus from the current menu list; your application should never access a menu list directly. To refer to a menu list, use the handle returned by `GetNewMBar` or `GetMenuBar`.

The Menu Manager inserts the Help menu, the Keyboard menu if necessary, and the Application menu into your application's menu list if your application calls the `GetNewMBar` function and your menu bar includes an Apple menu; your application then uses `SetMenuBar` to set the current menu list to the newly created menu list. The Menu Manager also inserts these menus into your application's current menu list if your application inserts the Apple menu into the current menu list using the `InsertMenu` procedure. Therefore, you should not make any assumptions about the last menu (or menus) in your application's current menu list.

When your application inserts a submenu into the current menu list, the Menu Manager stores a handle to the menu record of the submenu in the submenu portion of the current menu list. Similarly, when your application inserts a pop-up menu into the current menu list, the Menu Manager stores a handle to the menu record of the pop-up menu in the submenu portion of the current menu list.

Using the Menu Manager

You can define your application's menus and menu bar as resources and use Menu Manager routines to create and manage them. For example, whenever the user presses the mouse button while the cursor is in the menu bar, your application should call the `MenuSelect` function, allowing the user to choose a command from any menu. The `MenuSelect` function handles all user activity until the user releases the mouse button. The `MenuSelect` function displays and removes menus as the user drags the cursor through the menu bar, and it highlights enabled menu items as the user drags through a menu.

You should provide help balloons for each menu title and menu item of your application. You store information and text for help balloons in resources. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for complete and specific information on how to provide help balloons for the menus of your application. The BalloonWriter application, available from APDA, can also help you create help balloons for the menus of your application.

Menu Manager

Your application needs to initialize QuickDraw, the Font Manager, and the Window Manager before using the Menu Manager. Your application can accomplish this using the `InitGraf`, `InitFonts`, and `InitWindows` procedures. To initialize the Menu Manager, use the `InitMenus` procedure.

If your application uses pop-up menus, use the `Gestalt` function with the `gestaltPopUpAttr` selector to determine if the control definition function for pop-up menus is available. See *Inside Macintosh: Operating System Utilities* for information about the `Gestalt` function.

To create the pull-down menus in your application's menu bar, you need to

- create descriptions of each pull-down menu in 'MENU' resources
- create an 'MBAR' resource that lists the order and resource ID of each menu
- use the `GetNewMBar` function and `SetMenuBar` procedure to set up your menu bar and use the `DrawMenuBar` procedure to draw your menu bar

The next section, "Creating a Menu," explains these steps in detail.

After creating your application's menu bar, you can enable or disable your menu items, add marks such as checkmarks or dashes to menu items, or add items to any of your menus as needed. See "Enabling and Disabling Menu Items" on page 3-58, "Changing the Mark of Menu Items" on page 3-61, and "Adding Items to a Menu" beginning on page 3-64 for information on these topics.

"Handling User Choice of a Menu Command," beginning on page 3-70, shows how to handle mouse-down events in the menu bar, adjust the menus of your application, and determine if the user chose a keyboard equivalent of a command.

"Responding When the User Chooses a Menu Item," beginning on page 3-78, describes how your application should respond once the user chooses an item and also shows how to handle the user's choice of a command from the Apple and Help menus.

If your application displays dialog boxes, see "Accessing Menus From a Dialog Box" beginning on page 3-84.

Finally, if your application needs to create submenus or pop-up menus, see "Creating a Hierarchical Menu" on page 3-53 and "Creating a Pop-Up Menu" on page 3-56.

Creating a Menu

You use various Menu Manager routines to set up the menus and the menu bar for your application. You can use any of these methods to create pull-down menus for your application:

- You can create descriptions of your application's menus in 'MENU' resources and describe your application's menu bar in an 'MBAR' resource. You use the `GetNewMBar` function to read in descriptions of your menu bar and menus and create a new menu list, use the `SetMenuBar` procedure to set the current menu list to your application's menu list, and use the `DrawMenuBar` procedure to update the menu bar.
- You can create descriptions of your application's menus in 'MENU' resources, read them in using `GetMenu`, add them to the current menu list using `InsertMenu`, and update the menu bar using `DrawMenuBar`.

Menu Manager

- You can use `NewMenu` to create new empty menus; use `AppendMenu`, `InsertMenuItem`, `InsertResMenu`, or `AppendResMenu` to fill the menus with menu items; add the menus to the current menu list using `InsertMenu`; and update the menu bar using `DrawMenuBar`.

Whenever possible you should define your menus in menu ('MENU') resources and your menu bar in a menu bar ('MBAR') resource to make your application easier to localize.

To create a hierarchical menu, you need to create descriptions of the submenu and the menu to which the submenu is attached. Usually you create the description of both menus in 'MENU' resources. You typically read in the description of the hierarchical menu using `GetNewMBar` (if you also provide an 'MBAR' resource). To read in the description of the submenu and insert it in the current menu list, use the `GetMenu` function and `InsertMenu` procedure.

To create a pop-up menu, create descriptions of the pop-up menu and its menu items, create a control that uses the pop-up control definition function, and associate the control with a window or dialog box. You can display and manage the pop-up menu using the Dialog Manager or Control Manager routines.

Once the Menu Manager creates a menu for your application, if necessary you can add additional menu items to the menu using `AppendMenu`, `InsertMenuItem`, `InsertResMenu`, or `AppendResMenu`. You can use various Menu Manager routines to change the appearance of menu items.

The next sections describe how to create 'MENU' and 'MBAR' resources. "Creating a Hierarchical Menu" on page 3-53 describes how to create a menu that has a submenu, and "Creating a Pop-Up Menu" on page 3-56 describes how to create pop-up menus.

Creating a Menu Resource

Usually you should define your menus in menu ('MENU') resources so that you can easily localize the menu titles and menu items for other languages, cultures, or regions. A 'MENU' resource defines the menu title of a menu and the characteristics of menu items in a menu. Listing 3-1 shows a sample 'MENU' resource in Rez format for an application's Apple menu. (Rez is a resource compiler available with MPW. You can also define menus using a resource utility, such as ResEdit, available from APDA.)

Listing 3-1 Rez input for a 'MENU' resource for the Apple menu

```
#define mApple 128

resource 'MENU' (mApple, preload) { /*resource ID, preload resource*/
    mApple,                          /*menu ID*/
    textMenuProc,                    /*uses standard menu definition */
                                     /* procedure*/
    0b1111111111111111111111111111101, /*enable About item, */
                                     /* disable divider, */
                                     /* enable all other items*/
}
```

Menu Manager

```

enabled,                /*enable menu title*/
apple,                  /*menu title*/
{
    /*first menu item*/
    "About SurfWriter...", /*text of menu item */
                           /* (includes ellipsis)*/
                           /*item characteristics follow*/
    noicon,              /*icon number (if any) or */
                           /* script code (if any)*/
    nokey,               /*keyboard equivalent (if any) */
                           /* or submenu (if any) or */
                           /* small or reduced icon (if any)*/
    nomark,              /*marking character (if any) or */
                           /* menu ID of submenu (if any)*/
    plain;               /*style of menu item text*/
    /*second menu item*/
    "-",                /*item text (divider)*/
    noicon, nokey, nomark, plain /*item characteristics*/
}
};

```

You should also define help balloons for each of your application's menu items and each menu title when you create your menus. (Figure 3-21 on page 3-31 shows help balloons for an application's Cut command.) You define the help balloons for your application's menus in 'hmmu' resources. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for examples of how to create 'hmmu' resources.

Listing 3-1 defines the resource ID of the Apple menu as 128. You can use any number equal to or greater than 128 as a resource ID for a menu. By convention, many applications use 128 as the resource ID of the first menu in the application's menu bar (the Apple menu) and use sequential numbers for the resource IDs of following menus.

Listing 3-1 also defines the menu ID of the Apple menu as 128. Once your application creates the menu, the Menu Manager uses the defined menu ID to refer to this menu. The number you define for the menu ID of a menu does not have to match the resource ID of the menu, but it is usually more convenient to use the same number. You can use any number greater than 0 for the menu ID of a pull-down or pop-up menu; submenus of an application can only use menu IDs from 1 through 235; submenus of a desk accessory must use menu IDs from 236 through 255.

The listing specifies that this menu uses the standard menu definition procedure. If you choose to create your own menu definition procedure, list its resource ID instead of the `textMenuProc` constant.

Menu Manager

After the resource ID of the menu definition procedure is a 32-bit number (expressed as a 31-bit field followed by a `Boolean` field), where bits 1–31 indicate if the corresponding menu item is disabled or enabled, and bit 0 indicates whether the menu is enabled or disabled.

The listing specifies in the 31-bit field that the first menu item should be enabled, that the second menu item should be disabled, and that the following menu items (item numbers 3 through 31) should be enabled when the menu is first created. After creating a menu, your application can enable or disable menu items using the `EnableItem` or `DisableItem` procedure. If a menu contains more than 31 items, the Menu Manager automatically enables all items following the 31st item when the menu is enabled. Your application cannot disable any individual items following the 31st item. However, you can disable all items, including items after the 31st item, by disabling the entire menu.

Listing 3-1 specifies that the menu title should be enabled when it is first created. Your application can also disable or enable the menu title using the `DisableItem` or `EnableItem` procedure. When you disable a menu using the `DisableItem` procedure, the Menu Manager disables all menu items in the menu (including any items following the 31st item) and dims the title of the menu.

The resource listing identifies the title of the menu using the constant `apple`. If you specify the `apple` constant as the title, the Menu Manager uses a small Apple icon as the title of the menu. The Menu Manager uses a color Apple icon if the monitor is set to display colors. The listing then defines the characteristics of each menu item in the menu. For each menu item, you need to define the text and any other characteristics of the menu item. For example, Listing 3-1 defines the first item in the Apple menu as the About command; note that the text of this menu item specifies three ellipsis points (...). Specify three ellipsis points following the text of a menu command if your application displays a dialog box requesting information from the user before performing the command. In general, you should not use ellipses if your application displays a confirmation alert after the user chooses a menu command; the About command is an exception to this guideline.

Listing 3-1 defines other characteristics of the About command—it doesn't have an icon to the left of the menu item text, it doesn't have a keyboard equivalent, it doesn't have any mark to the left of the menu item text, and the font style of the menu item text is plain.

By specifying various combinations of values in the icon field and keyboard equivalent field, you can define an icon (normal, small, reduced, or color), a keyboard equivalent, a submenu, or the script code of a menu item. Note that some characteristics are mutually exclusive (for example, an item can have a keyboard equivalent or submenu, but not both), as described in the following paragraphs. Table 3-6 on page 3-46 summarizes how the Menu Manager interprets these item characteristics.

Menu Manager

Table 3-6 Specifying submenus, script codes, reduced icons, small icons, and color icons of a menu item in a menu resource

Keyboard equivalent field	Icon field	Marking character field	Description
\$1B		Menu ID of submenu	Indicates the item has a submenu. The marking character field specifies the menu ID of the submenu.
\$1C	Script code of item text		Indicates the item text uses the script defined by the script code specified in the icon field.
\$1D	Icon number of 'ICON' resource		Indicates the item has an icon defined by an 'ICON' resource and that it should be reduced to fit in a 16-by-16 bit rectangle.
\$1E	Icon number of 'SICN' resource		Indicates the item has an icon defined by an 'SICN' resource.
\$00 or >\$20	Icon number of 'ICON' or 'cicn' resource		Indicates the item has an icon defined by an 'ICON' or a 'cicn' resource. (A value greater than \$20 in the keyboard equivalent field specifies the item's keyboard equivalent.)

To assign an icon to a menu item, specify an icon number in place of the `noicon` constant. The icon number you specify should be a number from 1 through 255 (or from 1 through 254 for small icons and reduced icons); add 256 to your icon number and use the result for the resource ID of the color icon ('`cicn`') resource, icon ('`ICON`') resource, or small icon ('`SICN`') resource that describes the icons for the menu item. You must define the icon for a menu item in a '`cicn`', an '`ICON`', or an '`SICN`' resource; the Menu Manager uses only these types of resources for icons you define for your menu items. The Menu Manager first looks for a '`cicn`' resource with the calculated resource ID and uses that icon if it finds it. If it doesn't find a '`cicn`' resource (or if the computer doesn't have Color QuickDraw) and the keyboard equivalent field specifies \$1E, the Menu Manager looks for an '`SICN`' resource with the calculated resource ID. Otherwise, the Menu Manager looks for an '`ICON`' resource and plots it in a 32-by-32 bit rectangle, unless the keyboard equivalent field contains \$1D. If the keyboard equivalent field contains \$1D, the Menu Manager reduces the icon to fit in a 16-by-16 bit rectangle.

If you provide an '`ICON`' resource and specify the `nokey` constant or a value greater than \$20 as the keyboard equivalent, the Menu Manager enlarges the rectangle of the entire menu item to fit the 32-by-32 bit '`ICON`' resource. If you specify a value of \$1D as the keyboard equivalent of the menu item, the Menu Manager reduces the '`ICON`' resource to fit in a 16-by-16 bit rectangle. If you provide an '`SICN`' resource and specify a value of \$1E as the keyboard equivalent of a menu item, the Menu Manager plots the small icon in a 16-by-16 bit rectangle. If you provide a '`cicn`' resource, the Menu Manager automatically enlarges the enclosing rectangle of the menu item according to the rectangle specified in the '`cicn`' resource. (For the Apple and Application menus,

Menu Manager

the Menu Manager automatically reduces the icon to fit within the enclosing rectangle of a menu item or uses the appropriate icon from the application's icon family, such as an 'ics8' resource, if one is available.) See the chapter "Finder Interface" in this book for details on how to create icons for your application.

