3 Version 1.1

MENUS

Includes Demonstration Program Menus

Introduction

Types of Menus

A menu is a user interface element which allows the user to view, or choose from, a list of choices and commands provided by your application. There are basically three types of menus:

- **Pull-Down Menus.** A pull-down menu is identified by a menu title in the menu bar. Each pull-down menu comprises a menu title and one or more menu items.
- **Pop-Up Menus.** A pop-up menu is a menu which does not appear in the menu bar but rather appears on another part of the screen when the user presses the mouse button while the cursor is at a particular location. Pop-up menus are generally located within dialog boxes.
- **Submenus.** A submenu is a menu that is attached to another menu. A menu to which a submenu is attached is referred to as a **hierarchical menu**. Note that submenus should not normally be attached to pop-up menus as this tends to make the interface more complex and less intuitive to the user.

Pull-Down Menus

Menu Definition Functions and Menu Bar Definition Functions

The Menu Manager uses the following to display, and to perform basic operations on, menus and the menu bar:

- **Menu Definition Function.** When you define a menu, you must specify the required menu definition function. The Menu Manager uses that menu definition function to draw the menu items in a menu, determine which item the user chose, insert scrolling indicators as items in a menu, calculate the menu's dimensions, etc.
- **Menu Bar Definition Function.** The Menu Manager uses the menu bar definition function to draw and clear the menu bar, determine whether the cursor is currently within the menu bar or any currently displayed menu, calculate the left edges of menu titles, highlight a menu title, invert the entire menu bar, draw the menu's shadow box, and save/restore the bits behind a menu.

Standard Menu Definition Function and Menu Bar Definition Function

The system software provides a standard menu definition function and a standard menu bar definition function, which are stored as code resources in the System file. The standard menu definition function is the `'MDEF'` resource with a resource ID of 0. The standard menu bar definition function is the `'MBDF'` resource with a resource ID of 0.

When you define your menus and menu bar, you specify the definition functions that the Menu Manager should use when managing them. Ordinarily, you will use the standard functions; however, as with most other elements of the Macintosh user interface, the option is available to write your own custom definition function if you need to provide features not available in the standard definition functions.[1]

The Menu Bar and Menus

The Menu Bar

The menu bar extends across the top of the screen. As defined by the standard menu bar definition function, the menu bar is white and high enough to display menu titles in the height of the system font (Chicago 12 point for Roman Scripts) plus a single pixel bottom border.

Generally, the menu bar should always be visible. If you want to hide the menu bar for some reason, you should provide a method (for example, a keyboard equivalent for a menu command) to allow the user to make the menu bar reappear.

The 'MBAR' Resource. Each application has its own menu bar, which is defined by an `'MBAR'` resource. This resource lists the order and resource ID of each menu appearing in your menu bar. Your menu's `'MBAR'` resource should be defined such that the Apple menu is the first menu in the menu bar, with the **File** and **Edit** menus being the next two. The Help and Application menus do not need to be defined in the `'MBAR'` resource, since the Menu Manager automatically adds them to the menu bar when the application calls `GetNewMBar` provided that your menu bar includes the Apple menu.

Menus

All Macintosh applications should provide, as a minimum, the **standard menus**. The standard menus are the Apple menu, the **File** menu and the **Edit** menu.

Your application can disable any menu, which causes the Menu Manager to dim that menu's title and all associated menu items. The menu items can also be disabled individually. Your application should specify whether menu items are enabled or disabled when it first defines and creates a menu and can enable or disable items at any time thereafter.

The 'MENU' Resource. For each menu, you define the menu title and the individual characteristics of its menu items in a `'MENU'` resource.

The 'mctb' Resource. Ordinarily, the Menu Manager uses default colours (black text on a white background) for menus. However, the default colours of the title, item text, and background can be changed if you provide a menu colour table (`'mctb'`) resource with the same ID as the associated `'MENU'` resource[2].

Menu Items

A menu item can contain text or a dividing line (that is, a **divider**). A divider is always dimmed. Each menu item, other than dividers, can have a number of characteristics as follows:

---

[1] Chapter 18 — Lists and Custom List Definition Functions and Chapter 19 — Custom Control Definition Functions and VBL Tasks include examples of custom definition functions for other elements of the user interface known as lists and controls.

[2] If you use ResEdit to create your menu resources, the `'mctb'` resource will be created automatically when you specify a coloured title, item text and / or background within the menu resource editor.

- An icon, small icon, reduced icon or colour icon to the left of the menu item's text. (Note that items with small or reduced icons cannot have submenus.)

- A checkmark or other marking character to the left of the menu item's text, indicating 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). (An item that has a keyboard equivalent cannot have a submenu, a small icon or a reduced icon.)

- A triangular indicator to the right of a menu item's text to indicate that the item has a submenu. (An item that has a submenu cannot have a keyboard equivalent, a marking character, a small icon or a reduced icon.)

- A font style (bold, italic, etc.) for the menu item's text.

- The text of the menu item.

- The ellipsis character (**...**) as the last character in the text of the menu item, indicating that, before executing the command, the application will display a dialog box requesting more information from the user. (The ellipsis character should not be used in menu items which display informational dialogs or a confirmational alert.[3])

- A dimmed appearance when the application disables the item. (When the menu title is dimmed, all menu items in that menu are also dimmed.)

A menu can contain any number of menu items; however, only the first 31 can be disabled.

Groups of Menu Items

Where appropriate, menu items should be grouped, with each group separated by a divider. For example, a menu can contain commands which perform actions and commands which set attributes. The action commands which are logically related should be grouped, as should attribute commands which are interdependent. The attribute commands which are mutually exclusive, and those which form accumulating attributes (for example, **Bold**, ***Italic*** and **Underline**), should also be grouped.

Keyboard Equivalents for Menu Commands

The Menu Manager provides support for **Command-key equivalents**[4]. You detect a Command-key equivalent by examining the `modifiers` field of the event record for a keyboard event, which allows you to determine if the Command key was pressed at the same time as the keyboard event. If so, your application typically calls `MenuKey`, which determines if the one-byte character matches any of the keyboard equivalents defined for your menu items. (Note that `MenuKey` does not distinguish between uppercase and lowercase letters.)

Reserved Keyboard Equivalents. Apple reserves the following keyboard equivalents, which should be used in the **File** and **Edit** menus of your application:

---

[3]It is interesting to note, however, that Apple itself does not always obey this rule. For example, choosing **About This Macintosh...** (note the ellipsis) from the Finder's Apple menu displays an informational dialog box only.

[4]The term **keyboard equivalent** refers to a keyboard combination, such as ⌘-C, or any other combination of the Command key, another key and one or more modifier keys. The term **Command-key equivalent** refers specifically to a keyboard equivalent comprising the Command key and one other key other than a modifier key.

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**

Other common keyboard equivalents are:

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

Menus Added Automatically By the Menu Manager

The menus added automatically by the Menu Manager (the Help and Application menus) have icons as titles and are sometimes referred to as the **system-managed menus**. The Help menu is displayed only if space is available. The application menu is invariably displayed, overlapping the main part of a long menu if this becomes necessary.

Your application does not need to take any action if the user chooses an item from the Application menu. However, if the user chooses an item added by your application to the Help menu, your application is responsible for taking the appropriate action.

The Apple Menu

The Apple menu should be defined as the first in your application. Typically, applications provide an **About** command as the first menu item, followed by a divider. The remaining items are, of course, controlled by the contents of the Apple Menu Items folder in the System folder.

To create your application's Apple menu, firstly define the Apple menu title, the characteristics of your application's **About** command and the divider following it in a `'MENU'` resource. Then insert the contents of the Apple Menu Items folder into your application's Apple menu by calling `AppendResMenu`, with `'DRVR'` specified as the resource type in the parameter `theType`.

When the user chooses the **About** command, your application should display a dialog box or an alert box containing your application's name, version number, copyright information, any other information as required, and an **OK** button.

When the user chooses an item other than the **About** command, your application should call the `OpenDeskAcc` function, which schedules the chosen item for execution and then returns to your application. At the next call to `WaitNextEvent`, your application receives a suspend event and the chosen item becomes the foreground process.

The File Menu

The standard **File** menu contains commands related to the management of documents, plus the **Quit** command. The standard commands (see Fig 1) should be supported by your application. Any other commands added to the standard section of the menu should pertain to the management of documents.

The actions your application should take when **File** menu commands are chosen are detailed at Chapter 13 — Printing and Chapter 14 — Files.

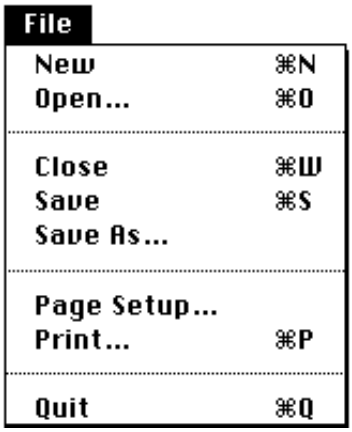

FIG 1 - STANDARD FILE MENU

The Edit Menu

The standard Edit menu (see Fig 2) provides commands which allow the user to edit the contents of their documents, to copy data between different applications using the Clipboard, and to facilitate data sharing between documents created by different applications via publish and subscribe.

