

This chapter describes how your application can use the Menu Manager to create and manage menus. Menus provide a simple and standard method for the user to view or choose from a list of commands and settings that your application provides. Every Macintosh application that has a user interface should support pull-down menus (that is, menus that the user “pulls down” by pressing the mouse button when the cursor is over the menu title in the menu bar).

This chapter shows how to

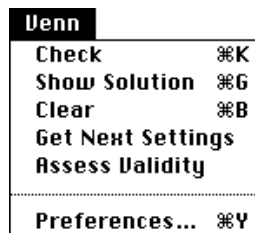
- create menu and menu bar resources
- open those resources to display the menu bar
- handle user clicks in the menu bar
- handle user choices of menu items
- handle keyboard equivalents of menu commands
- enable and disable menu items

Most Macintosh applications provide more menu handling than is illustrated in this chapter. For example, you might want to use pop-up menus in a window or dialog box. For a complete description of the capabilities of the Menu Manager and for code samples illustrating more advanced menu-handling techniques, see the chapter “Menu Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

About Menus

A *menu* is a user interface element that your application can create to allow the user to view or choose an item from a list of commands and options that your application provides. For example, the sample application Venn Diagrammer provides a menu (shown in Figure 8-1) that contains a list of commands for manipulating Venn diagrams.

Figure 8-1 A typical pull-down menu



This kind of menu is known as a *pull-down menu*, because the user “pulls down” the menu by clicking the *menu title* (the word “Venn” in the menu bar). A pull-down menu always has associated with it one or more *menu items*, rectangles containing text and other characteristics that identify a command that the user can choose to perform an

Menus

action. The menu shown in Figure 8-1 contains six menu items and one *divider* (the gray line used to separate the first five items from the last one). In addition, four of the menu items in that menu have *keyboard equivalents* associated with them. The user can invoke the menu command by pressing the appropriate combination of characters on the keyboard. For example, the user can make the Preferences dialog box appear by pressing the combination Command-Y.

Note

This chapter shows how to create and handle pull-down menus only. The word “menu” should therefore be understood to mean “pull-down menu.” ♦

The Menu Manager provides routines that allow you to create your application’s *menu bar* and menus, and to handle user actions in the menu bar and in individual menus. You’ll call these routines when you detect that a mouse-down event has occurred in the menu bar or when you detect that the user has typed a keyboard equivalent of a menu command. You’ll also call the Menu Manager to perform other operations on menus, such as changing menu item text or enabling and disabling menu items.

All Macintosh applications should support at least three standard menus: the Apple menu, the File menu, and the Edit menu. In addition, you’ll want to support other menus that contain commands and options specific to your application. The Venn Diagrammer application supports only one application-specific menu along with the three standard menus.

Creating Menus

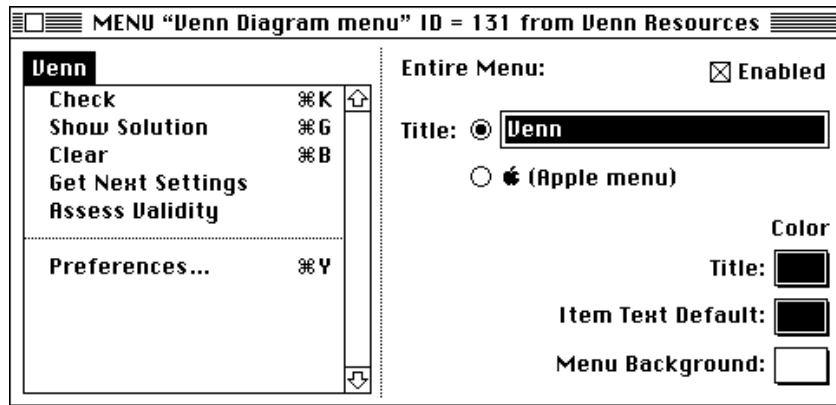
The easiest way to define menu titles and commands is to use a resource editor like ResEdit to create resources describing your application’s menu bar and the individual menus. It’s also possible to define your menu bar and menu items internally in your application, but you can make your application significantly easier to localize by isolating that information in resources.