To assign a keyboard equivalent to a menu item, specify the 1-byte character that the user types in addition to the Command key in place of the `nokey` constant in your resource definition for the menu item. If your application attaches a submenu to a menu item, then specify the `hierarchicalMenu` constant in place of the `nokey` constant. A menu item can have either a keyboard equivalent or submenu defined for it, but not both. To indicate that a menu item has an icon that is defined in an 'SICN' resource, specify `$1E` in place of the `nokey` constant. To indicate that a menu item has an icon that is defined in an 'ICON' resource and that the Menu Manager should reduce this icon to a 16-by-16 bit rectangle, specify `$1D` in place of the `nokey` constant. Menu items that have small icons or reduced icons cannot have keyboard equivalents.

To set the script code of a menu item's text, specify `$1C` in place of the `nokey` constant and define the desired script code in place of the `noicon` constant. If an item contains `$1C` in its keyboard equivalent field and a script code in its icon field, the Menu Manager draws the item's text in the script identified by the script code value if the corresponding script system is installed. If you do not specify a script code for a menu item, the Menu Manager displays the menu item's text in the system font of the current system script. For Roman scripts, the system font is Chicago and the system font size is 12.

To assign a mark that appears to the left of the menu item text and to the left of any icon, specify the marking character in place of the `nomark` constant in your resource definition. If the menu item has a submenu, then specify the menu ID of the submenu in place of the `nomark` constant. Submenus of an application must use menu IDs from 1 through 235; submenus of a desk accessory must use menu IDs from 236 through 255. Note that defining the menu ID of a submenu in a 'MENU' resource does not attach the submenu to its menu. You must use the `GetMenu` function and `InsertMenu` procedure to do this. "Creating a Hierarchical Menu," which begins on page 3-53, gives information on attaching a submenu to its menu.

To assign a font style to a menu item, in your 'MENU' resource use the constants `bold`, `italic`, `plain`, `outline`, and `shadow` to get their corresponding styles.

Listing 3-1 defines the second menu item as a divider. When you use a hyphen as the first character in the string that defines the text of a menu item, the Menu Manager creates a divider that extends across the entire width of the menu item. You cannot assign any other characteristics to a divider.

The 'MENU' resource for the Apple menu does not list any other menu items. Use the `AppendResMenu` procedure to add the desktop items to the Apple menu after your application creates the menu. See "Adding Items to the Apple Menu" on page 3-68 for more information.

Once you create a menu, you can append additional items to it using the `AppendMenu`, `InsertMenuItem`, `InsertResMenu`, or `AppendResMenu` procedure. You can also change the characteristics of individual menu items using Menu Manager routines. See "Changing the Appearance of Items in a Menu" on page 3-57 for more information.

Menu Manager

Figure 3-12 on page 3-24 shows a typical Edit menu for an application. Listing 3-2 shows a 'MENU' resource for this Edit menu.

Listing 3-2 Rez input for a 'MENU' resource for an Edit menu

```
#define mEdit 130
resource 'MENU' (mEdit, preload) {      /*resource ID, preload resource*/
    mEdit,                               /*menu ID*/
    textMenuProc,                       /*uses standard menu definition */
                                         /* procedure*/
    0b00000000000000000000000000000000100100000000000, /*enable/disable first 31 menu */
                                         /* items as appropriate*/
    enabled,                             /*enable title*/
    "Edit",                              /*text of menu title*/
    {                                    /*menu items*/
        "Undo",    noicon, "Z",    nomark, plain; /*keyboard equivalent Command-Z*/
        "-",       noicon, nokey,  nomark, plain;
        "Cut",     noicon, "X",    nomark, plain; /*keyboard equivalent Command-X*/
        "Copy",    noicon, "C",    nomark, plain; /*keyboard equivalent Command-C*/
        "Paste",   noicon, "V",    nomark, plain; /*keyboard equivalent Command-V*/
        "Clear",   noicon, nokey,  nomark, plain;
        "Select All",
            noicon, "A",    nomark, plain; /*keyboard equivalent Command-A*/
        "-",       noicon, nokey,  nomark, plain;
        "Create Publisher...",
            noicon, nokey,  nomark, plain;
        "Subscribe To...",
            noicon, nokey,  nomark, plain;
        "Publisher Options...",
            noicon, nokey,  nomark, plain;
        "-",       noicon, nokey,  nomark, plain;
        "Show Clipboard",
            noicon, nokey,  nomark, plain
    }
};
```

Listing 3-2 defines the resource ID of the Edit menu as 130, defines the menu ID of the Edit menu as 130, and specifies that this menu uses the standard menu definition procedure. The listing defines the initial enabled state of the first 31 menu items and also specifies that the menu title should be enabled when it is first created.

The resource listing defines the title of the menu, Edit. It then defines the characteristics of each menu item in the menu. For each menu item, you need to specify the text of the menu item and any other characteristics of the menu item. For example, Listing 3-2

Menu Manager

defines the first item in the Edit menu as the Undo command with these characteristics: there is no icon to the left of the menu item text, the menu item has a keyboard equivalent of Command-Z, it does not have any mark to the left of the menu item text, and the style of the menu item text is plain. The listing defines the second menu item as a divider line. It defines the Cut, Copy, and Paste commands; specifies keyboard equivalents for each of them; and defines the rest of the items in the menu.

Listing 3-3 shows another example of a resource description of a menu, the File menu of a typical application.

Listing 3-3 Rez input for a 'MENU' resource for a File menu

[illegible]

Creating a Menu Bar Resource

You typically define your application's menu bar using a menu bar ('MBAR') resource. Listing 3-4 shows an 'MBAR' resource, in Rez format, for a sample application.

Listing 3-4 Rez input for an 'MBAR' resource

```
#define rMenuBar      128
#define mApple        128
#define mFile         129
#define mEdit         130
```

Menu Manager

```
resource 'MBAR' (rMenuBar, preload) { /*resource ID, preload*/
    /*menus appear in the order listed here*/
    { mApple, mFile, mEdit };          /*resource IDs for menus in */
                                        /* this menu bar*/
};
```

Listing 3-4 defines the 'MBAR' resource with resource ID 128. This 'MBAR' resource defines the order and resource IDs of the menus contained in it; it defines its first three menus as the menus with resource IDs 128, 129, and 130. The Menu Manager uses the assigned resource IDs to read in the menus when it creates a menu bar from an 'MBAR' resource.

Setting Up Your Application's Menu Bar

To create a menu list as defined in an 'MBAR' resource, use the `GetNewMBar` function. For each menu defined by the 'MBAR' resource, the `GetNewMBar` function creates a menu record for the menu, creates each menu according to its resource definition in its corresponding 'MENU' resource, and inserts each menu into the new menu list. The `GetNewMBar` function returns a handle to the created menu list. For example, this code creates a menu list for the menu bar defined by the 'MBAR' resource with resource ID 128 (defined by the constant `rMenuBar`):

```
CONST
    rMenuBar = 128;
VAR
    menuBar:    Handle;

    menuBar := GetNewMBar(rMenuBar); {read menus and menu bar }
                                        { descriptions,create & return }
                                        { a handle to a new menu list}
```

Use the `SetMenuBar` procedure to set the current menu list to the menu list created by your application and the `DrawMenuBar` procedure to update the menu bar's appearance. For example, Listing 3-5 uses these two routines to set up the application's menu bar.

Listing 3-5 Setting up an application's menus and menu bar

```
PROCEDURE MyMakeMenus;
VAR
    menuBar:    Handle;
BEGIN
    {first use the GetNewMBar function to read menus in & create a }
    { new menu list. If you define an Apple menu, the Menu Manager }
    { inserts the Help and Application menus (and Keyboard menu if }
    { necessary) into the newly created menu list}
```

Menu Manager

```

menuBar := GetNewMBar(rMenuBar);
IF menuBar = NIL THEN
    EXIT(MyMakeMenus);
SetMenuBar(menuBar); {insert menus into the current menu list}
DisposHandle(menuBar);
                        {add desktop items in Apple Menu Items }
                        { folder to Apple menu}
AppendResMenu(GetMenuHandle(mApple), 'DRVR');
MyAdjustMenus;         {adjust items and enabled state of menus}
DrawMenuBar;          {draw the menu bar}
END;

```

The code in Listing 3-5 creates the application's menu bar by reading in the definition from the 'MBAR' resource with resource ID 128, and it uses `SetMenuBar` to set the current menu list to the newly created menu list. The code then adds the desktop items in the Apple Menu Items folder to the Apple menu using the `AppendResMenu` procedure.

You can use the `GetMenuHandle` function to get a handle to the menu record of any menu in the current menu list. You supply the menu ID of the desired menu as a parameter to `GetMenuHandle`, and `GetMenuHandle` returns a handle to the menu's menu record. Most Menu Manager routines require either a menu ID or a handle to a menu record as a parameter.

After creating the menu bar and adding any other menus or items as necessary, the code calls the `MyAdjustMenus` procedure to adjust the application's menus—for example, this procedure sets the enabled and disabled states of menu items in accordance with the current state of the application. (Listing 3-19 on page 3-74 shows the application-defined `MyAdjustMenus` procedure used in Listing 3-5.) After adjusting the menus, the code in Listing 3-5 uses `DrawMenuBar` to draw the menus in the menu bar according to their current enabled state and as they are defined in the current menu list.

Usually you'll define the menus of your application and its menu bar using 'MENU' resources and an 'MBAR' resource and using the `GetNewMBar` function to read the resource definitions. However, you can choose to read in a 'MENU' resource using the `GetMenu` function or to create a new empty menu using `NewMenu`. You can then insert a menu into the current menu list using the `InsertMenu` procedure. See “Creating Menus” on page 3-105 and “Adding Menus to and Removing Menus From the Current Menu List” on page 3-108 for information on forming your menus using these routines.

If your application uses a submenu, you need to use the `GetMenu` function and `InsertMenu` procedure to make these menus available to your application. See “Creating a Hierarchical Menu” on page 3-53 for information on creating submenus. If your application uses a pop-up menu, you can use the pop-up control definition function and Dialog Manager or Control Manager routines to create and display the pop-up menu. See “Creating a Pop-Up Menu” on page 3-56 for information on creating pop-up menus.

Menu Manager

The Menu Manager creates and initializes your application's menu color information table when your application calls `GetNewMBar`. You can add entries to your application's menu color information table if you want to use colors other than the default colors in your menus and menu bar. You can add entries to this table by providing menu color information table ('mctb') resources or by using the `SetMCEntries` procedure. Usually you should use the default colors to help maintain a consistent user interface.

If you add menu color entries to your application's menu color information table and your application uses more than one menu bar, you need to save a copy of your application's menu color information table before changing menu bars. Use the `GetMCInfo` function before calling `GetNewMBar` and call `SetMCInfo` afterward to restore the menu color information table. Listing 3-6 shows a routine that saves and then restores the menu color information table when creating a new menu bar.

Listing 3-6 Saving and restoring menu color information

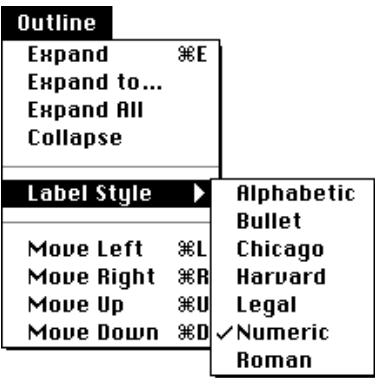
```
PROCEDURE MyChangeMenuBarAndSaveColorInfo;
CONST
    rMenuBar2 = 129;
VAR
    menu:           MenuHandle;
    menuBar:        Handle;
    currentMCTable: MCTableHandle;
    newMCTable:     MCTableHandle;
BEGIN
    currentMCTable := GetMCInfo;      {save menu color info table}
    IF currentMCTable = NIL THEN
        EXIT(MyChangeMenuBarAndSaveColorInfo);
    menuBar := GetNewMBar(rMenuBar2); {read menus in & create new menu list}
    IF menuBar = NIL THEN
        EXIT(MyChangeMenuBarAndSaveColorInfo);
    newMCTable := GetMCInfo;          {get new menu color info table}
    IF newMCTable = NIL THEN
        EXIT(MyChangeMenuBarAndSaveColorInfo);
    SetMCInfo(currentMCTable);        {restore previous menu color info table}
    SetMenuBar(menuBar);               {insert menus into the current menu list}
    DisposHandle(menuBar);
    AppendResMenu(GetMenuHandle(m2Apple), 'DRVr'); {add desktop items from }
                                                { Apple Menu Items folder to Apple menu}
    MyAdjustMenus;                    {adjust menu items}
    DrawMenuBar;                      {draw the menu bar}
END;
```


Creating a Hierarchical Menu

A hierarchical menu is a menu that has a submenu attached to one or more of its menu items. Submenus can be useful when your application needs to offer additional choices to the user without taking up extra space in the menu bar. If you use a submenu in your application, use it to give the user additional choices or to choose attributes, not additional commands.

A menu item of a pull-down menu is the title of the attached submenu. A menu item that has a triangle facing right in the location of the keyboard equivalent identifies that a submenu is attached to the menu item. The title of a submenu should represent the choices it contains. Figure 3-32 shows a menu with a submenu whose menu title is Label Style.

Figure 3-32 A menu item with a submenu



When a user drags the cursor through a menu and rests it on a menu item with a submenu attached to it, the Menu Manager displays the submenu after a brief delay. The title of the submenu remains highlighted while the user browses through the submenu; the Menu Manager unhighlights the menu title of the submenu when the user releases the mouse button.