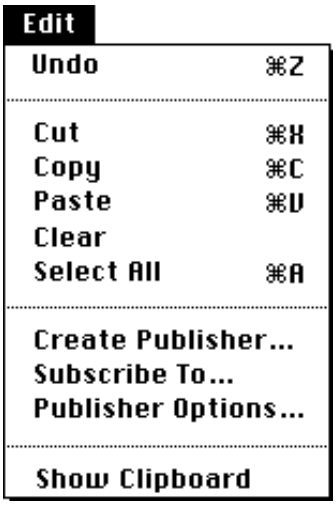

FIG 2 - STANDARD EDIT MENU

All Macintosh applications should include the standard editing commands (**Undo**, **Cut**, **Copy**, **Paste** and **Clear**) so as to support those operations in dialog boxes and in old-style desk accessories launched in the application's partition. (Old-style desk accessories utilise the host application's menus.)

An additional word or phrase should be added to **Undo** to clarify exactly what action your application will reverse. Other commands may be added if they are related to editing or changing the contents of your application's documents.

The Help Menu

You can add items to the end of the Help menu to give the user access to any online help that your application provides in addition to help balloons. Items are added to the Help menu using `HMGetHelpMenuHandle` and `AppendMenu`. When adding items, include the name of your application in the command so as to indicate to the user just which application the help relates to.

Help Balloons

In the Help menu, the effect of selecting **Show Balloons** and **Hide Balloons** is global and affects all applications. The Help Manager provides balloons for the Apple, Help and Application menu titles, for items in the Application menu, and for the standard items in the Help menu. Your application should provide the content of help balloons for all other menu items and menus in your application.

The Application 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 another application, the Menu Manager sends your application a suspend event.

Font Menus

If your application has a **Font** menu, you should list in that menu the names of all currently available fonts (that is, all those residing in the Fonts folder in the System folder). Fonts are added to the **Font** menu using `AppendResMenu` or `InsertResMenu`, which add items to the specified menu in alphabetical order.

Your application should indicate which font is in use by adding a checkmark to the left of the name in the **Font** menu. If the current selection contains more than one font, a dash should be placed next to the name of each font the selection contains. When the user starts entering text at the insertion point, your application should display text in the current font.

Font Attributes

Separate menus should be used to accommodate lists of font attributes such as styles and sizes. Since the system software supports both bitmapped and TrueType fonts, your application should not provide an upper limit for font sizes.

Pop-Up Menus

Pop-up menus are used to present the user with a list of choices in a dialog box or window. They are identified by a downward pointing triangle within the **pop-up box** (see Fig 3).

Pop-up menus work well when your application needs to present several choices to the user and it is acceptable to hide these choices until the menu is opened. (Other methods of displaying choices are checkboxes and radio buttons.) Pop-up menus should not be used for multiple choice lists or as a way to provide more commands. They should contains attributes rather than actions; accordingly, Command-key equivalents should not be used in pop-up menus.

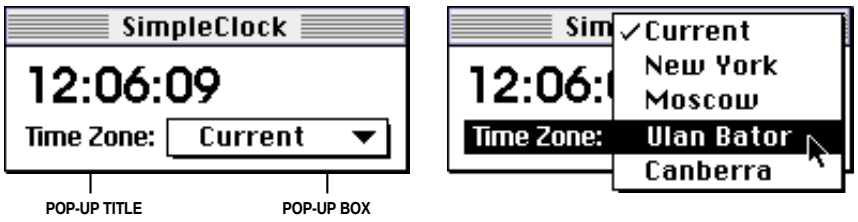

FIG 3 - POP-UP MENU (EXAMPLE)

If you do not provide a title for the pop-up menu, the current pop-up menu item serves as the title.

Pop-Up Control Definition Function

The standard pop-up menu is actually implemented as a **control**, its appearance and behaviour being determined by a pop-up **control definition function**. If the 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 with the menu. The pop-up control definition function handles all highlighting and unhighlighting and adds the checkmark to the current menu item. When the user releases the mouse button, it changes the text in the pop-up box and stores the item number of the chosen item as the value of the control[5].

Use of Control Manager Routines

The Control Manager function `GetControlValue` may be used to retrieve the value of the control.

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 Control Manager routines to display the pop-up menu and to handle user interaction with the control.

Type-In Pop-Up Menus

Type-in pop-up menus (see Fig 4) are used to offer the user a list of choices while, at the same time, allowing the user to type in an additional choice.

FIG 4 - TYPE-IN POP-UP MENU (EXAMPLE)

The standard pop-up control definition function, however, does not provide specific support for type-in menus; accordingly, you must create your own control definition function to handle such menus.

Hierarchical Menus

A hierarchical menu is a menu which has a submenu attached to it. Hierarchical menus should be used to provide the user with additional choices in the nature of attributes. They should not be used to provide additional commands. There should only ever be one hierarchical level, that is, there should be only one level of submenus.

Menu Records, Menu IDs, Item Numbers, and Menu Lists

The Menu Record

The Menu Manager maintains information about menus in **menu records**, a data structure of type `MenuInfo`:

```
struct MenuInfo
{
  short   menuID;       // Menu ID of the menu.
  short   menuWidth;    // Horizontal dimensions of the menu in pixels.
  short   menuHeight;   // Vertical dimensions of the menu in pixels.
  Handle  menuProc;     // Handle to the menu definition procedure.
  long    enableFlags;  // Enabled/disabled flags.
  Str255  menuData;     // Menu title
};
```

[5]Controls and their values are explained at Chapter 5 — Controls.

```
typedef struct MenuInfo MenuInfo;
typedef MenuInfo *MenuPtr, **MenuHandle;
typedef MenuHandle MenuRef;
```

You typically specify most of this information in a `'MENU'` resource. When you create a menu, the Menu Manager creates a menu record for the menu and returns a handle to that record. The Menu Manager automatically updates the menu record when you make any changes to the menu.

Menu IDs

To refer to a menu, you usually use either the menu's ID or the handle to the menu's menu record. Accordingly, you must assign a **menu ID** to each menu in your application as follows:

- Pull-down and pop-up menus must use a menu ID greater than 0.
- Submenus of an application must use a menu ID of from 1 to 235.

Item Numbers

To refer to a menu item, you use the item's **item number**. Item numbers in a menu start at 1.

The Menu List

The **menu list** contains handles to the menu records of one or more menus (although a menu list can, in fact, be empty). The end of a menu list contains handles to the menu records of submenus and pop-up menus, if any, the phrase "submenu portion of the menu list" referring to this portion of the list.

When your application initialises the Menu Manager, the Menu Manager creates the menu list. The menu list is initially empty but changes as your application adds menus to it or removes menus from it.

Creating Your Application's Menus

Creating Resources

Creating `'MENU'` Resources for Menus

A `'MENU'` resource defines the menu title and the characteristics of menu items in a menu. The following is a typical `'MENU'` resource in Rez format, in this case a 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. */
  0b11111111111111111111111111111101, /* Enable About, disable divider, enable rest. */
  enabled,                            /* Enable menu title. */
  apple,                              /* Menu title. */
  {
    "About This Application…",        /* FIRST MENU ITEM text. */
       noicon,                        /*   Icon number (if any). */
       nokey,                         /*   Keyboard equivalent or submenu or icon. */
       nomark,                        /*   Marking character or submenu ID. */
       plain;                         /*   Style of menu item text. */
    "-",                              /* SECOND MENU ITEM text. */
       noicon,nokey,nomark,plain
  }
};
```

Resource ID and Menu ID. The resource ID for this menu is specified as 128. Any number equal to or greater than 128 may be used as the resource ID for a menu. By convention, 128 is used as the resource ID of the Apple menu and sequential numbers are used for the remaining menus. Also by convention, the menu ID is usually set to the same number as the resource ID, though this is not strictly

necessary. (As previously stated, any number greater than 0 may be used as the menu ID of a pull-down or pop-up menu.)

Menu Definition Function. The listing specifies that this menu uses the standard menu definition function. The constant `textMenuProc` represents the standard `'MDEF'` resource ID.

Item Enable/Disable. The 32-bit number, which is expressed as a 31-bit field followed by a `Boolean` field, indicates whether the corresponding menu item is to be enabled or disabled, with bit 0 indicating whether the menu is enabled or disabled.

Title. The title of the menu is specified by the constant `apple`, causing the Menu Manager to use a small Apple icon as the title of the menu.

Item Text and Characteristics. The listing then defines the text and other characteristics of each menu item. By specifying various combinations of values in the icon field and keyboard equivalent field, you can define an icon (normal, small, reduced or colour), a keyboard equivalent, or a submenu. (Some of these characteristics are, as previously explained, mutually exclusive.)

Creating `'MENU'` Resources for Submenus

When a submenu is attached to a menu item in a pull-down menu, the name of the menu item is the title of the attached submenu. In Rez, 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 identify the menu ID of the submenu in place of the marking character. In the following example of a Rez input for a `'MENU'` resource, `Label style` is the menu item text and `mSubMenu` is the menu ID of the submenu:

```
#define mOutline  135
#define           mSubMenu  181
...
resource 'MENU' (mOutline, preload)
{
  mOutline,
  textMenuProc,
  0b00000000000000000000000000010000,
  enabled,
  "Outline",          /* Menu title */
  {
    "Expand",         noicon, "E",              nomark,     plain;
    "Collapse",       noicon, nokey,            nomark,     plain;
    "Label style  ",  noicon, hierarchicalMenu, mSubMenu ,  plain
  }
};

resource 'MENU' (mSubMenu, preload)
{
  mSubMenu ,
  textMenuProc,
  0b00000000000000000000000000000011,

  enabled,
  "Label Style  ",    /* Menu title (ignored because defined by parent item text.) */
  {
    "Alphabetic",     noicon, nokey, nomark, plain;
    "Bullet",         noicon, nokey, nomark, plain
  }
};
```

The menu items of a submenu are defined in the same way as for a pull-down menu

Creating an 'MBAR' Resource

Your application's menu bar is defined in an 'MBAR' resource. An example is as follows:

```
#define  rMenuBar  128
#define  mApple    128
#define  mFile     129
#define  mEdit     130
...
resource 'MBAR' (rMenuBar, preload)  /* Resource ID, preload */
{
  { mApple, mFile, mEdit };          /* Resource IDs of menus in this menu bar */
};
```

Help Balloons - 'hmmu' Resources

You should also define Help balloons for each of your application's menu items and each menu title. Help balloons are defined in 'hmmu' resources.

Creating the Menu Bar and Pull-Down Menus

Your application should call GetNewMBar to create a menu list as defined in an 'MBAR' resource. GetNewMBar returns a handle to the created menu list. For each menu defined by the resource, GetNewMBar creates a menu record, creates each menu according to the menu definition in its corresponding 'MENU' resource, and inserts each menu into the menu list.

SetMenuBar should then be used to set the **current menu list** as the menu list created by your application. A call to DrawMenuBar completes the process by drawing the menu bar, displaying all the menu titles in the current menu list.

Adding Menus to the Menu List

A menu may be added to the current menu list using one of the following procedures:

- Read the relevant 'MENU' resource in with GetMenu, add it to the current menu list with InsertMenu, and update the menu bar with DrawMenuBar.
- Use NewMenu to create a new empty menu, use AppendMenu, InsertMenuItem, InsertResMenu, or AppendResMenu to fill the menu with menu items, add the menu to the current menu list using InsertMenu, and update the menu bar using DrawMenuBar.

Note that GetMenuHandle may be used to obtain a handle to the menu record of any menu in the current menu list.

Creating a Hierarchical Menu

GetNewMBar does not read in the resource descriptions of submenus but simply records the menu ID of any submenu in the menu record. Submenu descriptions are read in with GetMenu and the submenu is inserted in the current menu list using InsertMenu, with the constant hierMenu passed as the second parameter to that call.[6]

Creating a Pop-Up Menu

As previously stated, pop-up menus are actually implemented as controls. To create a pop-up menu, define the pop-up menu and its menu items in the same way as for other menus (that is, using a 'MENU' resource), create a control which uses the standard pop-up control definition function (that is, specify

[6]As the user traverses menu items, if an item has a submenu, the MenuSelect function looks in the submenu portion of the menu list for the submenu. It then searches for a menu with a defined menu ID that matches the menu ID specified by the hierarchical menu item. If it finds a match, it attaches the submenu to the menu item.

the popupMenuProc constant in procID field of the resource description of the control), and associate the control with a window or dialog box[7].

If you specify popupMenuProc in the procID field of the resource description of a control, when your application creates the control (with a call to 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.

Pop-up Menus in Dialog Boxes

Dialog Manager or Control Manager routines, not Menu Manager routines, are used to display and manage 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 handle all user interaction. The Control Manager function GetControlValue will get the value of the control, which will equate to the number of the item selected by the user.

Pop-up Menus in Windows

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.

Changing the Appearance of Items in a Menu

Menu Manager routines may be used to change the appearance of items in a menu, for example, the font style, text or other characteristics. Most of the routines which get or set menu characteristics require three parameters:

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

Enabling and Disabling Menu Items

Specific menu items or entire menus are disabled and enabled using DisableMenu and EnableMenu, which both take a handle to the menu record that identifies the desired menu and either the item number of the menu to be enabled / disabled or a value of 0 to indicate that the entire menu is to be enabled / disabled.

When an entire menu is disabled or enabled, DrawMenuBar should be called to update the appearance of the menu bar. If you do not need to update the menu bar immediately, you can use InvalMenuBar instead of DrawMenuBar, causing the Event Manager to redraw the menu bar the next time it scans for update events. This will reduce the menu bar flicker which will occur if DrawMenuBar is called more than once in rapid succession.

If you disable an entire menu, the Menu Manager dims that menu's title at the next call to DrawMenuBar and dims all menu items 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.

Changing the Text and Font Style of Menu Items

GetMenuItemText and SetMenuItemText are used to get and set the text of a menu item. GetItemStyle and SetItemStyle are used to get and set the font style of a menu item.

---

[7] If you add the constant useWFont to the constant popupMenuProc, the pop-up title and menu item text will be drawn in the current graphics port's font rather than the system font.

Changing the Mark and Icons of Menu Items

`GetItemMark` and `SetItemMark` are used to get and set the marking character of a menu item. `GetItemIcon` and `SetItemIcon` are used to get and set the icon of a menu.

Adding Items to a Menu

Adding Items Other Than the Names of Resources

`AppendMenu` or `InsertMenuItem` are used to add items other than the names of resources (such as font resources) to a previously created menu. These functions allow you to specify the same characteristics for menu items as are available when defining a `'MENU'` resource. They require:

- A handle to the menu record of the menu involved.
- A string describing the items to add.

The string consists of the text of the menu item and any required characteristics. 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 the required characteristics. The following metacharacters may be used:

MetaCharacter	Description
; or Return	Separates menu items.
^	When followed by an icon number, defines the icon for the item.
!	When followed by a character, defines the mark for the item. If the keyboard equivalent field contains `0x1B`, 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, respectively, bold, italic, underline, outline or shadow.
/	When followed by a character, defines the keyboard equivalent for the item. When followed by `0x1B`, specifies that this menu item has a submenu. (To specify that the menu item has a script code, small icon, or reduced icon, use `SetItemCmd` to set the keyboard equivalent field to, respectively, `0x1C`,`0x1D` or `0x1E`.)
(	Defines the menu item as disabled.

As an example of the use of metacharacters, the following is a string list (`'STR#'`) resource, in Rez input format, which stores the text of some menu items:

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

The second string in this resource uses metacharacters to specify that the menu item is to be disabled, that it has an icon with a resource ID 258 (2+256)[8], that it has the "=" character as a marking character, that the text style is bold, and that the item has a keyboard equivalent of Command-E.

Examples

The following code uses `AppendMenu` to append a menu item with no specific characteristics other than its text to the menu identified by the menu handle. The text for the menu item is "`Pick a Colour…`" as stored in the preceding `'STR#'` resource.

[8]The Menu Manager adds 256 to the number you specify, and uses the result as the icon's resource ID.

```
MenuHandle  myMenu;
Str255      ItemString;
...
myMenu = GetMenuHandle(mLibrary);
GetIndString(itemString,300,1);
AppendMenu(myMenu,itemString);
```

To insert an item after a given menu item, use `InsertMenuItem`. The following code inserts the menu item "`Everything`" after the menu item with the item number specified in the `iRed` constant:

```
MenuHandle  myMenu;
Str255      ItemString;
...
myMenu = GetMenuHandle(mColours);
GetIndString(itemString,300,2);
InsertMenuItem(myMenu,itemString,iRed);
```

The following code appends multiple items to the **Edit** menu using `AppendMenu`:

```
MenuHandle  myMenu;
...
myMenu = GetMenuHandle(mEdit);
AppendMenu(myMenu,"\pUndo/Z;-;Cut/X;Copy/C;Paste/V");
```

`InsertMenuItem` differs from `AppendMenu` in the way it handles the given text string when that string contains multiple items, inserting them in reverse order. This code is equivalent to the last line of the preceding code:

```
InsertMenuItem(myMenu,"\pPaste/V;Copy/C;Cut/X-;-;Undo/Z",0);
```

The following code adds a divider to the **Edit** menu:

```
AppendMenu(myMenu,"\p(-");
```

Adding Items Comprising Resource Names to a Menu

`AppendResMenu` or `InsertResMenu` may be used to add items that consist of resource names to a menu.

For example, you can use `AppendResMenu` to add the names of all font resources in the Fonts folder as menu items in your application's **Font** menu. Similarly, `AppendResMenu` can be used to add all of the items from the Apple Menu Items folder to your application's Apple menu (with `'DRVR'` specified as the resource type in the call). These are common instances of when you will need to add items not already defined in a `'MENU'` resource.

Items are added to your application's Help menu using `AppendMenu` or `InsertMenuItem`.

Handling Menu Choices

Determining the Menu ID and Menu Item — `MenuSelect` and `MenuKey`

When the user presses the mouse button while the cursor is in the menu bar, your application should first adjust its menus (that is, enable or disable menu items and add or remove marks as required) and then call `MenuSelect`. `MenuSelect` tracks the mouse, displays menus, highlights menu titles, displays and highlights enabled menu items, handles all user activity until the user releases the mouse button, and returns a long integer as its function result. The long integer contains the menu ID in the high word and the item number in the low word.

If some of your menu items have keyboard equivalents, your application should detect such key-down events. If an examination of the `modifiers` field of the event record reveals that the Command key was down, your application should first adjust its menus and then call `MenuKey`. `MenuKey` scans the current menu list for a menu item that has a matching keyboard equivalent. Like `MenuSelect`, `MenuKey` returns a long integer indicating which menu item was chosen.

If the user did not actually choose a menu command with the mouse, or if the user pressed a keyboard combination which did not map to a keyboard equivalent, `MenuSelect` and `MenuKey` return 0 in the high word, the value in the low word being undefined.

The long word returned by `MenuSelect` and `MenuKey` should be passed as a parameter to an application-defined function which switches according to the menu ID in the high word and passes the low word to other application-defined functions which respond appropriately to that menu command.

Unhighlighting the Menu Title

Recall that one of the actions of `MenuSelect` and `MenuKey` is to highlight the menu title. Ordinarily, your application should not unhighlight the menu title (using `HiLiteMenu`) until it performs the action associated with the menu command chosen by the user. However, if, in response to a menu command, your application displays a modal dialog box containing an editable text item, you should unhighlight the menu title immediately so that the user can access the **Edit** menu.

Adjusting Menus

Menu adjustment should be on the basis of the type of window that is currently the frontmost window, for example, a text window, a desk accessory, a modal dialog box or a modeless dialog box. Accordingly, the application-defined menu adjustment function should first determine which window is the front window. The following are examples of menu adjustment functions:

```
void  doAdjustMenus(void)
{
  WindowPtr windowPtr;
  SInt16  windowType;

  windowPtr = FrontWindow();
  windowType = doGetWindowType(windowPtr);

  switch windowType
  {
    case kMyDocWindow:
      doAdjustFileMenuForDocWindow();
      doAdjustEditMenuForDocWindow();
      // Adjust others.
      break;

    case kMyDialogWindow:
      doAdjustMenusForDialogs();
      break;

    case kNil:
      doAdjustMenusNoWindows();
      break;
  };

  DrawMenuBar;
}

void doAdjustFileMenuForDocWindow(void)
{
  MenuHandle  menuHdl;

  menuHdl = GetMenuHandle(mFile);

  EnableItem (menuHdl,iNew);
  EnableItem (menuHdl,iOpen);
  DisableItem(menuHdl,iClose);
  DisableItem(menuHdl,iSave);
  DisableItem(menuHdl,iSaveAs);
  DisableItem(menuHdl,iPageSetup);
  DisableItem(menuHdl,iPrint);
  EnableItem (menuHdl,iQuit);
}
```

Handling Apple Menu Choices

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

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

The `OpenDeskAcc` function prepares to open the desktop object chosen by the user. For example, if the user chose a document created by the SimpleText application, `OpenDeskAcc` schedules SimpleText for execution (or prepares to open it if it was not already open) and returns to your application. On your application's next call to `WaitNextEvent`, your application receives a suspend event and the Process Manager makes SimpleText the foreground process, instructing it to open the chosen document.

Handling Help Menu Choices

Both the `MenuSelect` and `MenuKey` functions return the `kHMHelpMenuID` constant (`-16490`) in the high word if the user chooses an appended item from the Help menu. The item number of the appended item is returned in the low word. When the `kHMHelpMenuID` constant is detected, an application-defined function should be called to respond to the user's choice of a Help menu command. That function must accommodate the fact that Apple reserves the right to change the number of standard items in the Help menu.

Handling a Size Menu

Preamble

Font sizes in **Size** menus should be outlined to indicate which sizes are directly provided by the current font. For bitmapped fonts, you should outline only those sizes that exist in the Fonts folder. For TrueType fonts, all sizes supported by that font should be outlined. The current font size should be indicated with a checkmark. If the current selection contains more than one font size, a dash should be placed next to each font size in the selection.

Size menus should, in addition to displaying available font sizes, provide an **Other** command to enable the user to specify a size not currently listed in the menu. When the user chooses the **Other** command, the current font size should be displayed in a dialog box which allows the user to enter the desired font size. If the user chooses a size not already in the menu, a check mark should be added to the **Other** menu item and the chosen size should be added in parenthesis to the text of the **Other** command.

Handling the Menu Choice

The following is an example application-defined function which handles a user's choice of an item in the **Size** menu:

```
void  doHandleSizeCommand(SInt16 menuItem)
{
  SInt16   numItems;
  Boolean  addItem;
  SInt32   sizeChosen;

  numItems = CountMItems(GetMenuHandle(mSize));
  if(menuItem == numItems)           // If user chose Other, display dialog box. If the
  {                                   // user-specified size is not in the menu, add a
    doDisplayOtherBox(sizeChosen);   // checkmark to the Other command and add the new
  }                                   // font size to the text of the Other command.
  else                                // Return sizeChosen.
```

```
        {                                                   // User chose a size.  Remove marks
          doRemoveMarksFromSizeMenu();                      // from item/s showing previous size.
          CheckItem(GetMenuHandle(mSize),menuItem,true);  // Add mark to chosen item.
          sizeChosen = doItemToSize(menuItem);              // Convert item number to font size.
        }
        doResizeSelection(sizeChosen);     // Update document state or user selection.
      }
```

Accessing Menus From Alert and Dialog Boxes

When alert boxes and dialog boxes are displayed, the Dialog Manager and the Menu Manager interact to provide varying degrees of access to menus in your menu bar. In some circumstances, you can rely on the system software to disable the appropriate menus and menu items. In other circumstances, you application must contribute to, or control, the matter of menu access.

The subject of menu access when alert boxes, modal dialog boxes, moveable modal dialog boxes, and modeless dialog boxes are displayed is somewhat involved, and is addressed in detail at Chapter 6 — Dialogs and Alerts.

Main Menu Manager Constants, Data Types, and Routines

Constants

For `markChar` **Parameter of** `SetItemMark` **Calls**

```
noMark       = 0
commandMark  = 17
checkMark    = 18
diamondMark  = 19
appleMark    = 20
```

For `beforeID` **Parameter of** `InsertMenu` **to Insert Submenu or Pop-up Menu Into the Submenu Portion of the Menu List**

```
hierMenu     = -1
```

Data Types

Menu Record

```
struct MenuInfo
{
  short   menuID;       // Number that identifies the menu.
  short   menuWidth;    // Width (in pixels) of menu.
  short   menuHeight;   // Height (in pixels) of menu.
  Handle  menuProc;     // Menu definition procedure.
  long    enableFlags;  // Indicates whether menu and menu items are enabled.
  Str255  menuData;     // Title of menu.
};

typedef struct MenuInfo MenuInfo;
typedef MenuInfo *MenuPtr, **MenuHandle;

typedef MenuHandle MenuRef;
```

Routines

Note: Some Menu Manager routines can be accessed using more than one spelling of the routine's name, depending on the header files supported by your development environment. The following reflects the newest spellings, as specified in version 2.1 of the Universal Headers.

Initialising the Menu Manager

```
void     InitMenus();
```

Creating Menus

```
MenuRef  NewMenu(short menuID, ConstStr255Param menuTitle);
MenuRef  GetMenu(short resourceID);
```

Adding Menus to and Removing Menus From the Current Menu List

```
void     InsertMenu(MenuRef theMenu,short beforeID);
void     DeleteMenu(short menuID);
void     ClearMenuBar(void);
```

Getting a MenuBar Description From an 'MBAR' resource

```
Handle   GetNewMBar(short menuBarID);
```

Getting and Setting the Menu Bar

```
Handle   GetMenuBar(void);
void     SetMenuBar(Handle menuList);
short    GetMBarHeight(void);
```

Drawing the Menu Bar

```
void     DrawMenuBar(void);
void     InvalMenuBar(void);
```

Responding to User Choice of a Menu Command

```
long     MenuKey(short ch);
long     MenuSelect(Point startPt);
long     MenuChoice(void);
void     HiliteMenu(short menuID);
long     PopUpMenuSelect(MenuRef menu,short top,short left,short popUpItem);
```

Getting a Handle to a Menu Record

```
MenuRef  GetMenuHandle(short menuID);
OSErr    HMGetHelpMenuHandle(MenuHandle *mh);
```

Adding and Deleting Menu Items

```
void     AppendMenu(MenuRef menu,ConstStr255Param data);
void     InsertMenuItem(MenuRef theMenu,ConstStr255Param itemString,short afterItem);
void     DeleteMenuItem(MenuRef theMenu,short item);
void     AppendResMenu(MenuRef theMenu,ResType theType);
void     InsertResMenu(MenuRef theMenu,ResType theType,short afterItem);
```

Getting and Setting the Appearance of Menus