Note

As you learned in the chapter “Resources,” you can also create resources using the Rez resource-description language and a resource compiler. This chapter shows how to use ResEdit to create menu-related resources. ♦

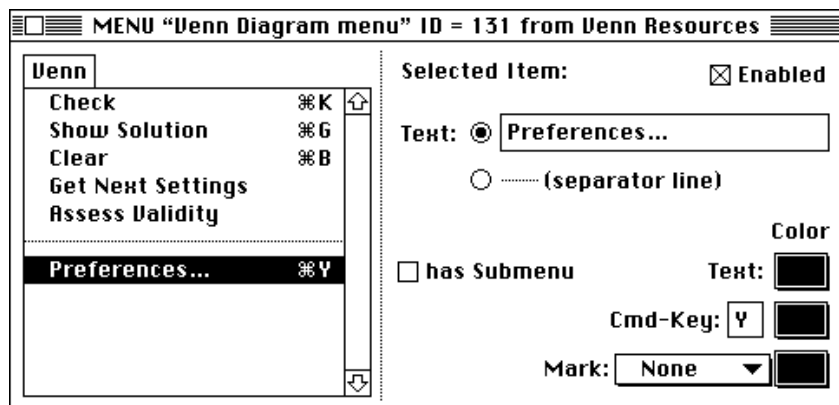
Creating a Menu Resource

You can define the menu title and characteristics of each individual menu item in a *menu resource* (a resource of type ‘MENU’). Figure 8-2 shows the appearance of ResEdit’s ‘MENU’ resource editor.

Figure 8-2 Defining a 'MENU' resource

As you can see, the menu title is currently selected. ResEdit allows you to change the menu title text or to designate this menu as the Apple menu. This window also lets you set the menu as initially enabled or disabled. In most cases, you'll want to have your menus initially enabled. The Venn Diagrammer application, however, disables the Edit menu because it does not support any text editing.

To edit the text of a menu command, you can click it. ResEdit highlights the selected command and changes the controls in the right side of the window, as shown in Figure 8-3.

Figure 8-3 Editing a menu command

You can use the controls in the right side of the window to change the menu item text, the keyboard equivalent, the menu's mark, and several other items. You can also designate the menu item as initially enabled or disabled. Once again, you'll probably want most items to be initially enabled. You can disable and reenablen menu items

Menus

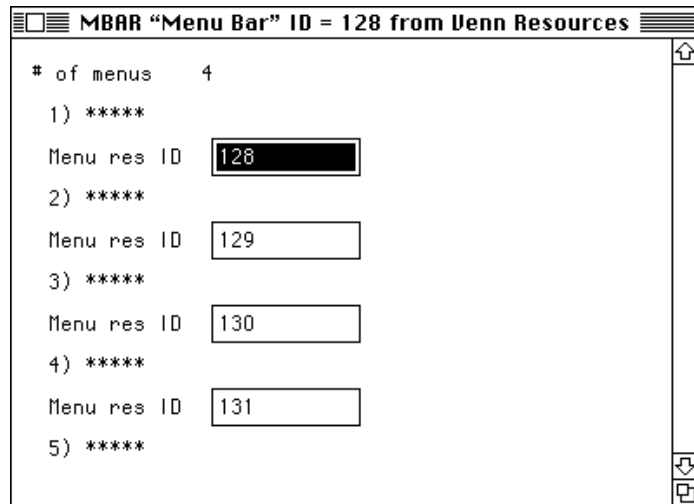
dynamically during your application's execution; see "Handling Menu Choices" beginning on page 156 for details.

Creating a Menu Bar Resource

You can define the order and resource IDs of the menus in your application in a *menu bar resource* (a resource of type 'MBAR'). You should define your 'MBAR' resource in such a way 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 can use ResEdit to create an 'MBAR' resource. Figure 8-4 shows the 'MBAR' resource window for the Venn Diagrammer application.

Figure 8-4 An 'MBAR' resource in ResEdit



An 'MBAR' resource is simply a list of the *menu IDs*, in the order you want the corresponding menu titles to appear from left to right in the menu bar.

Setting Up the Menu Bar and Menus

One of the very first things you need to do when your application starts running is set up your menu bar and menus. You can do this by calling the Menu Manager function `GetNewMBar`, which reads a specified 'MBAR' resource from your application's resource

Menus

fork and inserts each menu described there into the menu bar. You can define a constant that indicates which 'MBAR' resource to open.

```
CONST
    rMenuBar = 128;                {menu bar resource ID}
```

Listing 8-1 shows a standard way to call `GetNewMBar`.