Your application is responsible for placing any marks next to the current choice or attribute of the submenu. For example, in Figure 3-32 the application placed the checkmark next to the Numeric menu item to indicate the current choice. If the user makes a new choice from the menu, your application should update the menu items accordingly.

You can specify that a particular menu item has a submenu by identifying this characteristic (using the `hierarchicalMenu` constant) when you define the menu item in its 'MENU' resource. You cannot assign keyboard equivalents to a menu item that has a submenu. (You can define keyboard equivalents for the menu items in the submenu, but this is not recommended.) You identify the menu ID of the submenu in place of the marking character in the menu item's resource description. Thus, a menu item that has a submenu cannot have a marking character and cannot have a keyboard equivalent.

Menu Manager

To insert a submenu into the current menu list, you must use the `InsertMenu` procedure. The `GetNewMBar` function does not read in the resource descriptions of any submenus.

Listing 3-7 shows the 'MENU' resource for an application-defined menu called Outline. The Outline menu contains a number of menu items, including the Label Style menu item. The description of this menu item contains the constant `hierarchicalMenu`, which indicates that the item has a submenu. This menu item description also contains the menu ID of the submenu (defined by the `mSubMenu` constant). The menu ID of a submenu of an application must be from 1 through 235; the menu ID of a submenu of a desk accessory must be from 236 through 255.

The submenu is defined by the menu with the menu ID specified by the Label Style menu item. You define the menu items of a submenu in the same way as you would a pull-down menu (except you shouldn't define keyboard equivalents for items in a submenu, and you shouldn't attach a submenu to another submenu).

Listing 3-7 Rez input for a description of a hierarchical menu with a submenu

[illegible]

Menu Manager

```

resource 'MENU' (mSubMenu , preload) {
    mSubMenu ,                      /*menu ID*/
    textMenuProc,
    0b00000000000000000000000011111111,
    enabled,
    "Label Style",                  /*menu title (ignored--defined */
                                    /* by parent menu item text)*/
    {                               /*menu items*/
        "Alphabetic",              noicon, nokey, nomark, plain;
        "Bullet",                  noicon, nokey, nomark, plain;
        "Chicago",                 noicon, nokey, nomark, plain;
        "Harvard",                 noicon, nokey, nomark, plain;
        "Legal",                   noicon, nokey, nomark, plain;
        "Numeric",                 noicon, nokey, nomark, plain;
        "Roman",                   noicon, nokey, nomark, plain
    }
};

```

When you use `GetNewMBar` to read in menu descriptions and create a new menu list, `GetNewMBar` records the menu ID of any submenu in the menu record but does not read in the description of the submenu. To read a description of a submenu, use the `GetMenu` function. To actually insert a submenu into the current menu list, you must use the `InsertMenu` procedure.

When needed, your application can use the `GetMenu` function to read a description of the characteristics of a menu from a 'MENU' resource. The `GetMenu` function creates a menu record for the menu, allocating space for the menu record in your application's heap. The `GetMenu` function creates the menu and menu items (and fills in the menu record) according to its 'MENU' resource. The `GetMenu` function does not insert the menu into a menu list. When you're ready to add it to the current menu list, use the `InsertMenu` procedure.

Listing 3-8 uses the `GetMenu` function to read the description of a submenu and uses the `InsertMenu` procedure to insert the menu into the current menu list.

Listing 3-8 Creating a hierarchical menu

```

PROCEDURE MyMakeSubMenu (subMenuResID: Integer);
VAR
    subMenu: MenuHandle;
BEGIN
    subMenu := GetMenu(subMenuResID);
    InsertMenu(subMenu, -1);
END;

```

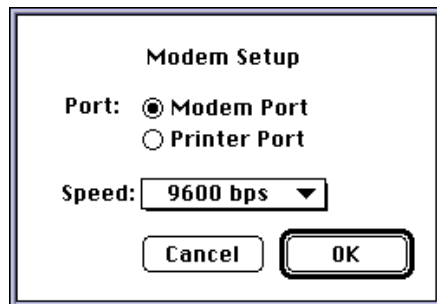
Menu Manager

To insert a submenu into the current menu list using the `InsertMenu` procedure, specify `-1` in the second parameter to insert the menu into the submenu portion of the menu list. As the user traverses menu items, if a menu item has a submenu the `MenuSelect` function looks in the submenu portion of the menu list for the submenu; it searches for a menu with a defined menu ID that matches the menu ID specified by the hierarchical menu item. If it finds a menu with a matching menu ID, it attaches the submenu to the menu item and allows the user to browse through the submenu.

Creating a Pop-Up Menu

In System 7, pop-up menus are implemented as controls. You define the menu items of a pop-up menu in the same way as in other menus (using a 'MENU' resource), and you define specific features of the pop-up menu itself (such as the location of the pop-up menu) in a control that uses the standard pop-up control definition function. Pop-up menus provide the user with a simple way to select from among a list of choices without having to move up to the menu bar. They are particularly useful in a dialog box that requires the user to specify a number of settings or values. Figure 3-33 shows an example of a pop-up menu in a dialog box.

Figure 3-33 A pop-up menu in a dialog box



To create a pop-up menu, create a control that uses the pop-up control definition function, define the pop-up menu and its menu items, and associate the control with a window or dialog box. You can use Dialog Manager or Control Manager routines to display pop-up menus.

For example, if you define a modal dialog box that contains a pop-up control and use the Dialog Manager to display and help handle events in the dialog box, the Dialog Manager automatically uses the pop-up control definition function to draw the control and also to handle user interaction when the user presses the mouse button while the cursor is over a pop-up control.

If your application defines a control in one of your application's windows, you can use `TrackControl` and other Control Manager routines to handle the pop-up menu.

The pop-up control definition function draws a box around the pop-up box, draws the drop shadow, inserts the text into the pop-up box, draws the downward-pointing triangle, and draws the pop-up title. When a dialog box contains a control that uses the

Menu Manager

pop-up control definition function and the user presses the mouse button while the cursor is in the pop-up control, the pop-up control definition function highlights the pop-up menu title, displays the pop-up menu, and handles all user interaction until the user releases the mouse button. When the user releases the mouse button, the pop-up control definition function closes the pop-up box, draws the user's choice in the pop-up box (or restores the previous item if the user did not make a new choice), stores the user's choice as the value of the control, and unhighlights the pop-up menu title. Your application can use the Control Manager function `GetControlValue` to get the value of the control and to determine the currently selected item in the pop-up menu.

To create a pop-up control, create a control and specify that the control uses the pop-up control definition function by specifying the `popupMenuProc` constant:

```
CONST popupMenuProc = 1008;          {pop-up menu control}
```

If you specify `popupMenuProc` (plus any appropriate variation code) as the `procID` field of the resource description of a control, when your application creates the control (by using the Dialog Manager or by using `GetNewControl`) the Control Manager creates the pop-up control, which includes the pop-up title and the pop-up box with a one-pixel drop shadow. The appearance of the pop-up title and the values in the menu are controlled by other values stored in the resource (or other parameters passed to `NewControl`). See the chapter “Control Manager” in this book for information on the `NewControl` function.

If your application does not use the standard pop-up control definition function, you can create your own control definition function to implement pop-up menus. In this case you can use the `PopUpMenuSelect` function to draw the pop-up menu and track the cursor within the menu. Your application is responsible for highlighting the title of the pop-up menu before calling `PopUpMenuSelect` and unhighlighting the title afterward (to duplicate the behavior of menu titles in the menu bar). Your application must also set the mark of the items in the pop-up menu as appropriate if you use the `PopUpMenuSelect` function.

For more information on creating controls, see the chapter “Control Manager” in this book. For listings that define the dialog box shown in Figure 3-33, see the chapter “Dialog Manager” in this book.

Changing the Appearance of Items in a Menu

You can change the appearance of an item in a menu using Menu Manager routines. For example, you can change the font style, text, or other characteristics of menu items. You can also enable or disable a menu item.

Most of the Menu Manager routines that get or set characteristics of a particular menu item require three parameters:

- a handle to the menu record of the menu containing the desired menu item
- the number of the menu item
- a variable that either specifies the data to set or identifies where to return information about that item

Enabling and Disabling Menu Items

Using the `EnableItem` and `DisableItem` procedures, you can enable and disable specific menu items or an entire menu. You pass as parameters to these two procedures a handle to the menu record that identifies the desired menu and either an item number that identifies the particular menu item to enable or disable or a value of 0 to indicate that the entire menu should be enabled or disabled.

Your application should always enable and disable any menu items as appropriate—according to the user’s content—before calling `MenuSelect` or `MenuKey`. For example, you should enable the Paste command when the scrap contains data that the user can paste. (Listing 3-19 on page 3-74 shows code that adjusts an application’s menus.)

When you disable or enable an entire menu, call `DrawMenuBar` to update the menu bar. The `DrawMenuBar` procedure draws the menus in the menu bar according to their current enabled state and as they are defined in the current menu list.

If you disable an entire menu, the Menu Manager dims the menu title at your application’s next call to `DrawMenuBar` and dims all items in the menu when it displays the menu. If you enable an entire menu, the Menu Manager enables only the menu title and any items that you did not previously disable individually; the Menu Manager does not enable any item that your application previously disabled by calling `DisableItem` with that menu item’s item number. For example, if all items in your application’s Edit menu are enabled, you can disable the Cut and Copy commands individually using `DisableItem`. If you choose to disable the entire menu by passing 0 as the menu item parameter to `DisableItem`, the menu and all its items are disabled. If you then enable the entire menu by passing 0 as the menu item parameter to `EnableItem`, the menu and its items are enabled, except for the Cut and Copy commands, which remain disabled. In this case, to enable the Cut and Copy commands, you must enable each one individually using `EnableItem`.

You can use `DisableItem` to disable items that aren’t appropriate at a given time. For example, you can disable the Cut and Copy commands when the user has not selected anything to cut or copy and disable the Paste command when the scrap is empty.

This code enables the File menu, disables the Cut and Copy commands in the Edit menu, and disables the application-defined menu Colors.

```
VAR
    menu: MenuHandle;

    menu := GetMenuHandle(mFile); {get a handle to the File menu}
    EnableItem(menu, 0);           {enable File menu and any items }
                                   { not individually disabled}
    DrawMenuBar;                  {update menu bar's appearance}

    menu := GetMenuHandle(mEdit); {get a handle to the Edit menu}
    DisableItem(menu, iCut);       {disable the Cut command}
    DisableItem(menu, iCopy);      {disable the Copy command}
```

Menu Manager

```

menu := GetMenuHandle(mColors); {get a handle to Colors menu}
DisableItem(menu, 0);           {disable Colors menu & all }
                                { items in it}
DrawMenuBar;                   {update menu bar's appearance}

```

If you disable or enable an entire menu, call `DrawMenuBar` when you need to update the menu bar's appearance. If you do not need to update the menu bar immediately, you can use the `InvalidMenuBar` procedure instead of `DrawMenuBar`, thus reducing flickering in the menu bar. Rather than drawing the menu bar twice as in the previous example, you can use `InvalidMenuBar` instead of `DrawMenuBar`, causing the Event Manager to redraw the menu bar the next time it scans for update events. The `InvalidMenuBar` procedure is available in System 7 and later. See page 3-114 for additional details on the `InvalidMenuBar` procedure.

Changing the Text of an Item

You can get or set the text of a menu item using Menu Manager routines.

To get the text of a menu item, use the `GetMenuItemText` procedure. For example, you can use the `GetMenuItemText` procedure to get the text of a menu item that you added to a menu using `InsertResMenu` or `AppendResMenu`.

To set the text of a menu item, use the `SetMenuItemText` procedure. You can use the `SetMenuItemText` procedure as a convenient way to change the text of a menu command that allows the user to toggle between two states. For example, if your application has a menu command that allows the user to either show or hide the Clipboard window, depending on whether the window is currently showing, you can change the text of the menu item at the appropriate time using the `SetMenuItemText` procedure.

Listing 3-9 changes the text of a menu item from Hide Clipboard to Show Clipboard or vice versa, based on the state of an application-defined global variable (`gToggleState`) that holds the state information.

Listing 3-9 Changing the text of a menu item

```

PROCEDURE MyToggleHideShow;
VAR
    myMenu:      MenuHandle;
    item:        Integer;
    itemString:  Str255;
BEGIN
    myMenu := GetMenuHandle(mEdit);
    item := iToggleHideShow;
    IF gToggleState = kShow THEN
    BEGIN
        GetIndString(itemString, kMyStrings, kShowClipboard);
        gToggleState := kHide;
    END
END

```

Menu Manager

```

ELSE
BEGIN
    GetIndString(itemString, kMyStrings, kHideClipboard);
    gToggleState := kShow;
END;
SetMenuItemText(myMenu, item, itemString);
END;

```

Note that if you use the `SetMenuItemText` procedure, you should define the text of the menu item in a string resource or string list resource (for example, using an 'STR' or 'STR#' resource). This makes your application easier to localize.

Changing the Font Style of Menu Items

You can change or get the font style of a menu item using the `SetItemStyle` or `GetItemStyle` procedure. To set the style of a menu item, specify a handle to the menu record of the menu containing the menu item whose style you want to set, specify the number of the menu item to set, and specify the desired style.

You specify the style using values from the set defined by the `Style` data type:

```

TYPE
    StyleItem = (bold, italic, underline, outline, shadow,
                 condense, extend);
    Style = SET OF StyleItem;

```

You can set the style of a menu item to zero, one, or more than one of the styles defined by the `StyleItem` data type. You can set the style of a menu item to the empty set to obtain the plain font style.