```
void     EnableItem(MenuRef theMenu,short item);
void     DisableItem(MenuRef theMenu,short item);
void     GetMenuItemText(MenuRef menu,short item,Str255 itemString);
void     SetMenuItemText(MenuRef theMenu, short item, ConstStr255Param itemString);
void     GetItemStyle(MenuRef theMenu,short item,Style *chStyle);
void     SetItemStyle(MenuRef theMenu,short item,short chStyle);
void     GetItemMark(MenuRef theMenu,short item,short *markChar);
void     SetItemMark(MenuRef theMenu,short item,short markChar);
void     CheckItem(MenuRef theMenu,short item,Boolean checked);
void     GetItemIcon(MenuRef theMenu,short item,short *iconIndex);
void     SetItemIcon(MenuRef theMenu,short item,short iconIndex);
void     GetItemCmd(MenuRef theMenu,short item,short *cmdChar);
void     SetItemCmd(MenuRef theMenu,short item,short cmdChar);
```

Disposing of Menus

```
void     DisposeMenu(MenuRef theMenu);
```

Counting Items in a Menu

```
short    CountMItems(MenuRef theMenu);
```

Highlighting the Menu Bar

```
void     FlashMenuBar(short menuID);
void     SetMenuFlash(short count);
```

Recalculating Menu Dimensions

```
void     CalcMenuSize(MenuRef theMenu);
```

Demonstration Program

```
// ##############################################################################################
// Menus.c
// ##############################################################################################
//
// This program:
//
// •  Opens a window.
//
// •  Creates these pull-down menus: Apple, File, Edit, Font, Size and Special.
//
//    The Apple menu includes an "About…" menu item for the program.
//
//    The second menu item in the Special menu contains a submenu.
//
//    A "Help" menu item for the program is appended to the Help menu.
//
// •  Creates a pop-up menu in the window.
//
// •  Displays text in the window indicating the menu selection made by the user.
//
// The implementation of the Size menu is nominal only.  The current size is indicated
// with a checkmark; however, the number of sizes shown is not font-dependent and there
// is no "Other" item.
//
// Because the primary purpose of the program is to demonstrate menu creation and
// handling, no code is included to update and activate/deactivate the window or to
// respond to events which are not relevant to the demonstration.
//
// The program is terminated by selecting Quit from the File menu, by pressing the
// keyboard equivalent for that item (Command-Q), or by clicking in the window's go-away
// box.
//
// The program utilises the following resources:
//
// •  A 'WIND' resource (purgeable) (initially not visible).
//
// •  An 'MBAR' resource (preload, non-purgeable).
//
// •  'MENU' resources for the drop-down, hierarchical and pop-up menus (all preload,
//    all non-purgeable).
//
// •  A 'CNTL' resource for the pop-up menu (purgeable).
//
// ##############################################################################################

// ……… includes

#include <Fonts.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <SegLoad.h>
#include <ToolUtils.h>
#include <Balloons.h>
#include <Devices.h>

// ……… defines

#define mApple          128
#define  iAbout         1
```

```
#define mFile           129
#define  iQuit          11
#define mEdit           130
#define  iUndo          1
#define  iCut           3
#define  iCopy          4
#define  iPaste         5
#define  iClear         6
#define mFont           131
#define mStyle          132
#define  iPlain         1
#define  iBold          3
#define  iItalic        4
#define  iUnderline     5
#define  iOutline       6
#define  iShadow        7
#define mSize           133
#define  iTen           1
#define  iTwelve        2
#define  iEighteen      3
#define  iTwentyFour    4
#define mSpecial        134
#define  iFirstItem     1
#define  hmSecondItem   100
#define   siFirstSub    1
#define   siSecondSub   2
#define pControlResID   128
#define  pSydney        1
#define  pNewYork       2
#define  pLondon        3
#define  pRome          4
#define rWindowResource 128

// ……… global variables

Boolean gDone;
SInt16  gCurrentFont =  1;
Style   gCurrentStyle = 0;
SInt16  gCurrentSize =  2;

// ……… function prototypes

void    main                    (void);
void    doInitManagers          (void);
void    doGetMenus              (WindowPtr);
void    doEvents                (EventRecord *);
void    doMouseDown             (EventRecord *);
SInt16  doCheckForControlAndValue (EventRecord *,WindowPtr);
void    doAdjustMenus           (void);
void    doMenuChoice            (SInt32);
void    doPopupMenuChoice       (SInt16);
void    doAppleMenu             (SInt16);
void    doFileMenu              (SInt16);
void    doEditMenu              (SInt16);
void    doFontMenu              (SInt16);
void    doStyleMenu             (SInt16);
void    doSizeMenu              (SInt16);
void    doSpecialMenu           (SInt16);
void    doSubMenus              (SInt16);
void    doHelpMenu              (SInt16);
void    drawItemString          (Str255);

// ############################################################################## main

void  main(void)
{
  EventRecord eventRec;
  WindowPtr   windowPtr;

  // ……… initialize managers

  doInitManagers();

  // ……… open a window

  if(!(windowPtr = GetNewWindow(rWindowResource,NULL,(WindowPtr) -1)))
  {
```

```
      SysBeep(10);
      ExitToShell();
    }

    SetPort(windowPtr);

    // ……… set up menu bar and menus, then show window and pop-up menu

    doGetMenus(windowPtr);
    ShowWindow(windowPtr);
    DrawControls(windowPtr);

    // ……… event loop


    gDone = false;

    while(!gDone)
    {
      if(WaitNextEvent(everyEvent,&eventRec,180,NULL))
        doEvents(&eventRec);
    }
  }

  // ########################################################################## doInitManagers

  void  doInitManagers(void)
  {
    MaxApplZone();
    MoreMasters();

    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);

    InitCursor();
    FlushEvents(everyEvent,0);
  }

  // ############################################################################## doGetMenus

  void  doGetMenus(WindowPtr windowPtr)
  {
    Handle        menubarHdl;
    MenuHandle    menuHdl;
    OSErr         osErr;
    ControlHandle popupControlHdl;

    menubarHdl=GetNewMBar(128);
    if(menubarHdl == NULL)
      ExitToShell();
    SetMenuBar(menubarHdl);
    DrawMenuBar();

    menuHdl = GetMenuHandle(mApple);
    if(menuHdl != NULL)
      AppendResMenu(menuHdl,'DRVR');
    else
      ExitToShell();

    menuHdl = GetMenuHandle(mFont);
    if(menuHdl != NULL)
      AppendResMenu(menuHdl,'FONT');
    else
      ExitToShell();

    menuHdl = GetMenu(hmSecondItem);
    if(menuHdl != NULL)
      InsertMenu(menuHdl,hierMenu);
    else
      ExitToShell();

    osErr = HMGetHelpMenuHandle(&menuHdl);
    if(osErr == noErr)
```

```
    AppendMenu(menuHdl,"\pMenus Help");
  else
    ExitToShell();

  popupControlHdl = GetNewControl(pControlResID,windowPtr);
  if(popupControlHdl == NULL)
    ExitToShell();

  doFontMenu(gCurrentFont);
  doStyleMenu(gCurrentStyle);
  doSizeMenu(gCurrentSize);
}

// ############################################################################## doEvents

void  doEvents(EventRecord *eventRecPtr)
{
  SInt8 charCode;

  switch(eventRecPtr->what)
  {
    case mouseDown:
      doMouseDown(eventRecPtr);
      break;

    case keyDown:
    case autoKey:
      charCode = eventRecPtr->message & charCodeMask;
      if((eventRecPtr->modifiers & cmdKey) != 0)
      {
        doAdjustMenus();
        doMenuChoice(MenuKey(charCode));
      }
      break;

    case updateEvt:
      BeginUpdate((WindowPtr)eventRecPtr->message);
      EndUpdate((WindowPtr)eventRecPtr->message);
      break;

    case osEvt:
      HiliteMenu(0);
      break;
  }
}

// ############################################################################ doMouseDown

void  doMouseDown(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;
  SInt16    partCode, popupItem;
  SInt32    menuChoice;

  partCode = FindWindow(eventRecPtr->where,&windowPtr);

  switch(partCode)
  {
    case inSysWindow:
      SystemClick(eventRecPtr,windowPtr);
      break;

    case inMenuBar:
      doAdjustMenus();
      menuChoice = MenuSelect(eventRecPtr->where);
      doMenuChoice(menuChoice);
      break;

    case inContent:
      if(windowPtr != FrontWindow())
        SelectWindow(windowPtr);
      else
      {
        popupItem = doCheckForControlAndValue(eventRecPtr,windowPtr);
        if(popupItem)
          doPopupMenuChoice(popupItem);
      }
```