Listing 8-1 Setting up the menu bar and menus

```
PROCEDURE DoSetupMenus;
    VAR
        menuBar:    Handle;
BEGIN
    menuBar := GetNewMBar(rMenuBar);
    IF menuBar = NIL THEN
        DoBadError(eCantFindMenus);

        SetMenuBar(menuBar);
        DisposeHandle(menuBar);
        AppendResMenu(GetMenuHandle(mApple), 'DRVR');
        DrawMenuBar;
    END;
```

The routine `DoSetupMenus` creates the application's menu bar by reading in the definition from the 'MBAR' resource with resource ID `rMenuBar`. The `GetNewMBar` function returns a handle to the menu bar information stored in that resource and in the 'MENU' resources whose IDs are contained in the 'MBAR' resource. Notice that `DoSetupMenus` makes sure that the value of the returned handle isn't `NIL`; if it is, you shouldn't continue.

Note

Checking that `GetNewMBar` returns handle with a non-`NIL` value is probably overkill. It's extremely unlikely that the Menu Manager will have a problem reading your menu-related resources or finding enough free memory to hold the menu list to which `menuBar` is a handle. Nonetheless, it's best to make sure, because passing `AppendResMenu` a handle whose value is `NIL` is likely to cause your application to crash. As a result, `DoSetupMenus` calls the application-defined routine `DoBadError` (defined in Listing 9-5 on page 178) to alert the user of the problem and terminate the application. If the application can't even put up its menu bar, there's no point in continuing to run. (See Figure 7-2 on page 134 for the alert box displayed if the menu resources can't be found.) ♦

Menus

If `GetNewMBar` returns a handle with a non-NIL value, then `DoSetupMenus` calls the procedure `SetMenuBar` to install the individual menus into the menu bar. At that point, you no longer need the handle and you can dispose of it (by calling the Memory Manager routine `DisposeHandle`). Next `DoSetupMenus` calls the `AppendResMenu` procedure to add the items in the Apple Menu Items folder to the Apple menu. Finally, the `DoSetupMenus` procedure displays the menu bar by calling the `DrawMenuBar` procedure.

Handling Menu Choices

Your application is informed of user menu choices in a slightly roundabout fashion. First, your application receives a mouse-down event indicating that the user has clicked in the menu bar. At that time, you should call the Menu Manager function `MenuSelect` to determine which menu and menu item, if any, the user chose. When you call `MenuSelect`, the Menu Manager pulls down the appropriate menu and tracks all subsequent mouse movement in the menu. When the user releases the mouse button, `MenuSelect` exits and returns to your application a long integer that indicates which menu and item the user chose. The high-order word of that long integer contains the menu number, and the low-order word contains the menu item number.

To coordinate the menu numbers and menu item numbers with the menus and menu items as defined in your 'MBAR' and 'MENU' resources, you'll probably want to define a set of constants, as shown in Listing 8-2.

Listing 8-2 Defining menu numbers and menu item numbers

```
CONST
    mApple      = 128;           {resource ID of Apple menu}
    iAbout      = 1;             {our About... dialog}

    mFile       = 129;           {resource ID of File menu}
    iNew        = 1;
    iClose      = 2;
    iQuit       = 4;

    mEdit       = 130;           {resource ID of Edit menu}
    iUndo       = 1;
    iCut        = 3;
    iCopy       = 4;
    iPaste      = 5;
    iClear      = 6;

    mVenn       = 131;           {resource ID of Venn menu}
```

Menus

```

iCheckVenn      = 1;
iDoVenn         = 2;
iClearVenn      = 3;
iNextTask       = 4;
iCheckArg       = 5;
iGetVennPrefs   = 7;

```

Note

The divider in a menu counts as a menu item, even though the user can't choose it. ♦

In general, you'll define a routine like `DoMenuCommand` shown in Listing 8-3 to handle all menu choices. Both your mouse-down event handler (Listing 6-9 on page 121) and your key-down event handler (Listing 8-5 on page 160) call `MenuSelect`. It is passed either the result of `MenuSelect` (for menu selections) or `MenuKey` (for keyboard equivalents of menu selections).