Listing 3-10 shows code that sets the style of menu items listed in an application's `Style` menu.

Listing 3-10 Setting the font style of menu items

```

VAR
    menu:      MenuHandle;
    itemStyle: Style;

    menu := GetMenuHandle(mStyle); {get a handle to the Style menu}
    itemStyle := [italic];
    SetItemStyle(menu, iItalic, itemStyle); {set to italic style}
    itemStyle := [bold];
    SetItemStyle(menu, iBold, itemStyle); {set item to bold style}
    itemStyle := [bold, Italic];
    SetItemStyle(menu, iBoldItal, itemStyle); {bold & italic style}
    itemStyle := [];
    SetItemStyle(menu, iPlain, itemStyle); {set item to plain style}

```

To get the style of a menu item, you can use the `GetItemStyle` procedure.

Changing the Mark of Menu Items

You can change or get the mark of a menu item using the `SetItemMark` or `GetItemMark` procedure. To set the mark of a menu item to a checkmark, you can use either the `CheckItem` or the `SetItemMark` procedure.

To set the mark of a menu item, specify a handle to the menu record of the menu containing the item whose mark you want to set, specify the number of the menu item to set, and specify the mark to use as the marking character of the menu item.

You typically use checkmarks and dashes in menus that contain commands that set attributes and that you have grouped in accumulating groups. For example, you use a combination of checkmarks and dashes in the Style menu to indicate whether the selection contains more than one style. Figure 3-8 on page 3-15 shows an example of using checkmarks and dashes in a menu. “Groups of Menu Items” beginning on page 3-14 gives guidelines for determining how to group your menu items.

You specify the mark of the menu item by passing a character as one of the parameters to the `SetItemMark` procedure. You should use only the standard marking characters, such as the checkmark, diamond, or dash, in your menu items; avoid using other marks that might confuse the user. You can use the constants listed here to specify that the item has no mark or to set the marking character to a checkmark or diamond:

```
CONST noMark      = 0;      {no marking character}
      checkMark   = $12;    {checkmark}
      diamondMark = $13;    {diamond symbol}
```

As another example of the use of marks in menus, Listing 3-11 shows code that sets the mark of items in an application-defined Directory menu. It sets the marking character of the menu item of the last directory accessed to a checkmark, sets the marking character of the second-last directory accessed to the diamond mark, and removes the mark from the third-last directory accessed.

Listing 3-11 Adding marks to and removing marks from menu items

```
VAR
    menu:      MenuHandle;
    itemMark:  Char;

                                {get handle to Directory menu}
    menu := GetMenuHandle(mDirectory);

    itemMark := CHR(checkMark);
    SetItemMark(menu, gLastDir, itemMark); {set mark to checkmark}

    itemMark := CHR(diamondMark);
    SetItemMark(menu, gOldLastDir, itemMark); {set mark to diamond}

    itemMark := CHR(noMark);
    SetItemMark(menu, gSecondLastDir, itemMark); {remove any mark}
```

Menu Manager

You can also set the mark of a menu item to a checkmark using the `CheckItem` procedure:

```
VAR
    menu:      MenuHandle;
                                {get handle to Directory menu}
    menu := GetMenuHandle(mDirectory);
    CheckItem(menu, gLastDir, TRUE);      {set to checkmark}
    CheckItem(menu, gSecondLastDir, FALSE); {remove checkmark or }
                                           { any other mark}
```

Changing the Icon or Script Code of Menu Items

You can change or get the icon of a menu item using the `SetItemIcon` or `GetItemIcon` procedure. You can also use these procedures to get or set the script code of a menu item's text.

To set the script code of a menu item using the `SetItemIcon` procedure, you need to

- specify a handle to the menu record of the menu containing the item whose script code you want to set
- specify the number of the menu item to set
- specify the script code

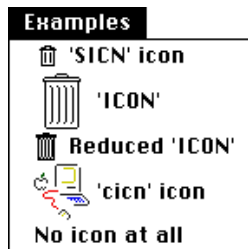
To set a menu item's script code, you must also define the keyboard equivalent field of the item to `$1C`. If an item contains `$1C` in its keyboard equivalent field and a script code in its icon field, the Menu Manager draws the item in the script identified by the script code value if the corresponding script system is installed.

To set the icon of a menu item using the `SetItemIcon` procedure, you need to

- specify a handle to the menu record of the menu containing the item whose icon you want to set
- specify the number of the menu item to set
- specify the icon number (the Menu Manager uses the icon number to generate the resource ID of the icon)

The icon number that you specify to `SetItemIcon` must be a value from 1 through 255 for color icons or icons, from 1 through 254 for small icons and reduced icons, or 0 to specify that the item doesn't have an icon. The Menu Manager adds 256 to the number you specify and uses this calculated number as the icon's resource ID. For example, if you specify the icon number as 5, the Menu Manager uses the Resource Manager to find the icon with resource ID 261. The Menu Manager first looks for an icon resource of type `'cicn'`; if it can't find one with the calculated resource ID number (or if the computer doesn't have Color QuickDraw), it looks for a resource of type `'SICN'` if the keyboard equivalent field contains `$1E`; otherwise, it looks for an `'ICON'` resource.

Use either an `'ICON'` or `'SICN'` resource if you want to provide only a black-and-white icon. In addition, provide a `'cicn'` resource if you want the Menu Manager to use a color icon when Color QuickDraw is available. Figure 3-34 shows examples of icons in a menu item generated from icon resources: an `'SICN'` resource, an `'ICON'` resource, an `'ICON'` resource reduced to fit in a 16-by-16 bit rectangle, and a `'cicn'` resource.

Figure 3-34 Icons in menu items

The Menu Manager automatically fits the icon in the menu item according to your specifications. If the Menu Manager uses a 'cicn' resource, it automatically enlarges the enclosing rectangle of the menu item according to the rectangle specified in the 'cicn' resource. If the Menu Manager uses an 'ICON' resource and the item specifies the nokey constant as the keyboard equivalent, the Menu Manager enlarges the rectangle of the menu item to fit the 32-by-32 bit 'ICON' resource. You can request that the Menu Manager reduce an 'ICON' resource to the size of a 16-by-16 bit small icon by specifying a value of \$1D as the item's keyboard equivalent. To request that the Menu Manager use an 'SICN' resource instead of an 'ICON' resource, specify a value of \$1E as the item's keyboard equivalent.

This code sets the icon of a menu item to a specified icon.

```
VAR
    menu:      MenuHandle;
    itemIcon:  Byte;

    itemIcon := 5;
    menu := GetMenuHandle(mWeather);
    {set the icon for this item in the Weather menu}
    SetItemIcon(menu, iBeachWeather, itemIcon);
```

Listing 3-12 shows the Rez description of three menu items, each of which contains icons. The first menu item has an icon with resource ID 261 (5 plus 256) and is defined by a resource type of either 'cicn' or 'ICON'. The second menu item has an icon with resource ID 262 (6 plus 256) and is identified by either a 'cicn' resource or an 'ICON' resource; however, in this case, the value of \$1D requests the Menu Manager to reduce the 'ICON' resource to a small icon. The third menu item has an icon with resource ID 263 (7 plus 256) and is defined by either a 'cicn' resource or an 'SICN' resource.

Listing 3-12 Specifying icons for menu items

```
#define mWeather 138
resource 'MENU' (mWeather, preload) {
    mWeather,
    textMenuProc,
```

See the chapter “Finder Interface” in this book for details on how to create icons.

Usually you define a menu and all its items in a 'MENU' resource. Occasionally you might need to add items to a menu after you've created it. After creating a menu (using `NewMenu`, `GetMenu`, or `GetNewMBar`), you can add items to it using the `AppendMenu`, `InsertMenuItem`, `AppendResMenu`, or `InsertResMenu` procedure.

If you add items to your application's Help menu, you'll need to use `AppendMenu` or `InsertMenuItem` to add the additional items. This section discusses how to add items using the `AppendMenu` and `InsertMenuItem` procedures, and "Adding Items to the Help Menu" on page 3-67 shows a specific example of adding items to the Help menu.

If you need to add items other than the names of resources to a previously created menu, you can use the `AppendMenu` or `InsertMenuItem` procedure. You can use `AppendMenu` to add items to the end of a menu; note that you can add items to only the end of the menu when using `AppendMenu`. Use `InsertMenuItem` to add items after any given item in a menu. When you add items to a menu using `AppendMenu` or `InsertMenuItem`, you can specify the same characteristics for menu items that are available to you when defining 'MENU' resources.

Menu Manager

You specify a handle to the menu record of the menu to which you want to add the item or items, and you specify a string describing the items to add as parameters to the `AppendMenu` or `InsertMenuItem` procedure. The string you pass to these procedures should consist of the text and any characteristics of the menu items. You can specify a hyphen as the menu item text to create a divider line. You can also use various metacharacters in the text string to separate menu items and to specify certain characteristics of the menu items. The metacharacters aren't displayed in the menu.

Here is a list of the metacharacters you can use in the text string that you specify to the `AppendMenu` or `InsertMenuItem` procedure:

Metacharacter	Description
;	Separates menu items.
^	When followed by an icon number, defines the icon for the item. If the keyboard equivalent field contains \$1C, this number is interpreted as a script code.
!	When followed by a character, defines the mark for the item. If the keyboard equivalent field contains \$1B, this value is interpreted as the menu ID of a submenu of this menu item.
<	When followed by one or more of the characters B, I, U, O, and S, defines the character style of the item to Bold, Italic, Underline, Outline, or Shadow, respectively.
/	When followed by a character, defines the keyboard equivalent for the item. When followed by \$1B, specifies that this menu item has a submenu. To specify that the menu item has a script code, small icon, or reduced icon, use the <code>SetItemCmd</code> procedure to set the keyboard equivalent field to \$1C, \$1D, or \$1E, respectively.
(Defines the menu item as disabled.

You can specify any, all, or none of these metacharacters in the text string to define the characteristics of a menu item. Note that the metacharacters that you specify aren't displayed in the menu item. (To use any of these metacharacters in the text of a menu item, first use `AppendMenu` or `InsertMenuItem`, specifying at least one character as the item's text. Then use the `SetMenuItemText` procedure to set the item's text to the desired string.)

Note

If you add menu items using the `AppendMenu` or `InsertMenuItem` procedure, you should define the text and any marks or keyboard equivalents in resources for easier localization. ♦

Listing 3-13 shows a string list ('STR#') resource that stores the text of the menu items used in the next examples.

Menu Manager

Listing 3-13 Rez input for text of menu items

```
resource 'STR#' (300, "Text for appended menu items") {
    {
        /*[1]*/
        "Just Text";
        /*[2]*/
        "Pick a Color...";
        /*[3]*/
        " (^2!=Everything<B/E";
    }
};
```

Here's code that uses the `AppendMenu` procedure to append a menu item with no specific characteristics other than its text to the menu identified by the menu handle in the `myMenu` variable. The text for the menu item is "Just Text" as stored in the 'STR#' resource with resource ID 300.

```
VAR
    myMenu:      MenuHandle;
    itemString:  Str255;

    myMenu := GetMenuHandle(mLibrary);
    GetIndString(itemString, 300, 1);
    AppendMenu(myMenu, itemString);
```

To insert an item after a given menu item, use the `InsertMenuItem` procedure. For example, this code inserts the menu item `Pick a Color` after the menu item with the item number specified by the `iRed` constant. The text for the new menu item consists of the string "Pick a Color..." as stored in the 'STR#' resource with resource ID 300.

```
VAR
    myMenu:      MenuHandle;
    itemString:  Str255;

    myMenu := GetMenuHandle(mColors);
    GetIndString(itemString, 300, 2);
    InsertMenuItem(myMenu, itemString, iRed);
```

If you do not explicitly specify a value for an item characteristic in the text string that you pass to `AppendMenu` or `InsertMenuItem`, the procedure assigns the default value for that characteristic. The Menu Manager defines the default item characteristics as no icon, no marking character, no keyboard equivalent, and plain text style. `AppendMenu` and `InsertMenuItem` enable the added menu items unless you specify otherwise.

This code appends a menu item with the text "Everything" to the menu identified by the menu handle in the `myMenu` variable. The text and other characteristics of this menu item are stored in the 'STR#' resource shown in Listing 3-13. It also specifies that this

Menu Manager

menu item is disabled, has an icon with resource ID 258 (2 + 256), and has the “=” character as a marking character; the style of the text is Bold; and the menu item has a keyboard equivalent of Command-E.

```
VAR
    myMenu:      MenuHandle;
    itemString: Str255;

    myMenu := GetMenuHandle(mLibrary);
    GetIndString(itemString, 300, 3);
    AppendMenu(myMenu, itemString);
```

This code appends multiple items to the Edit menu using `AppendMenu`:

```
VAR
    myMenu:      MenuHandle;

    myMenu := GetMenuHandle(mEdit);
    AppendMenu(myMenu, 'Undo/Z;-;Cut/X;Copy/C;Paste/V');
```

The `InsertMenuItem` procedure differs from `AppendMenu` in how it handles the given text string when the text string specifies multiple items. The `InsertMenuItem` procedure inserts the items in the reverse of their order in the text string. For example, this code inserts menu items into the Edit menu using `InsertMenuItem` and is equivalent to the previous example:

```
VAR
    myMenu:      MenuHandle;

    myMenu := GetMenuHandle(mEdit);
    InsertMenuItem(myMenu, 'Paste/V';Copy/C;Cut/X;-;Undo/Z', 0);
```

Once you’ve added a menu item to a menu, you can change any of its characteristics using Menu Manager routines, as described in “Changing the Appearance of Items in a Menu” on page 3-57.