```
292         break;
293
294       case inDrag:
295         DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
296         break;
297
298       case inGoAway:
299         if(TrackGoAway(windowPtr,eventRecPtr->where))
300           gDone = true;
301         break;
302     }
303   }
304
305   // ############################################################ doCheckForControlAndValue
306
307   SInt16  doCheckForControlAndValue(EventRecord *eventRecPtr,WindowPtr windowPtr)
308   {
309     ControlHandle controlHdl;
310     SInt16        startControlValue = 0, finishControlValue = 0;
311
312     SetPort(windowPtr);
313     GlobalToLocal(&eventRecPtr->where);
314
315     if(FindControl(eventRecPtr->where,windowPtr,&controlHdl))
316     {
317       startControlValue = GetControlValue(controlHdl);
318       TrackControl(controlHdl,eventRecPtr->where,(ControlActionUPP) -1);
319       finishControlValue = GetControlValue(controlHdl);
320     }
321
322     if(finishControlValue != startControlValue)
323       return finishControlValue;
324     else
325       return 0;
326   }
327
328   // ################################################################################ doAdjustMenus
329
330   void  doAdjustMenus(void)
331   {
332     // Adjust menus here.
333   }
334
335   // ################################################################################ doMenuChoice
336
337   void  doMenuChoice(SInt32 menuChoice)
338   {
339     SInt16  menuID, menuItem;
340
341     menuID   = HiWord(menuChoice);
342     menuItem = LoWord(menuChoice);
343
344     if(menuID == 0)
345       return;
346
347     switch(menuID)
348     {
349       case mApple:
350         doAppleMenu(menuItem);
351         break;
352
353       case mFile:
354         doFileMenu(menuItem);
355         break;
356
357       case mEdit:
358         doEditMenu(menuItem);
359         break;
360
361       case mFont:
362         doFontMenu(menuItem);
363         break;
364
365       case mStyle:
366         doStyleMenu(menuItem);
367         break;
368
```

```
      case mSize:
        doSizeMenu(menuItem);
        break;

      case mSpecial:
        doSpecialMenu(menuItem);
        break;

      case hmSecondItem:
        doSubMenus(menuItem);
        break;

      case kHMHelpMenuID:
        doHelpMenu(menuItem);
        break;
    }

    HiliteMenu(0);
  }

  // ########################################################################## doPopupMenuChoice

  void  doPopupMenuChoice(SInt16 popupItem)
  {
    switch(popupItem)
    {
      case pSydney:
        drawItemString("\pSydney");
        break;

      case pNewYork:
        drawItemString("\pNew York");
        break;

      case pLondon:
        drawItemString("\pLondon");
        break;

      case pRome:
        drawItemString("\pRome");
        break;
    }
  }

  // ############################################################################## doAppleMenu

  void  doAppleMenu(SInt16 menuItem)
  {
    Str255  itemName;
    SInt16  daDriverRefNum;

    if(menuItem == iAbout)
      drawItemString("\pAbout Menus…");
    else
    {
      GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
      daDriverRefNum = OpenDeskAcc(itemName);
    }
  }

  // ############################################################################## doFileMenu

  void  doFileMenu(SInt16 menuItem)
  {
    if(menuItem  == iQuit)
      gDone = true;
  }

  // ############################################################################## doEditMenu

  void  doEditMenu(SInt16 menuItem)
  {
    switch(menuItem)
    {
      case iUndo:
        drawItemString("\pUndo");
        break;
```

```
446
447       case iCut:
448         drawItemString("\pCut");
449         break;
450
451       case iCopy:
452         drawItemString("\pCopy");
453         break;
454
455       case iPaste:
456         drawItemString("\pPaste");
457         break;
458
459       case iClear:
460         drawItemString("\pClear");
461         break;
462     }
463   }
464
465   // ########################################################################## doFontMenu
466
467   void  doFontMenu(SInt16 menuItem)
468   {
469     MenuHandle  fontMenuHdl;
470     Str255      fontName;
471     SInt16      fontNumber;
472
473     fontMenuHdl = GetMenuHandle(mFont);
474
475     CheckItem(fontMenuHdl,gCurrentFont,false);
476     CheckItem(fontMenuHdl,menuItem,true);
477
478     gCurrentFont = menuItem;
479
480     GetMenuItemText(fontMenuHdl,menuItem,fontName);
481     GetFNum(fontName,&fontNumber);
482     TextFont(fontNumber);
483
484     drawItemString(fontName);
485   }
486
487   // ########################################################################## doStyleMenu
488
489   void  doStyleMenu(SInt16 menuItem)
490   {
491     MenuHandle  styleMenuHdl;
492
493     switch(menuItem)
494     {
495       case iPlain:
496         gCurrentStyle = 0;
497         break;
498
499       case iBold:
500         if(gCurrentStyle & bold)
501           gCurrentStyle -= bold;
502         else
503           gCurrentStyle |= bold;
504         break;
505
506       case iItalic:
507         if(gCurrentStyle & italic)
508           gCurrentStyle -= italic;
509         else
510           gCurrentStyle |= italic;
511         break;
512
513       case iUnderline:
514         if(gCurrentStyle & underline)
515           gCurrentStyle -= underline;
516         else
517           gCurrentStyle |= underline;
518         break;
519
520       case iOutline:
521         if(gCurrentStyle & outline)
522           gCurrentStyle -= outline;
```

```
523         else
524           gCurrentStyle |= outline;
525         break;
526
527       case iShadow:
528         if(gCurrentStyle & shadow)
529           gCurrentStyle -= shadow;
530         else
531           gCurrentStyle |= shadow;
532         break;
533     }
534
535     styleMenuHdl = GetMenuHandle(mStyle);
536
537     CheckItem(styleMenuHdl,iPlain,    gCurrentStyle == 0);
538     CheckItem(styleMenuHdl,iBold,     gCurrentStyle & bold);
539     CheckItem(styleMenuHdl,iItalic,   gCurrentStyle & italic);
540     CheckItem(styleMenuHdl,iUnderline,gCurrentStyle & underline);
541     CheckItem(styleMenuHdl,iOutline,  gCurrentStyle & outline);
542     CheckItem(styleMenuHdl,iShadow,   gCurrentStyle & shadow);
543
544     TextFace(gCurrentStyle);
545
546     drawItemString("\pStyle change");
547   }
548
549   // ############################################################################## doSizeMenu
550
551   void  doSizeMenu(SInt16 menuItem)
552   {
553     MenuHandle  sizeMenuHdl;
554
555     switch(menuItem)
556     {
557       case iTen:
558         TextSize(10);
559         break;
560
561       case iTwelve:
562         TextSize(12);
563         break;
564
565       case iEighteen:
566         TextSize(18);
567         break;
568
569       case iTwentyFour:
570         TextSize(24);
571         break;
572     }
573
574     sizeMenuHdl = GetMenuHandle(mSize);
575
576     CheckItem(sizeMenuHdl,gCurrentSize,false);
577     CheckItem(sizeMenuHdl,menuItem,true);
578
579     gCurrentSize = menuItem;
580
581     drawItemString("\pSize change");
582   }
583
584   // ############################################################################ doSpecialMenu
585
586   void  doSpecialMenu(SInt16 menuItem)
587   {
588     if(menuItem == iFirstItem)
589       drawItemString("\pFirst Item");
590   }
591
592   // ############################################################################## doSubMenus
593
594   void  doSubMenus(SInt16 menuItem)
595   {
596     switch(menuItem)
597     {
598       case siFirstSub:
599         drawItemString("\pSubitem 1");
```