Listing 8-3 Handling menu selections

```

PROCEDURE DoMenuCommand (menuAndItem: LongInt);
VAR
    myMenuNum: Integer;
    myItemNum: Integer;
    myResult: Integer;
    myDAName: Str255;
    myWindow: WindowPtr;
BEGIN
    myMenuNum := HiWord(menuAndItem);
    myItemNum := LoWord(menuAndItem);
    GetPort(myWindow);

    CASE myMenuNum OF
        mApple:
            CASE myItemNum OF
                iAbout:
                    BEGIN
                        DoAboutBox;
                    END;
                OTHERWISE
                    BEGIN
                        GetMenuItemText(GetMenuHandle(mApple), myItemNum,
                                      myDAName);
                        myResult := OpenDeskAcc(myDAName);
                    END;
            END;
    END;

```

Menus

```

END;
mFile:
BEGIN
    CASE myItemNum OF
        iNew:
            myWindow := DoCreateWindow;
        iClose:
            DoCloseWindow(FrontWindow);
        iQuit:
            DoQuit;
        OTHERWISE
            ;
    END;
END;
mEdit:
BEGIN
    IF NOT SystemEdit(myItemNum - 1) THEN
        ;
    END;
mVennD:
BEGIN
    myWindow := FrontWindow;
    CASE myItemNum OF
        iCheckVenn:
            DoVennCheck(myWindow);
        iDoVenn:
            DoVennAnswer(myWindow);
        iClearVenn:
            DoVennClear(myWindow);
        iNextTask:
            DoVennNext(myWindow);
        iCheckArg:
            DoVennAssess(myWindow);
        iGetVennPrefs:
            DoModelessDialog(rVennDPrefsDial, gPrefsDialog);
        OTHERWISE
            ;
    END;
END;

OTHERWISE
;

```

Menus

```

END;
HiliteMenu(0);
END;

```

The DoMenuCommand procedure is passed a long integer that encodes the menu number and item number of the chosen item. As you can see, DoMenuCommand consists mainly of a CASE statement that branches on the menu number. Each menu number, in turn, consists mainly of a CASE statement that branches on the menu item number. In this simple way, you can handle all menus and all menu items.

Most of the innermost branches just call application-defined routines to handle the appropriate menu item choice. (For example, if the user chooses Quit from the File menu, then DoMenuCommand calls the application-defined routine DoQuit.) The code that handles choices in the Apple menu (Listing 8-4) is slightly different, however.

Listing 8-4 Handling Apple menu selections

```

iAbout:
    BEGIN
        DoAboutBox;
    END;
OTHERWISE
    BEGIN
        GetMenuItemText(GetMenuHandle(mApple), myItemNum, myDAName);
        myResult := OpenDeskAcc(myDAName);
    END;

```

If the user chooses the command About Venn Diagrammer (picked out by the constant iAbout), then DoMenuCommand calls the application-defined routine DoAboutBox (see Listing 7-7 on page 145). Otherwise, the user must have chosen a desk accessory or other item in the Apple menu. In that case, DoMenuCommand retrieves the name of the desk accessory (by calling GetMenuItemText) and passes that name to the OpenDeskAcc function.

Because Venn Diagrammer doesn't support any text editing, it simply calls the system software routine SystemEdit to handle user choices in the Edit menu. SystemEdit checks whether the frontmost window belongs to a desk accessory; if so, it passes the menu choice to the desk accessory and returns TRUE. The parameter to SystemEdit is interpreted so you can pass the item number less 1 of the standard Edit menu commands.

Before exiting, DoMenuCommand calls the Menu Manager procedure HiliteMenu to undo the menu title highlighting provided automatically by MenuSelect or MenuKey.

Handling Keyboard Equivalents

Keyboard equivalents of menu 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 character, you can then determine if this combination maps to a known *Command-key equivalent* by calling the Menu Manager function `MenuKey`. Listing 8-5 shows the Venn Diagrammer application's `DoKeyDown` procedure, which handles key-down events and determines if a keyboard equivalent was pressed.