Adding Items to the Help Menu

You add items to the Help menu by using the `HMGetHelpMenuHandle` function and either the `AppendMenu` or `InsertMenuItem` procedure.

The `HMGetHelpMenuHandle` function returns a copy of the handle to the menu record of your application’s Help menu. Do not use the `GetMenuHandle` function to get a handle to the Help menu, because `GetMenuHandle` returns a handle to the global Help menu, not the Help menu that is specific to your application. Once you have a handle to the Help menu that is specific to your application, you can add items to it using the `AppendMenu` procedure or other Menu Manager routines. For example, Listing 3-14 adds the menu item displayed in Figure 3-19 on page 3-30.

Menu Manager

Listing 3-14 Adding an item to the Help menu

```

PROCEDURE MyAddHelpItem;
VAR
    myMenu:      MenuHandle;
    myErr:       OSErr;
    itemString:  Str255;
BEGIN
    myErr := HMGetHelpMenuHandle(myMenu);
    IF myErr = noErr THEN
        IF myMenu <> NIL THEN
            BEGIN
                {get the string (with index kSurfHelp) from the 'STR#' }
                { resource with resource ID kMyStrings}
                GetIndString(itemString, kMyStrings, kSurfHelp);
                AppendMenu(myMenu, itemString);
            END;
        END;
    END;

```

When you add items to the Help menu, the Help Manager automatically adds a divider between the end of the standard Help menu items and your items.

Be sure to use an 'hmnv' resource and specify the kHMHelpMenuID constant as the resource ID to provide help balloons for items you've added to the Help menu. (The Help Manager automatically creates the help balloons for the Help menu title and the standard Help menu items.) See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for specific information on the 'hmnv' resource.

The Help Manager automatically processes the event when a user chooses any of the standard menu items in the Help menu. The Help Manager automatically enables and disables help when the user chooses Show Balloons or Hide Balloons from the Help menu. The setting of Balloon Help is global and affects all applications. See "Handling the Help Menu" on page 3-81 for information on responding to the user when the user chooses one of your appended items.

Adding Items to the Apple Menu

To insert the items contained in the Apple Menu Items folder into your application's Apple menu, use the `AppendResMenu` or `InsertResMenu` procedure and specify 'DRVr' as the resource type. Doing so causes this procedure to automatically add all items in the Apple Menu Items folder to the specified menu.

The user can place any desktop object in the Apple Menu Items folder. When the user places an item in this folder, the system software automatically adds it to the list of items in the Apple menu of all open applications.

Menu Manager

After inserting the Apple menu into your application's menu bar (by using `GetNewMBar` or `GetMenu` and `InsertMenu`), your application can add items to it. Listing 3-15 shows code that uses `GetMenuHandle` to get a handle to the application's Apple menu. The code then uses the `AppendResMenu` procedure, specifying that `AppendResMenu` should add all desktop items in the Apple Menu Items folder to the application's Apple menu.

Listing 3-15 Adding menu items to the Apple menu

```
VAR
    myMenu:           MenuHandle;

    myMenu := GetMenuHandle(mApple);
    IF myMenu <> NIL THEN
        AppendResMenu(myMenu, 'DRVR'); {add desktop items in the }
                                       { Apple Menu Items folder }
                                       { to the Apple menu}
```

Listing 3-16 on page 3-70 shows a complete sample that sets up an application's menu bar, adds items to the Apple menu, adds items to the Font menu, and then updates the appearance of the menu bar.

Adding Fonts to a Menu

If your application provides a Font menu, you typically include the description of the menu in a 'MENU' resource, include a description of your menu bar in an 'MBar' resource, and use `GetNewMBar` to create your menu bar and all menus in the menu bar. Once you've created the menu, you can use `AppendResMenu` to add the names of all font resources in the Fonts folder of the System Folder (or in system software versions earlier than 7.1, in the System file) as menu items in your application's Font menu. (You can also use `InsertResMenu` to insert the fonts into your menu.)

Listing 3-16 on the next page shows how to add names of font resources in the Fonts folder to an application's Font menu. The `AppendResMenu` procedure adds all resources of the specified type to a given menu. If you specify the resource type 'FONT' or 'FOND', the Menu Manager adds all resources of type 'FOND' and 'FONT' to the menu. ('NFNT' and 'sfnt' resources are specified through 'FOND' resources.)

The `AppendResMenu` and `InsertResMenu` procedures perform special processing for any font resources they find that have font numbers greater than \$4000. The Menu Manager automatically sets the keyboard equivalent field of the menu item to \$1C and stores the script code in the icon field of the menu item for any such 'FOND' resource. The Menu Manager displays a font name in its corresponding script if the script system for that font is installed.

Listing 3-16 Adding font names to a menu

```

PROCEDURE MyMakeAllMenus;
VAR
    menu:                MenuHandle;
    menuBar:             Handle;
BEGIN
                                {read menus in & create new menu list}
    menuBar := GetNewMBar(rMenuBar);
    IF menuBar = NIL THEN
        EXIT(MyMakeAllMenus);
    SetMenuBar(menuBar); {insert menus into the current menu list}
    DisposHandle(menuBar);
    myMenu := GetMenuHandle(mApple);
    IF myMenu <> NIL THEN                                {add desktop items in }
        AppendResMenu(myMenu, 'DRVR');                    { Apple Menu Items }
                                                { folder to Apple menu}

    myMenu := GetMenuHandle(mFont);
    IF myMenu <> NIL THEN
        AppendResMenu(myMenu, 'FONT');                    {add font names to the }
        { Font menu--this adds all bitmapped and TrueType fonts }
        { in the Fonts folder to the Font menu}
    MyAddHelpItem;                {add app-specific item to Help menu}
    MyAdjustMenus;                {adjust menu items}
    DrawMenuBar;                  {draw the menu bar}
END;

```

Your application should indicate the current font to the user by placing the appropriate mark next to the text of the menu item that lists the font name. (“Changing the Mark of Menu Items” on page 3-61 explains how to add marks to and remove marks from menu items; Figure 3-13 on page 3-26 and Figure 3-14 on page 3-27 show examples of typical Font menus.)

If your application allows the user to change the font style or font size of text, you should provide separate Size and Style menus. See “Handling a Size Menu” beginning on page 3-82 for information on providing a Size menu in your application.

Handling User Choice of a Menu Command

If the user presses the mouse button while the cursor is in the menu bar, your application should first adjust its menus (enable or disable menu items and add marks to or remove marks from any items as appropriate to the user’s context) and then call the `MenuSelect` function to allow the user to choose a menu command. The `MenuSelect` function handles all user interaction until the user releases the mouse button and returns a value as its function result that indicates which (if any) menu item the user chose.

Menu Manager

For a command with a keyboard equivalent, your application should allow the user to choose the command by pressing the keys that correspond to the keyboard equivalent of that menu command. If the user presses the Command key and another key, your application should adjust its menus and then call the `MenuKey` function to map this combination to a keyboard equivalent. The `MenuKey` function returns as its function result a value that indicates the corresponding menu and menu item of the keyboard equivalent.

When the user chooses a menu command, your application should perform the action associated with that command. The `MenuSelect` and `MenuKey` functions highlight the menu title of the menu containing the chosen menu command. After your application performs any operation associated with the menu command chosen by the user, your application should unhighlight the menu title by using the `HiliteMenu` procedure.

However, if in response to a menu command your application displays a window that contains editable text (such as a modal dialog box), you should unhighlight the menu title immediately so that the user can access the Edit menu or other appropriate menus. In other words, any time the user can use a menu, make sure that the menu title is not highlighted.

When the user chooses a menu command that involves an operation that takes a long time, display the animated wristwatch cursor or display a status dialog box to give the user feedback that the operation is in progress.

If you want the users of your application to be able to record their actions (such as menu commands, text input, or any sequence of actions) for later playback, your application should send itself Apple events whenever a user performs a significant action. To do this for menu commands, your application typically sends itself an Apple event to perform the action associated with the chosen menu command. For example, when a user chooses the New command from the File menu, your application can choose to send itself a Create Element event. Your application then creates the new document in response to this event. For information on sending Apple events in response to menu commands, see *Inside Macintosh: Interapplication Communication*.

The next sections show how your application can

- determine if the user pressed the mouse button while the cursor was in the menu bar
- adjust its menus—enabling and disabling commands according to the current state of the document—before displaying menus or before responding to the user’s choice of a keyboard equivalent of a command
- determine if the user chose the keyboard equivalent of a menu command
- respond to the user when the user chooses a menu command

The next sections also show how your application should respond when the user chooses an item from the Apple or Help menu.

Menu Manager

Handling Mouse-Down Events in the Menu Bar

You can determine when the user has pressed the mouse button while the cursor is in the menu bar by examining the event record for a mouse-down event. You can use the Window Manager function `FindWindow` to map the mouse location at the time of the mouse-down event to a corresponding area of the screen. If the cursor was in the menu bar, your application should call the `MenuSelect` function, allowing the user to choose a menu command.

Listing 3-17 shows an application-defined procedure, `DoEvent`, that determines whether a mouse-down event occurred and, if so, calls another application-defined procedure to handle the mouse-down event. (For a complete discussion of how to handle events, see the “Event Manager” chapter in this book.)

Listing 3-17 Determining whether a mouse-down event occurred

```
PROCEDURE DoEvent (event: EventRecord);
BEGIN
    CASE event.what OF
        mouseDown:                {handle mouse-down event}
            DoMouseDown(event);
        {handle other events appropriately}
    END; {of CASE}
END;
```

Listing 3-18 shows an application-defined procedure, `DoMouseDown`, that handles mouse-down events. The `DoMouseDown` procedure determines where the cursor was when the mouse button was pressed and then responds appropriately.

Listing 3-18 Determining when the cursor is in the menu bar

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
    part:      Integer;
    thisWindow: WindowPtr;
BEGIN
    part := FindWindow(event.where, thisWindow);
    CASE part OF
        inMenuBar: {mouse down in menu bar, respond appropriately}
            BEGIN
                {adjust marks and enabled state of menu items}
                MyAdjustMenus;
                {let user choose a menu command if desired}
                DoMenuCommand(MenuSelect(event.where));
            END;
    END;
```

Menu Manager

```

        {handle other mouse-down events appropriately}
    END; {of CASE}
END;

```

You can use the `FindWindow` function to map the mouse location at the time the user pressed the mouse button to general areas of the screen. If the mouse location is in the menu bar, the `FindWindow` function returns the constant `inMenuBar`. In Listing 3-18, if the mouse location associated with the mouse-down event is in the menu bar, the `DoMouseDown` procedure first calls another application-defined procedure, `MyAdjustMenus`, to adjust the menus. Listing 3-19 shows the `MyAdjustMenus` procedure.

The `DoMouseDown` procedure then calls an application-defined procedure, `DoMenuCommand`. The `DoMouseDown` procedure passes as a parameter to the `DoMenuCommand` procedure the value returned from the `MenuSelect` function.

The `MenuSelect` function displays menus and handles all user interaction until the user releases the mouse button. The `MenuSelect` function returns a long integer indicating whether the user chose a menu command, and if so, it indicates which menu and which command the user chose.

Listing 3-24 on page 3-79 shows the `DoMenuCommand` procedure.

Adjusting the Menus of an Application

Your application should always adjust its menus before calling `MenuSelect` or `MenuKey`. For example, you should enable and disable any menu items as necessary and add checkmarks or dashes to items that are attributes. When you adjust your application's menus, you should enable and disable menu items according to the type of window that is in the front. For example, when a document window is the frontmost window, you should enable items as appropriate for that document window. When a modeless dialog box or modal dialog box is the frontmost window, enable those items as appropriate to that particular dialog box. Listing 3-19 shows an application-defined routine, `MyAdjustMenus`, that adjusts the menus of the `SurfWriter` application appropriately.

The `MyAdjustMenus` procedure first determines what kind of window is in front and then adjusts the application's menus appropriately. The application-defined `MyGetWindowType` procedure returns a value that indicates whether the window is a document window, a dialog window, or a window belonging to a desk accessory. It also returns the constant `kNil` if there isn't a front window. (See the chapter "Window Manager" in this book for a listing of the `MyGetWindowType` procedure.) The `MyAdjustMenus` procedure calls other application-defined routines to adjust the menus as appropriate for the given window type.

Menu Manager

Listing 3-19 Adjusting an application's menus

```

PROCEDURE MyAdjustMenus;
VAR
    window:           WindowPtr;
    windowType:       Integer;
BEGIN
    window := FrontWindow;
    windowType := MyGetWindowType(window);
    CASE windowType OF
        kMyDocWindow:
            BEGIN {document window is in front, adjust items appropriately}
                MyAdjustFileMenuForDocWindow;
                MyAdjustEditMenuForDocWindow;
                {adjust other menus as needed}
            END; {of adjusting menus for a document window}
        kMyDialogWindow:
            {adjust menus accordingly for any dialog box}
            MyAdjustMenusForDialogs;
        kDAWindow:{adjust menus accordingly for a DA window}
            MyAdjustMenusForDA;
        kNil:{adjust menus accordingly when there isn't a front window}
            MyAdjustMenusNoWindows;
    END; {of CASE}
    DrawMenuBar;
END;

```

Listing 3-20 shows the application-defined procedure `MyAdjustFileMenuForDocWindow`. This procedure enables and disables the File menu for the application's document window, according to the state of the document. For example, this application always allows the user to create a new document or open a file, so the code enables the New and Open menu items. The code also enables the Close, Save As, Page Setup, Print, and Quit menu items. If the user has modified the file since last saving it, the code enables the Save command; otherwise, it disables the Save command.

Listing 3-20 Adjusting the File menu for a document window