```
600         break;
601
602       case siSecondSub:
603         drawItemString("\pSubitem 2");
604         break;
605     }
606   }
607
608   // ############################################################################## doHelpMenu
609
610   void  doHelpMenu(SInt16 menuItem)
611   {
612     MenuHandle  helpMenuHdl;
613     SInt16      origHelpItems, numItems;
614
615     HMGetHelpMenuHandle(&helpMenuHdl);
616
617     numItems = CountMItems(helpMenuHdl);
618     origHelpItems = numItems - 1;
619
620     if(menuItem > origHelpItems)
621       drawItemString("\pMenus Help");
622   }
623
624   // ############################################################################ drawItemString
625
626   void  drawItemString(Str255 eventString)
627   {
628     RgnHandle tempRegion;
629     WindowPtr windowPtr;
630     Rect      scrollBox;
631
632     windowPtr = FrontWindow();
633     tempRegion = NewRgn();
634
635     scrollBox = windowPtr->portRect;
636     scrollBox.top = scrollBox.top + 50;
637
638     ScrollRect(&scrollBox,0,-24,tempRegion);
639     DisposeRgn(tempRegion);
640
641     MoveTo(8,286);
642     DrawString(eventString);
643   }
644
645   // ##############################################################################################
```

Demonstration Program Comments

When this program is run, the user should make menu selections from all menus, including the Apple menu, the Help menu and the pop-up menu. Selections should be made using the mouse and, where appropriate, the Command key equivalents. The user should also note the effects on the menu bar of clicking outside, then inside, the program's window, that is, of sending the program to the background and returning it to the foreground.

#define

Lines 59-86 establish constants relating to the pull-down and hierarchical menu IDs and resources, menu item numbers and subitem numbers. The constant at Line 87 represents the resource ID of the 'CNTL' resource associated with the popup menu, and Lines 88-91 represent the item numbers of the items in this menu. The constant at Line 92 represents the resource ID for the 'WIND' resource.

Golbal Variables

The global variable gDone relates to the main event loop. When set to true, the loop will exit and the program will terminate. The remaining three global variables will hold the current choices, in terms of item numbers, from the Font, Style and Size menus.

main

The main() function initialises the system software managers (Line 132), creates a window and makes its graphics port the current port (Lines 136-142), calls the application-defined function which sets up the menus (Line 146), shows the window and pop-up menu (Lines 147-148) and enters the main event loop (Lines 153-159)

doGetMenus

doGetMenus sets up the menu bar and the various menus.

At Line 189, GetNewMBar reads in the 'MENU' resources for each menu specified in the 'MBAR' resource and creates a menu record for each of those menus. (Note that the error handling here and in other areas of this program is somewhat rudimentary: the program simply terminates (Lines 190-191).) At Lines 192-193, SetMenuBar makes the newly created menu list the current list and DrawMenuBar draws the menu bar.

Lines 195-199 add the contents of the Apple Menu Items folder to the Apple menu. The use of 'DRVR' as the second parameter to the AppendResMenu call is automatically interpreted to mean that the Apple menu is being created, so that all items in the Apple Menu Items folder are added rather than resources of type 'DRVR'.

Lines 201-205 add the names of all resident fonts to the Font menu. Using 'FONT' in the second parameter in the call to AppendResMenu causes all such resources to be searched out and their names added to the specified menu.

Lines 207-211 insert the application's single submenu into the submenu portion of the menu list. GetNewMBar does not read in the resource descriptions of submenus, so the first step is to read in the 'MENU' resource with GetMenu. InsertMenu inserts a menu record for this menu into the menu list at the location specified in the second parameter to this call. Using the constant hierMenu (-1) as the second parameter causes the menu to be installed in the submenu portion of the menu list.

Lines 213-217 append a menu item with the name "Menus Help" to the Help menu.

Line 219 sets up the popup menu. GetNewControl loads the specified 'CNTL' resource into a control record and creates the control in the specified window. (The 'CNTL' resource specifies popUpMenuProc in the procID field, so the control is created as a popup menu. The Min field of the 'CNTL' resource description contains the 'MENU' resource ID for the popup menu, which ensures that the GetNewControl call will load the popup menu resource and create a menu record in the submenu portion of the menu list.)

Lines 223-225 set checkmarks against the appropriate font, style and size menu items according to the initialised values of the associated global variables.

doEvents

doEvents switches according to the type of low-level or Operating System event received. Further processing is called for in the case of mouse-down or Command key equivalents, these being central to the matter of menu handling.

In the case of key-down and auto-key events, the character code is first extracted from the event record's message field (Line 242). A check is then made of the modifiers field to establish whether the Command key was also pressed at the time (Line 243). If so, menu enabling/disabling is attended to (Line 245) before the call to MenuKey (Line 246) establishes whether the character code is associated with a currently enabled menu or submenu item in the menu list. If a match is found, MenuKey returns a long integer containing the menu ID in the high word and the item number in the low word, otherwise it returns 0 in the high word. This long integer is then passed to the function doMenuChoice.

The call to HiliteMenu at Line 256 is made to unhighlight the Apple menu title when the user brings the demonstration program to the foreground having previously sent it to the background by choosing an Apple Menu Items folder item from the Apple menu.

doMouseDown

doMouseDown first establishes the window and window part in which the mouse-down event occurred (Line 269), and switches accordingly. This demonstration program is specifically interested in mouse-downs in the menu bar and the content region of the window, the latter because the pop-up menu is located in the window.

Lines 273-275 pass mouse-downs in a system window to SystemClick for further handling.

If the event occurred in this program's menu bar (Line 277), menu enabling/disabling is attended to (Line 278) before the call to MenuSelect (Line 279). MenuSelect tracks the user's

actions until the mouse button is released, at which time it returns a long integer. If the user actually chose a menu item, this long integer contains the menu ID in the high word and the item number in the low word, otherwise it contains 0 in the high word. At Line 280, this long integer is passed to the function doMenuChoice.

If the mouse-down event occurred in the content region of the window (Line 283), and if the window to which the mouse-down refers is not the front window, SelectWindow is called to effect basic window activation/deactivation (Lines 284-285). If, however, the window receiving the mouse-down is the front window (Line 286), Line 288 calls an application-defined function which checks whether the cursor was within the pop-up's control rectangle and, if it was, returns the control's value if the user actually chose a pop-up menu item. If the user actually chose a pop-up menu item (that is, if the received value was non-zero), the control's value is passed to an application-defined function which handles the choice (Lines 289-290). (The control's value equates to the chosen menu item's number.)

Lines 294-296 respond to a mouse-down in the drag bar. Not that, if the window is dragged to a new position, the pop-up menu will be redrawn automatically, together with the window, with no assistance from the program.

Lines 298-301 respond to a mouse-down in the go-away box, setting gDone to true and thus terminating the program if the cursor is still within the go-away box when the mouse button is released.

doCheckForControlAndValue

doCheckForControlAndValue ascertains whether a mouse-down event occurred within the pop-up menu's control rectangle and, if so, whether the user actually chose an item from the menu.

Line 312 ensures that the window's graphics port is set as the current port. Line 313 then converts the contents of the event record's where field from global to local coordinates, that is, to the coordinate system of the current graphics port.

These local coordinates are required in the call to FindControl at Line 315. This call establishes whether the event occurred in the popup menu control or elsewhere in the content region. If an active control was located at the point specified in its first parameter, FindControl will receive a handle to that control into its third parameter and return a part code, otherwise it will return 0.

If an active control is detected, it can be safely assumed in this program that it is the pop-up menu's control. (The window contains no other controls.) Accordingly, the control's current value is saved at Line 317 preparatory to handling over control to TrackControl at Line 318. TrackControl tracks user action until the mouse button is released.

Note the third parameter of the TrackControl call. The pop-up menu control definition function contains code which is referred to as an "action procedure", and which is invoked repeatedly as long as the mouse button remains down. This action procedure will not be invoked unless TrackControl's third parameter, in calls relating to pop-up menus, is set to (ControlActionUPP) -1. (Action procedures are addressed at Chapter 5 — Controls, and action procedures within control definition functions are addressed at Chapter 19 — Custom Control Definition Functions and VBL Tasks.)

On release of the mouse button, Line 319 gets the control's new value. This will only have been changed by the pop-up control definition function if the user did not release the mouse button with the cursor outside the menu and if the user actually chose a new menu item. If the control's value before and after the TrackControl call differs, Lines 322-323 cause the control's new value to be returned to the calling function. (For pop-up menus, the control's value equates to the item number of the menu.) If there is no difference, Lines 324-325 return zero to the calling function, which will defeat the calling by that function of further application-defined pop-up menu handling functions.

doAdjustMenus

doAdjustMenus is called when a mouse-down occurs in the menu bar and when examination of a key-down event reveals that a menu item's keyboard equivalent has been pressed. No action is taken in this simple program because only one window, whose content never changes, is ever open.

(Later demonstration programs contain examples of menu adjustment functions which cater for specific circumstances. For example, the menu adjustment function in the demonstration program at Chapter 6 — Dialogs and Alerts accommodates the situation where the front window could be either a document window, a movable modal dialog box, or a modeless dialog box.)

doMenuChoice

doMenuChoice takes the long integer returned by the MenuSelect and MenuKey calls, extracts the high word (the menu ID) and the low word (the menu item number) and switches according to the menu ID.

At lines 341-342, the menu ID and the menu item number are extracted from the long integer. Lines 344-345 will cause an immediate return if the high word equals 0, (meaning that either the mouse button was released when the pointer was outside the menu box or MenuKey found no menu list match for the key pressed in conjunction with the Command key).