Listing 8-5 Handling Command-key equivalents

```
PROCEDURE DoKeyDown (myEvent: EventRecord);
    VAR
        myKey:      char;
    BEGIN
        myKey := chr(BAnd(myEvent.message, charCodeMask));
        IF (BAnd(myEvent.modifiers, CmdKey) <> 0) THEN
            BEGIN
                DoMenuAdjust;
                DoMenuCommand(MenuKey(myKey));
            END;
        END;
```

The `DoKeyDown` procedure first extracts the pressed key from the `message` field of the event record and then examines the `modifiers` field to determine whether the Command key was also pressed. If so, the application first adjusts its menus and then calls the `DoMenuCommand` procedure defined in Listing 8-3 on page 157. In turn, `DoKeyDown` passes to `DoMenuCommand` the value returned from the `MenuKey` function. If the key combination pressed by the user is not the keyboard equivalent of any currently enabled menu item, then `MenuKey` sets the high-order word of its return value to 0.

Note

The Venn Diagrammer application does not accept any text input from the user. As a result, the `DoKeyDown` procedure shown in Listing 8-5 doesn't need an `ELSE` clause to handle keypresses in which the Command key is not held down. ♦

Several keyboard equivalents (listed in Table 8-1) are reserved for common commands in the File and Edit menus. If your application supports these commands, you should assign these equivalents to the specified commands. Otherwise, you should ignore these keyboard equivalents.

Table 8-1 Reserved keyboard equivalents

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

IMPORTANT

You should never assign the keyboard equivalents listed in Table 8-1 to other menu commands. This helps ensure predictable behavior among all applications. ▲

Adjusting Menus

At any given time during the execution of your application, it's likely that some of the commands in your menus will not be appropriate. For example, if the front window is a dialog window, then any menu commands that manipulate only document windows should be disabled. Similarly, if the desktop shows no windows belonging to your application, then the Close command in the File menu should be disabled. When a menu item is disabled, it is drawn in a dimmed text and is not highlighted when the cursor passes over it. This disabling prevents the user from choosing those commands.

An easy way to achieve this effect is to call an application-defined routine that adjusts the menus according to the current application context just before you call either `MenuSelect` or `MenuKey`. Listing 8-6 shows the version of `DoMenuAdjust` used by the Venn Diagrammer application.

Listing 8-6 Adjusting menus

```
PROCEDURE DoMenuAdjust;  
  VAR  
    myWindow:   WindowPtr;  
    myMenu:     MenuHandle;  
    count:      Integer;
```

Menus

```

BEGIN
    myWindow := FrontWindow;

    IF myWindow = NIL THEN
        DisableMenuItem(GetMenuHandle(mFile), iClose)
    ELSE
        EnableMenuItem(GetMenuHandle(mFile), iClose);

    myMenu := GetMenuHandle(mVennD);
    IF IsAppWindow(myWindow) THEN
        FOR count := 1 TO kNumTools DO
            EnableMenuItem(myMenu, count)
        ELSE
            FOR count := 1 TO kNumTools DO
                DisableMenuItem(myMenu, count);

    IF IsDAccWindow(myWindow) THEN
        EnableMenuItem(GetMenuHandle(mEdit), 0)
    ELSE
        DisableMenuItem(GetMenuHandle(mEdit), 0);
    DrawMenuBar;
END;

```

The `DoMenuAdjust` procedure calls `FrontWindow` to get a pointer to the frontmost window belonging to the Venn Diagrammer application. If there is no window belonging to the Venn Diagrammer application, `DoMenuAdjust` disables the Close menu command in the File menu. Conversely, if there is a window belonging to the application, `DoMenuAdjust` enables the Close command.

If the front window is a document window, then `DoMenuAdjust` enables all the document-specific commands in the Venn menu; otherwise, it disables all those commands. (`DoMenuAdjust` retrieves the menu handle by calling `GetMenuHandle` and passes that handle to `EnableMenuItem` or `DisableMenuItem`.)

You can disable or enable an entire menu by passing `DisableMenuItem` or `EnableMenuItem` the value 0 in place of a menu item number. This is the strategy that `DoMenuAdjust` follows for the Edit menu. Venn Diagrammer does no editing of its own, so `DoMenuAdjust` makes certain to enable the Edit menu only when a desk accessory window is frontmost. When you call `DisableMenuItem` or `EnableMenuItem` in this way, however, you also need to call the Menu Manager procedure `DrawMenuBar` to update the menu bar's appearance.