```

PROCEDURE MyAdjustFileMenuForDocWindow;
VAR
    window:           WindowPtr;
    menu:             MenuHandle;
    myData:           MyDocRecHnd;

```

Menu Manager

```

BEGIN
    window := FrontWindow;
    menu := GetMenuHandle(mFile); {get a handle to the File menu}
    IF menu = NIL THEN             {add your own error handling}
        EXIT (MyAdjustFileMenuForDocWindow);
    EnableItem(menu, iNew);
    EnableItem(menu, iOpen);
    EnableItem(menu, iClose);
    myData := MyDocRecHnd(GetWRefCon(window));
    IF myData^^.windowDirty THEN
        EnableItem(menu, iSave)
    ELSE
        DisableItem(menu, iSave);
    EnableItem(menu, iSaveAs);
    EnableItem(menu, iPageSetup);
    EnableItem(menu, iPrint);
    EnableItem(menu, iQuit);
END;

```

Listing 3-21 shows the application-defined `MyAdjustEditMenuForDocWindow` procedure.

Listing 3-21 Adjusting the Edit menu for a document window

```

PROCEDURE MyAdjustEditMenuForDocWindow;
VAR
    window:           WindowPtr;
    menu:             MenuHandle;
    selection, undo:   Boolean;
    isSubscriber:      Boolean;
    undoText:          Str255;
    offset:            LongInt;
BEGIN
    window := FrontWindow;
    menu := GetMenuHandle(mEdit); {get a handle to the Edit menu}
    IF menu = NIL THEN             {add your own error handling}
        EXIT (MyAdjustEditMenuForDocWindow);
    undo := MyIsLastActionUndoable(undoText);
    IF undo THEN {if action can be undone}
        BEGIN
            SetMenuItemText(menu, iUndo, undoText);
            EnableItem(menu, iUndo);
        END
    END

```

Menu Manager

```

ELSE          {if action can't be undone}
BEGIN
    SetMenuItemText(menu, iUndo, gCantUndo);
    DisableItem(menu, iUndo);
END;
selection := MySelection(window);
IF selection THEN
BEGIN    {enable editing items if there's a selection}
    EnableItem(menu, iCut);
    EnableItem(menu, iCopy);
    EnableItem(menu, iCreatePublisher);
END
ELSE
BEGIN    {disable editing items if there isn't a selection}
    DisableItem(menu, iCut);
    DisableItem(menu, iCopy);
    DisableItem(menu, iCreatePublisher);
END;
IF GetScrap(NIL, 'TEXT', offset) > 0 THEN
    EnableItem(menu, iPaste)    {enable if something to paste}
ELSE
    DisableItem(menu, iPaste); {disable if nothing to paste}
EnableItem(menu, iSelectAll);
EnableItem(menu, iSubscribeTo);
IF MySelectionContainsSubscriberOrPublisher(isSubscriber) THEN
BEGIN {selection contains a single subscriber or publisher}
    IF isSubscriber THEN {selection contains a subscriber}
        SetMenuItemText(menu, iPubSubOptions, gSubOptText)
    ELSE
        {selection contains a publisher}
        SetMenuItemText(menu, iPubSubOptions, gPubOptText);
    EnableItem(menu, iPubSubOptions);
END
ELSE {selection contains either no subscribers or publishers }
    { or contains at least one subscriber and one publisher}
    DisableItem(menu, iPubSubOptions);
IF (gPubCount > 0) OR (gSubCount > 0) THEN
    EnableItem(menu, iShowHideBorders)
ELSE
    DisableItem(menu, iShowHideBorders);
END;

```

The procedure in Listing 3-21 adjusts the items in the Edit menu as appropriate for a document window of the application. The code enables the Undo command if the application can undo the last command, enables the Cut and Copy commands if there's a selection that can be cut or copied, enables the Paste command if there's text data in the

Menu Manager

scrap, and enables the menu items relating to publishers and subscribers appropriately, according to whether the current selection contains a publisher or subscriber. The application-defined `MySelectionContainsSubscriberOrPublisher` function returns `TRUE` if the current selection contains a single subscriber or a single publisher and returns `FALSE` otherwise. If the `MySelectionContainsSubscriberOrPublisher` function returns `TRUE`, the code sets the text for the Publisher Options (or Subscriber Options) command and enables the menu item. If the function returns `FALSE`, the code disables the Publisher Options (or Subscriber Options) command.

Determining if the User Chose a Keyboard Equivalent

Keyboard equivalents of commands allow the user to invoke a menu command from the keyboard. You can determine if the user chose the keyboard equivalent of a menu command by examining the event record for a key-down event. If the user pressed the Command key in combination with another 1-byte character, you can determine if this combination maps to a Command-key equivalent by using the `MenuKey` function.

If your application supports keyboard equivalents that use other modifier keys in addition to the Command key, your application should examine the `modifiers` field and take any appropriate action; depending on the modifier keys you use, your application may or may not be able to use `MenuKey` to map the key to the menu command.

Listing 3-22 shows an application-defined procedure, `DoEvent`, that determines whether a key-down event occurred and, if so, calls an application-defined routine to handle the key-down event.

Listing 3-22 Determining when a key is pressed

```
PROCEDURE DoEvent (event: EventRecord);
BEGIN
    CASE event.what OF
        keyDown, autoKey:      {handle keyboard events}
            DoKeyDown(event);
        {handle other events appropriately}
    END; {of CASE}
END;
```

If your application determines that the user pressed a key, you need to determine whether the user chose the keyboard equivalent of a menu command. You can do this by examining the `modifiers` field of the event record describing the key-down event. If the Command key was also pressed, then your application should call the `MenuKey` function. The `MenuKey` function scans the current menu list for a menu item that has a matching keyboard equivalent and returns the menu and menu item, if any. Although you should not define the same keyboard equivalent for more than one command, the `MenuKey` function scans the menus from right to left, scanning the items from top to bottom, and returns the first matching keyboard equivalent that it finds.

Menu Manager

If your application uses other keyboard equivalents in addition to Command-key equivalents, you can examine the state of the modifier keys and use the Event Manager function `KeyTranslate`, if necessary, to help map the keyboard equivalent to a particular menu item. See the discussion of 'KCHR' resources in *Inside Macintosh: Text* for information on how various keyboard combinations map to specific character codes.

Listing 3-23 shows an application's `DoKeyDown` procedure that handles key-down events and determines if a keyboard equivalent was pressed.

Listing 3-23 Checking a key-down event for a keyboard equivalent

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

Listing 3-23 extracts the pressed key from the message field of the event record and then examines the `modifiers` field to determine if the Command key was also pressed. If so, the application first adjusts its menus and then calls an application-defined procedure, `DoMenuCommand`. The `DoKeyDown` procedure passes as a parameter to the `DoMenuCommand` procedure the value returned from the `MenuKey` function.

Listing 3-24 shows the `DoMenuCommand` procedure.

Responding When the User Chooses a Menu Item

Your application can use the `MenuSelect` function to determine when the user chooses a menu command, and your application can use the `MenuKey` function to determine when the user presses the keyboard equivalent for a menu command. Both `MenuSelect` and `MenuKey` return a long integer value that indicates which menu and menu item the user chose.

The `MenuSelect` and `MenuKey` functions return the menu ID of the menu in the high word and the menu item number in the low word of their function result. If the user did not choose a menu command or if the user pressed a keyboard combination that does

Menu Manager

not map to any keyboard equivalent in your application's menus, the functions return 0 in the high word and the value of the low word is undefined. The `MenuSelect` function also returns 0 in the high word when the user selects an item in the Application or Keyboard menu. The `MenuSelect` function (and `MenuKey` function, if the command has a keyboard equivalent) returns the `kHMMHelpMenuID` constant in the high word and the menu item in the low word when the user selects an item that your application appended to the Help menu.

Listing 3-24 shows an application-defined procedure, `DoMenuCommand`. This procedure takes the appropriate action based on which menu command the user chose.

The `DoMenuCommand` procedure is called by the application after the application determines that either the user pressed the mouse button while the cursor was in the menu bar (in which case the application calls `MenuSelect` to allow the user to choose a command) or the user pressed the Command key and another key (in which case the application calls the `MenuKey` function). In either case, the application passes the function result returned by `MenuSelect` or `MenuKey` as a parameter to the `DoMenuCommand` procedure.

Listing 3-24 Responding to the user's choice of a menu command

```
PROCEDURE DoMenuCommand (menuResult: LongInt);
VAR
    menuID, menuItem: Integer;
BEGIN
    menuID := HiWord(menuResult);    {get menu ID of menu}
    menuItem := LoWord(menuResult);  {get menu item number}
    CASE menuID OF
        mApple:
            MyHandleAppleCommand(menuItem);
        mFile:
            MyHandleFileCommand(menuItem);
        mEdit:
            MyHandleEditCommand(menuItem);
        mFont:
            MyHandleFontCommand(menuItem);
        mSize:
            MyHandleSizeCommand(menuItem);
        kHMMHelpMenuID:
            MyHandleHelpCommand(menuItem);
        mOutline:
            MyHandleOutlineCommand(menuItem);
        mSubMenu: {user chose item from submenu}
            MyHandleSubLabelStyleCommand(menuItem);
    END; {end of CASE menuID}
    HiliteMenu(0); {unhighlight what MenuSelect or MenuKey hilited}
END;
```

Menu Manager

The `DoMenuCommand` procedure calls other application-defined routines to perform the requested action. After performing the action associated with the chosen menu item, your application should use the `HiliteMenu` procedure to unhighlight the menu title to indicate that the requested action is complete.

Handling the Apple Menu

When the user chooses an item from the Apple menu, the `MenuSelect` function returns the menu ID of your application's Apple menu in the high word and returns the chosen menu item in the low word of its function result.

If your application provides an About command as the first menu item in the Apple menu and the user chose this item, you should display your application's About box. Otherwise your application should use the `GetMenuItemText` procedure to get the menu item text and then call the `OpenDeskAcc` function, passing the text of the chosen menu item as a parameter.

Listing 3-25 shows an application-defined procedure, `MyHandleAppleCommand`, that the application calls in response to the user's choice of an item from the Apple menu.

Listing 3-25 Responding to the user's choice of an item from the Apple menu

```
PROCEDURE MyHandleAppleCommand (menuItem: Integer);
VAR
    itemName: Str255;
    daRefNum: Integer;
BEGIN
    CASE menuItem OF
        iAbout:           {bring up alert for About}
            DisplayMyAboutBox;
        OTHERWISE
            BEGIN {all non-About items in this menu are desktop items, }
                { for example, DA's, other apps, documents, etc.}
                GetMenuItemText(GetMenuHandle(mApple), menuItem,
                                itemName);
                daRefNum := OpenDeskAcc(itemName);
            END;
        END; {of CASE}
    END;
```

When the user chooses an item other than your application's About command from the Apple menu, your application should call the `OpenDeskAcc` function. The `OpenDeskAcc` function prepares to open the desktop object chosen by the user; for example, if the user chose a document created by the TeachText application, the `OpenDeskAcc` function schedules the TeachText application for execution (or prepares to open it if it isn't already open) and returns to your application. On your application's next call to `WaitNextEvent`, your application receives a suspend event, and then

Menu Manager

the Process Manager makes TeachText the foreground application and instructs TeachText to open the chosen document.

Handling the Help Menu

Both the `MenuSelect` and `MenuKey` functions return the `kHMHelpMenuID` constant (–16490) in the high word when the user chooses an appended item from the Help menu. The item number of the appended menu item is returned in the low word of the function result.

The `DoMenuCommand` procedure shown in Listing 3-24 determines which menu command was chosen by the user. If the user chose a command from the Help menu, the `DoMenuCommand` procedure calls the application-defined procedure `MyHandleHelpCommand`. Listing 3-26 shows the application-defined procedure `MyHandleHelpCommand`. This procedure illustrates how the SurfWriter application responds to the user's choice of an item from the application's Help menu. Note that you should use the `HMGetHelpMenuHandle` function, not the `GetMenuHandle` function, to get a handle to your application's Help menu.

Listing 3-26 Responding to the user's choice of a command from the Help menu