Lines 347-384 switch according to the menu ID, calling the appropriate application-defined individual menu handling function. Note the handling of the hierarchical menu at Lines 377-379. Note also that, at Line 381, the kHMHelpMenuID constant (-16490) is returned in the high word if the user chooses an appended item from the Help menu.

MenuKey and MenuSelect leave the menu title highlighted if an item was actually selected. Accordingly, Line 386 unhighlights the menu title when the action associated with the user's drop-down menu choice is complete.

doPopupMenuChoice

doPopupMenuChoice switches according to the short integer returned by GetCtlValue, which represents the popup menu item number chosen by the user. This function completes the popup menu handling in this demonstration.

doAppleMenu

doAppleMenu takes the short integer representing the menu item. If this value represents the first item in the Apple menu (the inserted "About…" item), text representing this item is drawn in the scrolling display (Lines 420-421).

If the value passed to the doAppleMenu function represents other items in the Apple menu (Line 422), the call to GetMenuItemText at Line 424 gets the string representing the item's name. This string (which excludes metacharacters) is used as the parameter in the OpenDeskAcc call at Line 425. OpenDeskAcc opens the chosen object and passes control the chosen object.

doFileMenu

doFileMenu handles selections from the File menu. In this demonstration, only the Quit item is enabled, all other items having been disabled in the File menu's 'MENU' resource. When this item is chosen, the global variable gDone is set to true (Line 434), causing termination of the program.

doEditMenu

doEditMenu switches according to the menu item number, drawing text representing the chosen item in the window.

doFontMenu

doFontMenu first gets a handle to the Font menu record (Line 473) required by the CheckItem calls at Lines 475-476. The CheckItem calls uncheck the current font menu item and check the menu item passed to the doFontMenu function. This latter menu item number is then assigned to the gCurrentFont global variable (Line 478).

The call to GetMenuItemText at Line 480 extracts the string representing the item's name. This string is passed as the first parameter in the call to GetFNum (Line 481), which gets the font number associated with the name. This number is then used in the call to TextFont at Line 482, which will cause subsequent text drawing to be conducted in the specified font. Line 484 draws the name of the font in that font

doStyleMenu

doStyleMenu switches according to the menu item chosen in the Style menu. Lines 493-533 cause bits in the global variable gCurrentStyle to be set or unset according to the font styles selected. The code reflects the fact that Bold, Italic, Underline, Outline and Shadow style selections are additive, not mutually exclusive, and that a selection of Plain must unset all bits in gCurrentStyle. The code also reflects the requirement that, except in the case of the Plain item, the selection of a checked item must cause that item to be unchecked, and vice versa.

With the appropriate bit settings of gCurrentStyle attended to, a handle to the Style menu record is then obtained (Line 535). This is required for the CheckItem calls at Lines 537-543, which check or uncheck the individual menu items according to whether the third argument evaluates to, respectively, true or false.

At Line 544, the call to TextFace sets the style for subsequent text drawing. Line 546 draws some text to prove that the desired effect was achieved.

doSizeMenu

doSizeMenu switches according to the menu item chosen in the Size menu, sets the text size for all text drawing to that size (555-572), unchecks the current size item (Line 576) and checks the newly selected item (Line 577). gCurrentSize is set to the selected menu item number (Line 579) before the function returns.

doSpecialMenu

doSpecialMenu handles a selection of the first item in the Special menu. Since the second item is the title of a submenu, only the first item is attended to in this function.

doSubMenus

doSubMenus switches according to the chosen subitem in the hierarchical menu represented by the second menu item in the Special menu.

doHelpMenu

doHelpMenu handles the selection of the "Menus Help" item added by this program to the system-managed Help Menu. This code reflects the fact that Apple reserves the right to add items to the Help menu in future versions of the system software.

At Line 615, HMGetHelpMenuHandle gets a handle to the Help menu record. At Line 617, the call to CountMItems returns the number of items in the Help menu. Since we know that we have added one item to this menu, Line 618 will establish the original number of help items. If the value passed to the doHelpMenu function is greater than this (Line 620), it must therefore represent the item number of our "Menus Help" item, in which case some text is drawn in the window to register the fact (Line 621).

drawItemString

The function drawItemString is incidental to the demonstration, being called by the menu selection handling functions to draw text in the application's window to reflect the user's menu choices. It is similar to the function drawEventString in the demonstration program at Chapter 2 — Low-Level and Operating System Events.

Creating 'MBAR' and 'MENU' Resources Using ResEdit

When learning to create the major resource types in ResEdit, it is recommended that you open Macintosh C to the page containing the relevant example resource definition in Rez input format and relate what you are doing within ResEdit to that definition. Accordingly, the methodology used in the following is to "walk through" the 'MBAR' and 'MENU' resources for the Menus demonstration program, relating what you see in ResEdit to the example definitions in this chapter.

Open the chap03cw_demo demonstration program folder and double-click on the Menus.µ.rsrc icon to start ResEdit and open Menus.µ.rsrc. The **Menus.µ.rsrc** window opens.

'MBAR' Resources

Double-click the MBAR icon. The **MBARs from Menus.µ.rsrc** window opens. One 'MBAR' resource (ID 128) appears in the list in the window. Double-click that list entry. The **MBAR ID = 128 from Menus.µ.rsrc** window opens.

The following relates the example 'MBAR' resource in Rez input format in this chapter to the ResEdit display and interface:

`resource 'MBAR'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **MBAR** was clicked, and the dialog's **OK** button was clicked.
`(rMenuBar,`	Choose **Resource/Get Resource Info**. The **Info for MBAR 128 ...** window opens. Note the editable text text item **ID:**. (ResEdit automatically assigns 128 as the ID of the first `'MBAR'` resource you create.)
`preload)`	Also note, in the **Attributes** section, that the **Preload** checkbox is checked. Close the **Info for MBAR 128 ...** window.

`mApple,`
`mFile,`
`mEdit`
	Back in the **MBAR ID = 128 from Menus.µ.rsrc** window, note entries 1), 2), and 3).
	MENU resource IDs are added by clicking on the next entry number (e.g., 8) *****), choosing **Resource/Insert New Field**, and entering the resource ID at the associated Menu res ID editable text item.
	MENU resources may be deleted by clicking the entry number (e.g. 8) *****) and choosing **Edit/Cut**.

Close the **MBAR ID = 128 from Menus.µ.rsrc** window. **Close the MBARs from Menus.µ.rsrc** window.

'MENU' Resources

Double-click the MENU icon. **The MENUs from Menus.µ.rsrc** window opens. Double-click the Apple menu icon (`'MENU'` resource ID 128). The **MENU ID = 128 from Menus.µ.rsrc** window opens.

The following relates the example `'MENU'` resource in Rez input format in this chapter to the ResEdit display and interface:

`resource 'MENU'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **MENU** was clicked, and the dialog's **OK** button was clicked.
`(mApple,`	`mApple` is the resource ID (128). Choose **Resource/Get Resource Info**. The **Info for MENU 128 ...** window opens. Note the editable text item titled **ID:**. This is where you set the resource ID. ResEdit automatically assigns 128 as the resource ID of the first `'MENU'` resource you create.
`preload)`	While the **Info for MENU 128 ...** window is open, compare the **Attributes** check boxes to the Resource Attributes table at Chapter 1. Note that the **Preload** checkbox is checked.
`mApple,`	Close the **Info for MENU 128 ...** window and choose **MENU/Edit Menu & MDEF ID**. A dialog box opens. Note the editable text item titled **Menu ID**. Note that the Menu ID is the same as the `'MENU'` resource ID displayed in the **Info for MENU 128 ...** window. ResEdit automatically makes the MENU ID the same as the `'MENU'` resource ID, although you can assign a different MENU ID here if you want to.
`textMenuProc,`	Also note that **0** appears in the editable text item titled **MDEF ID**. This means that the standard menu definition function is specified. Choose **Cancel** to close the dialog.
`0b11111111111 ...,`	At the left of the **MENU ID = 128 ...** window, click, in turn, on the **About Menus...** item and the **separator-line** item, and observe the checkbox titled **Enabled** at the right of the window.
`enable,`	Click on the menu title (the apple icon) and observe the checkbox titled **Enabled** at the right of the window.
`apple,`	Click on the menu title (the apple icon) and note the radio button titled **Apple menu** under the editable text item at the right of the window.

`"About ... ",`	Click on the **About Menus...** item in the list at the left of the window and note that you can edit this in the editable text item at the right of the window.
`noIcon,`	Choose **MENU/Choose an Icon**. A **Choose an icon ...** dialog opens. (Simply note this for now. The matter of icons in menu items is addressed at Chapter 12). Close the dialog.
`noKey,`	Note the small editable text item titled **Cmd-Key** at bottom right. This is where you enter the Command key equivalent for a menu item.
`noMark,`	Note the popup menu titled **Mark** at bottom right. This is where you can specify the mark to be inserted into the menu item.
`plain;`	Open the **Style** menu. (Note that this menu has been used to set the style of the items in the **Style** menu (`'MENU'` resource ID 132).
`-,`	Click on the separator-line item and note the radio button titled **(separator line)** under the editable text item at the right of the window.

`noIcon,`
`noKey,`
`noMark,`
`plain`
	(As above)

Note that, when you click on the menu's title at the left of the window, three **Color** pop-ups appear at the bottom right of the **MENU ID = ...** window. If you use these pop-ups to specify colours for the title, item text and/or background, ResEdit automatically creates a `'mctb'` (menu color table) resource with the same resource ID as the associated `'MENU'` resource.

Hierarchical Menu

At the example hierarchical menu resource in Rez input format in this chapter, note the line beginning `Label Style` and the following description of the associated submenu.

With the **MENUs from Menus.µ.rsrc** window open, double click the **Special** menu icon (`'MENU'` resource ID 134). The **MENU ID = 134 ...** window opens. Click on the item **Second Item**. Note that the **has Submenu** checkbox is checked and that the **ID:** box shows an ID of 100.

Close the **MENU ID = 134 ...** window. In the **MENUs from Menus.µ.rsrc** window, note the `'MENU'` resource with ID 100. Also note that the name of the menu item **Second Item** is also the title of the `'MENU'` resource with ID 100.

Pop-up Menu

With the **MENUs from Menus.µ.rsrc** window open, note the `'MENU'` resource with ID 135.

The matter of `'CNTL'` resources for pop-up menus is addressed at Chapter 5 — Controls.