```
PROCEDURE MyHandleHelpCommand (menuItem: Integer);
VAR
    myHelpMenuHdl:      MenuHandle;
    origHelpItems, numItems: Integer;
    myErr:              OSErr;
BEGIN
    {get handle to your application's Help menu}
    myErr := HMGetHelpMenuHandle(myHelpMenuHdl);
    IF myErr <> noErr THEN
        EXIT(MyHandleHelpCommand);
    {count the number of items in the Help menu}
    numItems := CountMItems(myHelpMenuHdl);
    origHelpItems := numItems - kNumMyHelpItems;
    IF menuItem > origHelpItems THEN
        BEGIN {user chose an item added by this application}
            {adjust this application's global variables that hold item }
            { numbers of the menu items that this application appended}
            gMyHelpItem1 := origHelpItems + 1;
            gMyHelpItem2 := origHelpItems + 2;
            MyHelp(menuItem);
        END;
    END;
END;
```

Apple reserves the right to change the number of standard items in the Help menu. To determine the number of items in the Help menu, call the `CountMItems` function.

Handling a Size Menu

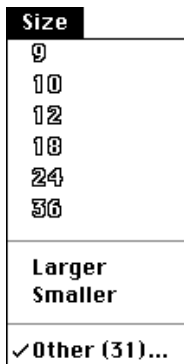
Your application can provide a Size menu to let the user choose various sizes of a font. Your Size menu should also provide the user with a method for specifying a size that isn't currently listed in the menu. For example, you can choose to provide an Other command that displays a dialog box allowing the user to choose a different font size. If the user chooses a font size not already in the menu, add a checkmark to the Other menu command and add the chosen size in parentheses to the text of the Other command.

Your application should outline font sizes to indicate which sizes are directly provided by the current font. For bitmapped fonts, outline only those sizes that actually exist in the Fonts folder. For TrueType fonts, outline all sizes that the TrueType font supports.

Your application should indicate the current font size to the user by placing a checkmark next to the text of the menu item that lists the current font size. If the current selection contains more than one font size, place a dash next to the name of each font size that the selection contains. ("Changing the Mark of Menu Items" on page 3-61 explains how to add marks to and remove marks from menu items.)

Figure 3-35 shows a Size menu as it appears after the user chooses a new font size of 31 by using the Other command. In Figure 3-35 the sizes 9, 10, 12, 18, 24, and 36 are the standard sizes provided by the application. Your application should place a checkmark next to the Other command to indicate that the current font size is a size other than a standard size. If the selection contains only one nonstandard size, include the size of the font in parentheses following the text Other. In Figure 3-35 the current selection contains a nonstandard size of 31, so the application places the checkmark next to the Other command and includes 31 in parentheses following the Other text. If the selection contains multiple nonstandard sizes, include the text Mixed in parentheses following the word Other. If the selection contains one or more standard sizes and only one nonstandard size, place a dash next to each standard size that the selection contains and place a dash next to the Other command with the nonstandard size included in parentheses in the text of the Other command.

Figure 3-35 A Size menu with user-specified size added



Menu Manager

When the user chooses the Other command, you should display the current font size in a dialog box and allow the user to choose a new size. Figure 3-16 on page 3-28 shows a sample dialog box an application might display in response to the user's choice of the Other command.

You should always specify the text of the Other command in the plain font style (as shown in Figure 3-35) and never outlined, regardless of whether the current font is a TrueType font that supports that size or a bitmapped font that exists at that size in the Fonts folder.

Listing 3-27 shows an application-defined procedure that handles the user's choice of an item in the Size menu shown in Figure 3-35.

Listing 3-27 Handling the Size menu

```
PROCEDURE MyHandleSizeCommand (menuItem: Integer);
VAR
    numItems:      Integer;
    addItem:      Boolean;
    itemString:    Str255;
    itemStyle:    Style;
    sizeChosen:    LongInt;
BEGIN
    numItems := CountMItems(GetMenuHandle(mSize));
    IF menuItem = numItems THEN
        BEGIN
            {user chose Other command}
            {display a dialog box to allow the user to choose any }
            { size. If the user-specified size is not in the menu, }
            { add a checkmark to the Other command and add the }
            { new font size to the text of the Other command}
            MyDisplayOtherBox(sizeChosen);
        END
    ELSE
        BEGIN
            IF (menuItem = (numItems - 2)) OR
               (menuItem = (numItems - 3)) THEN
                DoMakeLargerOrSmaller(menuItem, sizeChosen)
            ELSE
                BEGIN
                    {user chose size displayed in the menu}
                    {remove checkmark or dashes from menu items showing }
                    { previous size}
                    MyRemoveMarksFromSizeMenu;
                    {add checkmark to menu item of new current size}
                    CheckItem(GetMenuHandle(mSize), menuItem, TRUE);
                    sizeChosen := MyItemToSize(menuItem);
                END;
            END;
        END;
    END;
```

Menu Manager

```

        {update the document's state or the user's selection as needed}
        MyResizeSelection(sizeChosen);
    END;

```

If the user chooses an item from the Size menu, the `MyHandleSizeCommand` procedure first counts the current number of items in the menu. If the user chooses the last item in the menu (the Other command), the procedure displays a dialog box like the one shown in Figure 3-16 on page 3-28 to let the user choose a size other than the ones currently shown in the menu. The application-defined function `MyDisplayOtherBox` also adds a checkmark to the Other command if the user chose a new size, adds the new size to the text of the Other command, and returns the chosen size in the `sizeChosen` variable.

If the user chose the Larger or Smaller command from the Size menu, the code calls an application-defined routine, `DoMakeLargerOrSmaller`, to perform the requested action. The `DoMakeLargerOrSmaller` procedure also adds a checkmark and adds the new size to the text of the Other command if the new size does not match any size in the menu. The procedure returns the chosen size in the `sizeChosen` variable.

If the user chose any size currently displayed in the menu, the `MyHandleSizeCommand` procedure adjusts the marking character of the menu items appropriately. The code removes the checkmark from the previous menu item and adds a checkmark to the menu item representing the new size chosen by the user. The code uses an application-defined function, `MyItemToSize`, to map the item number of the chosen menu item to a given size and returns this size in the `sizeChosen` variable.

The code then uses the application-defined procedure `MyResizeSelection` to update the document's state and resize the user's selection, if any, to the chosen size.

Accessing Menus From a Dialog Box

In System 7, the Menu Manager or your application can allow the user to access selected menus in the menu bar while interacting with an alert box or a modal dialog box. This allows users to make menu selections while your application is displaying an alert box or a modal dialog box. For example, a user might want to turn on Balloon Help for assistance in figuring out how to respond to an alert box. Similarly, if the modal dialog box contains several editable text fields, the user might find it simpler to copy text from one text field and paste it into another. Figure 3-36 shows a modal dialog box with an editable text field. Note that only the Edit and Help menus are enabled and all other menus are disabled. This gives the user access to editing commands and also to Balloon Help.

Note

In System 6, user access to menus in the menu bar is prohibited from an alert box or a modal dialog box unless your application specifically allows it. For example, in System 6, your application must provide a filter procedure to replace the standard filter procedure if you want to support the keyboard equivalents of the standard Edit menu commands in a modal dialog box. In System 7, you can let the Menu Manager enable these commands for you. ♦

Figure 3-36 Menu access from a modal dialog box

When your application displays a modeless or movable modal dialog box, your application should adjust its menus as appropriate for that dialog box. For example, when a movable modal dialog box is the frontmost window, your application should enable the Apple menu, enable the Edit menu if your dialog box contains an editable text item, enable or disable any other menus as needed, and disable any items it added to the Help menu if the user can't perform those actions while the dialog box is displayed.

When your application displays an alert box, system software automatically disables all of your application's menus except for the Help menu (in which all items are disabled except for the Show Balloons/Hide Balloons command).

When your application displays a modal dialog box, your application should also enable and disable its menus as appropriate. For example, you should enable the Edit menu if your dialog box contains an editable text item and disable any items it added to the Help menu if the user can't perform those actions while the dialog box is displayed. If your application handles access to the menu bar from a modal dialog box, it should disable the Apple menu or the first item in the Apple menu.

If your application does not specifically handle access to the menu bar from an alert box or a modal dialog box, in some cases the Menu Manager automatically disables the appropriate menus for you, as described in the following paragraphs.

When your application displays an alert box or a modal dialog box (that is, a window of type `dBoxProc`), the Menu Manager (in conjunction with the Dialog Manager) always appropriately adjusts the system-handled menus and performs these actions:

1. Disables all menu items in the Help menu except the Show Balloons (or Hide Balloons) command, which it enables.
2. Disables all menu items in the Application menu.
3. Enables the Keyboard menu if it appears in the menu bar, except for the About Keyboards command, which it disables.

Menu Manager

In addition, if your application then calls the `ModalDialog` procedure, the Menu Manager (in conjunction with the Dialog Manager) performs two other actions:

4. Disables all of your application's menus.
5. Enables commands with the standard keyboard equivalents Command-X, Command-C, and Command-V if the modal dialog box contains a visible and active editable text field. The user can then use either the menu commands or their keyboard equivalents to cut, copy, and paste text. (The menu item having keyboard equivalent Command-X must be one of the first five menu items.)

When the user dismisses the modal dialog box, the Menu Manager restores all menus to the state they were in prior to the appearance of the modal dialog box.

In some cases actions 4 and 5 do not occur when you call `ModalDialog`. The enabling and disabling described in steps 4 and 5 do not occur if any of these conditions is true:

- Your application does not have an Apple menu.
- Your application has an Apple menu, but the menu is disabled when the modal dialog box is displayed.
- Your application has an Apple menu, but the first item in that menu is disabled when the dialog box is displayed.

Note

If your application already handles access to the menu bar from a modal dialog box and you do not want the automatic menu enabling and disabling provided by System 7 to occur, you should ensure that one or more of those conditions is true when you display a modal dialog box. ♦

When your application displays alert boxes or modal dialog boxes with no editable text items, your application can allow system software to handle menu bar access. In all other cases, your application should handle its own menu bar access.

System software always leaves the Help, Keyboard, and Application menus and their commands available when you display movable modal dialog boxes and modeless dialog boxes. For these types of dialog boxes, you must disable menus as appropriate and handle menu bar access as appropriate given their contents.

When your application displays a movable modal dialog box (a window of type `movableDialogProc`), your application does not need to adjust the system-handled menus but should disable all its other menus except the Apple menu and—if your movable modal dialog box contains editable text items—the Edit menu. Leave the Apple menu enabled so that the user can use it to open other applications, and leave the Edit menu enabled so that the user can use the Cut, Copy, and Paste commands within the editable text item. (You can also leave your Undo and Clear commands enabled; otherwise, disable all other commands in the Edit menu.)

When your application removes a movable modal dialog box, modeless dialog box, or modal dialog box with editable text items, your application must restore to their previous states any menus that it disabled prior to displaying the dialog box. See the chapter “Dialog Manager” in this book for additional information on dialog boxes.

Writing Your Own Menu Definition Procedure

The Menu Manager uses the menu definition procedure and menu bar definition function to display and perform basic operations on menus and the menu bar. The menu definition procedure performs all the drawing of menu items within a menu and performs all the actions that might differ between one type of menu and another. The menu bar definition function draws the menu bar and performs most of the drawing activities related to the display of menus when the user moves the cursor between menus.

Apple provides a standard menu bar definition function, stored as a resource in the System file. The standard menu bar definition procedure is the 'MBDF' resource with resource ID 0. When you create your menus and menu bar, by default the Menu Manager uses the standard menu bar definition function to manage them. Although the Menu Manager lets you provide your own menu bar definition function, Apple recommends that you always use the standard menu bar definition function.

The Menu Manager uses the standard menu bar definition function to

- draw the menu bar
- clear the menu bar
- determine if the cursor is in the menu bar or any currently displayed menus
- calculate the left edges of menu titles
- highlight a menu title
- invert the entire menu bar
- erase the background color of a menu and draw the menu's structure (shadow)
- save or restore the bits behind a menu

Apple provides a standard menu definition procedure, stored as a resource in the System file. The standard menu definition procedure is the 'MDEF' resource with resource ID 0. The standard menu definition procedure handles three types of menus: pull-down, pop-up, and hierarchical; it also implements scrolling in menus. When you define your menus, you specify the menu definition procedure that the Menu Manager should use when managing them. You'll usually want to use the standard definition procedure for your application. However, if you need a feature not provided by the standard menu definition procedure (for example, if you want to include more graphics in your menus), you can write your own menu definition procedure.

The Menu Manager uses the standard menu definition procedure to

- calculate a menu's dimensions
- draw the menu items in a menu
- highlight and unhighlight menu items as the user moves the cursor between them
- determine which item the user chose from a menu

Menu Manager

If you provide your own menu definition procedure, it should also perform these tasks. Your menu definition procedure should also support scrolling in menus and color in menus and provide support for Balloon Help.

If you provide your own menu definition procedure, store it in a resource of type 'MDEF' and include its resource ID in the description of each menu that uses your own menu definition procedure. If you create a menu using `GetMenu` (or `GetNewMBar`), the Menu Manager reads the menu definition procedure into memory and stores a handle to it in the `menuProc` field of the menu's menu record.

When your application uses `GetMenu` (or `GetNewMBar`) to create a new menu that uses your menu definition procedure, the Menu Manager creates a menu record for the menu and fills in the `menuID`, `menuProc`, `enableFlags`, and `menuData` fields according to the menu's resource description. The Menu Manager also reads in the data for each menu item and stores it as variable data at the end of the menu record. The menu definition procedure is responsible for interpreting the contents of the data. For example, the standard menu definition procedure interprets this data as described in "The Menu Resource" beginning on page 3-151. After reading in a resource description of a menu, the Menu Manager requests the menu definition procedure to calculate the size of the menu and to store these values in the `menuWidth` and `menuHeight` fields of the menu's menu record.

Note that when drawing a menu, the Menu Manager first requests your menu definition procedure to calculate the dimensions (the menu rectangle) of the menu. Next the Menu Manager requests the menu bar definition function to draw the structure (shadow) of the menu and erase the contents of the menu to its background color. Then the Menu Manager requests your menu definition procedure to draw the items in the menu. As the user moves the cursor into and out of menu items, the Menu Manager requests your menu definition procedure to highlight and unhighlight items appropriately. Your menu definition procedure should also determine when to add scrolling indicators to a menu and scroll the menu appropriately when the cursor is in a scrolling item. Your menu definition is responsible for showing and removing any help balloons associated with a menu item.

When the Menu Manager requests your menu definition procedure to perform an action on a menu, it provides your procedure with a handle to its menu record. This allows your procedure to access the data in the menu record and to use any data in the variable data portion of the menu record to appropriately handle the menu items. However, your menu definition procedure should not assume that the A5 register is properly set up, so your procedure can't refer to any of the QuickDraw global variables.

The Menu Manager passes a value to your menu definition procedure in the `message` parameter that indicates the action your menu definition procedure should perform. The Menu Manager always passes a handle to the menu record of the menu that the operation should affect in the parameter `theMenu`. Depending on the requested action, the Menu Manager passes additional information in other parameters.

Listing 3-28 shows how you might declare a menu definition procedure.

Listing 3-28 A sample menu definition procedure

```
PROCEDURE MyMDEF (message: Integer; theMenu: MenuHandle;
                 VAR menuRect: Rect; hitPt: Point;
                 VAR whichItem: Integer);

{any support routines used by the main program of your MDEF }
{ go here}

BEGIN
    CASE message OF
        mDrawMsg:
            MyDrawMenu(theMenu, menuRect);
        mChooseMsg:
            MyChooseItem(theMenu, menuRect, hitPt, whichItem);
        mSizeMsg:
            MySizeTheMenu(theMenu);
        mPopUpMsg:
            MyCalcMenuRectForOpenPopUpBox(theMenu, hitPt, menuRect);
    END;
END;
```

The next sections describe in more detail how your menu definition procedure should respond when it receives the `mDrawMsg`, `mChooseMsg`, or `mSizeMsg` constant in the message parameter. For a complete description of the menu definition procedure and the parameters passed to your procedure by the Menu Manager, see “The Menu Definition Procedure” beginning on page 3-148.

Calculating the Dimensions of a Menu

Whenever the Menu Manager creates a menu or needs to calculate the size of a menu that is managed by your menu definition procedure, the Menu Manager calls your procedure and specifies the `mSizeMsg` constant in the message parameter, requesting that your procedure calculate the size of the menu.

Listing 3-29 on page 3-90 shows an application-defined support routine, `MySizeTheMenu`, used by the application’s menu definition procedure. After calculating the height and width of the menu’s rectangle, the menu definition procedure stores the values in the `menuWidth` and `menuHeight` fields of the menu’s menu record.

Listing 3-29 Calculating the size of a menu

```

PROCEDURE MySizeTheMenu(theMenu: MenuHandle);
VAR
    itemDataPtr:    Ptr;
    numItems:       Integer;
BEGIN
    HLock(Handle(theMenu));
    WITH theMenu^^ DO
    BEGIN {menuData points to title of menu and additional item data}
        itemDataPtr := @menuData;
        {skip past the menu title}
        itemDataPtr := POINTER(ORD4(itemDataPtr)+ itemDataPtr^ +1);
    END;
    numItems := CountMItems(theMenu);
    {calculate the height of the menu--each item's height can vary }
    { according to whether the item has an icon or a script code defined. }
    { The height of the menu should not exceed the height of the }
    { screen minus the menu bar height. }
    { Store the height in the menu's menu record}
    theMenu^^.menuHeight := MyCalcMenuHeight(itemDataPtr, numItems);

    {calculate the width of the menu (the width of the longest item): }
    { for each item calculate the width as }
    { width = iconWidth + markWidth + textWidth + subMenuWidth }
    {      + cmdKeyComboWidth }
    { If an item doesn't have a characteristic, use 0 as the width of }
    { that characteristic. }
    { To calculate the width of item's text, must consider script code and }
    { width of the font. }
    { The width of the menu should not exceed the right or left }
    { boundaries of the screen. }
    { Store the width in the menu's menu record}
    theMenu^^.menuWidth := MyCalcMenuWidth(itemDataPtr, numItems);
    HUnlock(Handle(theMenu));
END;

```

Drawing Menu Items in a Menu

Whenever the user presses the mouse button while the cursor is in the menu title of a menu managed by your menu definition procedure, the Menu Manager calls the menu bar definition function to highlight the menu title, draw the structure of the menu, and erase the contents of the menu to its background color. The Menu Manager then calls your menu definition procedure and specifies the `mDrawMsg` constant in the message

Menu Manager

parameter, requesting that your procedure draw the menu items. When your menu definition procedure receives this constant, it should draw the menu items of the menu specified by the parameter `theMenu` inside the rectangle specified by the `menuRect` parameter. The Menu Manager sets the current graphics port to the Window Manager port before calling your menu definition procedure. Your menu definition procedure can determine how to draw the menu items by examining the data in the menu record.

If your menu definition procedure supports color menus, your procedure should check the application's menu color information table for the colors to use to draw each item. If the application's menu color information table contains a color entry for an item, draw the item using that color. If the table does not contain an item entry for a particular item, use the default item color defined in the menu title entry. If a menu title entry doesn't exist, use the default item color defined in the menu bar entry. If the menu bar entry doesn't exist, draw the item using black on white.

If your menu definition procedure supports scrolling menus, it should insert scrolling indicators if necessary when drawing the menu items.

Listing 3-30 shows an application-defined support routine, `MyDrawMenu`, used by the application's menu definition procedure. The `MyDrawMenu` procedure draws each item in the menu, according to the item's defined characteristics. Disabled items should be drawn using the colors returned by the `GetGray` function. Pass the RGB color of the item's background in the `bkgnd` parameter to the `GetGray` function; pass the RGB color of the item's enabled text in the `fgnd` parameter. The `GetGray` function returns `TRUE` if there's an available color between the two specified colors and returns in the `fgnd` parameter the color in which you should draw the item.

Listing 3-30 Drawing menu items

```
PROCEDURE MyDrawMenu(theMenu: MenuHandle; menuRect: Rect);
VAR
    numItems:      Integer;
    itemRect:      Rect;
    item:          Integer;
    currentOffset: LongInt;
    nextOffset:    LongInt;
BEGIN
    numItems := CountMItems(theMenu);
    currentOffset := 0;
    nextOffset := 0;
    FOR item := 1 TO numItems DO
        BEGIN
            {calculate the enclosing rectangle for this item}
            itemRect := MyCalcItemRect(item, menuRect, currentOffset, nextOffset);
            {draw the item--index into the item-specific data from the menu record }
            { to get the characteristics of this menu item and draw the item }
```

Menu Manager

```

{ according to its defined characteristics. For example, draw the item's }
{ text in its defined style & font of its defined script, draw any icon, }
{ mark, submenu indication, or keyboard equivalent, and draw each }
{ characteristic of the item according to its color entry in the menu's }
{ menu color information table. }
{ Draw disabled items in gray--use the GetGray function to return the }
{ appropriate color. Also draw dividers using the gray color }
{ returned by GetGray}
    MyDrawTheItem(item, itemRect, menuRect, currentOffset);
END;
{if your menu supports scrolling, insert scrolling indicators if needed}
MyInsertScrollingArrows(menuRect);
END;

```

Determining Whether the Cursor Is in an Enabled Menu Item

Whenever the user drags the cursor into or out of a menu item of a displayed menu managed by your menu definition procedure, the Menu Manager calls your procedure and specifies the `mChooseMsg` constant in the message parameter, requesting that your procedure determine whether the cursor is in a menu item and that your procedure highlight or unhighlight the menu item as appropriate. When your menu definition procedure receives this constant, it should use the menu rectangle specified in the `menuRect` parameter, the mouse location specified in the `hitPt` parameter, and the item number specified in the `whichItem` parameter to determine the proper action to take.

To see whether the user chose an enabled item, your menu definition procedure should determine whether the specified mouse location is inside the rectangle specified by the `menuRect` parameter, and, if so, it should check whether the menu is enabled. If the menu is enabled, your menu definition procedure should determine whether the mouse location specified in the `hitPt` parameter is in an enabled menu item.

If the mouse location is in an enabled menu item, your menu definition procedure should unhighlight the item specified by the `whichItem` parameter, highlight the new item, and return the new item number in `whichItem`.

If the mouse location isn't in an enabled menu item, your menu definition procedure should unhighlight the item specified by the `whichItem` parameter and return 0 in the `whichItem` parameter.

When your menu definition procedure draws a menu item in its highlighted state in a color menu, it should reverse the background color and the item color and then draw the menu item. When your menu definition procedure needs to return a menu item to its normal (unhighlighted) state, it should reset the background color and item color of that menu item and draw the menu item.

If your menu definition procedure supports scrolling menus, it should scroll the menu when the user moves the cursor into the area of the indicator, or when the cursor is directly above or below the menu. If the user can scroll the menu up (by dragging the

Menu Manager

cursor past the last item to view more items), place a downward-pointing triangular indicator in place of the last item in the menu. If the user can scroll the menu down (by dragging the cursor past the first item to view the items originally at the top of the menu), place an upward-pointing triangular indicator in place of the first item in the menu.

For all menus, your menu definition procedure should set the global variable `MenuDisable` appropriately each time a new item is highlighted. Set `MenuDisable` to the menu ID and item number of the last menu item chosen, whether or not it's disabled. The `MenuChoice` function uses the value in `MenuDisable` to determine if a chosen menu item is disabled.

Listing 3-31 shows an application-defined support routine, `MyChooseItem`, used by the application's menu definition procedure. This routine determines which item, if any, the point specified by the `hitPt` parameter is in. If the item is in an enabled menu item that is different from the previous item, the `MyChooseItem` procedure unhighlights the old item and highlights the new item. However, the `MyChooseItem` procedure does not highlight the new item if the item is in a divider or disabled item.

The procedure also removes any help balloons as appropriate and, if Balloon Help is turned on, displays any help balloon of the new item (for any item other than a divider or scrolling indicator). The `MyChooseItem` procedure returns the item number of the new item in the `whichItem` parameter or returns 0 if no item is chosen. Although not shown in the listing, if the item is a disabled item, the procedure returns 0 in the `whichItem` parameter and sets the `MenuDisable` global variable to the menu ID and item number of the disabled item.

Listing 3-31 Choosing menu items

```
PROCEDURE MyChooseItem (theMenu: MenuHandle; menuRect: Rect; hitPt: Point;
                        VAR whichItem: Integer);
VAR
    oldWhichItem:      Integer;
    MenuChoicePtr:     ^LongInt;
    numItems, item, max: Integer;
    itemChosen:        Integer;
    inScroll:          Integer;
    currentOffset:     LongInt;
    nextOffset:        LongInt;
BEGIN
    oldWhichItem := whichItem;
    whichItem := 0;
    itemChosen := 0;
    MenuChoicePtr := POINTER(kLowMemMenuDisable);
    numItems := CountMItems(theMenu);
    {find out whether the hitPt is in an item's rectangle, and if so, }
    { determine which item }
```

Menu Manager

```

item := 1;
max := numItems + 1;
currentOffset := 0;
nextOffset := 0;
REPEAT
    itemRect := MyCalcItemRect(item, menuRect, currentOffset, nextOffset);
    IF PtInRect(hitPt, itemRect) THEN {hitPt is in this item}
        itemChosen := item;
        item := item + 1;
UNTIL (item = MAX) OR (itemChosen <> 0);
IF itemChosen = 0 THEN
BEGIN {the mouse isn't in any item of this menu;unhighlight previous item}
    MyNotInMenu(menuRect, oldWhichItem);
END
ELSE
BEGIN {the mouse is in this menu item. }
    { First see if a previous item was highlighted}
    IF ((oldWhichItem <> 0) AND (oldWhichItem <> itemChosen)) THEN
    BEGIN
        {a previous item was highlighted--unhighlight it}
        itemRect := MyCalcOldItemRect(oldWhichItem, menuRect);
        IF HMGetBalloons THEN {if Balloon Help is on then }
            HMRemoveBalloon;{ remove any balloon that might be showing}
        MyHighlightItem(itemRect, oldWhichItem, FALSE);
    END;
    IF HMGetBalloons and MyIsItemDivider(itemChosen) THEN
        {Balloon Help is on and item is divider}
        HMRemoveBalloon;{remove any balloon that might be showing}
    IF MyIsItemEnabled(itemChosen) THEN
    BEGIN
        {the item is enabled, so highlight the item the cursor is in}
        itemRect := MyCalcNewItemRect(itemChosen, menuRect, currentOffset);
        {the highlighting routine must also support scrolling correctly }
        { (if the cursor is in a scrolling item, don't highlight the item)}
        inScroll := MyIsScrollItem(itemChosen);
        MyHighlightItem(itemRect, itemChosen, inScroll);
        IF HMGetBalloons AND inScroll THEN
            HMRemoveBalloon {remove any balloon that might be showing}
        ELSE
        BEGIN {display help balloon for this item, if any}
            IF HMGetBalloons THEN

```

Menu Manager

```

BEGIN
    IF StillDown THEN {mouse button is still down in this item}
        {this routine sets up the needed parameters and then }
        { calls HMShowMenuBalloon}
        MyShowMenuBalloon(itemChosen, itemRect);
    END;
END;
END;
END;
END;

```

Menu Manager Reference

This section describes the data structures and routines of the Menu Manager. It also describes various resources, including the resources you can use to create your menus and menu bar, the 'MBAR' and 'MENU' resources.

Data Structures

This section describes the menu record, menu list, and menu color information table. The Menu Manager maintains information about the menus in your application in menu records. The Menu Manager maintains information about all the menus in a menu bar in a data structure called the *menu list*.

The Menu Manager stores color information about your application's menus in a menu color information table. You can add entries to your application's menu color information table if you want to use colors other than the default colors for your menu bar or menus. You can add entries to this table by using the `SetMCEntries` procedure or by providing 'mctb' resources.

The Menu Record

A menu record contains information about a single menu. Your application should never manipulate or access the fields of a menu record; instead your application should use Menu Manager routines to create and manage the menus in your application. To refer to a menu, use a handle to the menu's menu record.

The `MenuInfo` data type defines the menu record. The `MenuHandle` data type is a handle to a menu record.

```

TYPE  MenuPtr      = ^MenuInfo;    {pointer to a menu record}
      MenuHandle   = ^MenuPtr;     {handle to a menu record}

```