Chapter 9

Menus

Menus are as important to the Windows user interface as they are on the Macintosh. On both systems, they are the principal means by which the user can issue commands and direct the actions of the program. With a few minor differences, Windows and Macintosh menus look and feel pretty much the same to the user: they pop up when you click their title in the menu bar, offering a set of actions or options from which to choose by pointing with the mouse. From the programmer's point of view, menus are integrated into the Windows message mechanism, simplifying the task of decoding and responding to the user's menu choices. In this chapter, we'll learn the basics of how Windows menus work and how to handle them in your programs.

Windows menus come in two main flavors. A *top-level menu* corresponds to the Macintosh menu bar: a horizontal series of text items that typically represent the titles of individual menus (see Figure 9–1). In fact, a top-level menu is often referred to informally in Windows as a "menu bar," though it's actually just a special type of menu that happens to be displayed horizontally on the screen instead of vertically. Note that, unlike its Macintosh counterpart, a top-level menu always belongs to a particular window and appears below that window's title bar, rather than globally at the top of the screen. Only a top-level window—not a child—can have a menu bar. Not every window has to have a menu bar, but every menu bar must belong to a window.

What Macintosh programmers normally think of as a menu is called a *pop-up menu* (or *submenu*) in Windows (Figure 9–2). This type of menu remains hidden from view until the user clicks the mouse in its title. The menu then "pops up" on the screen, allowing the user to choose one of its items with the mouse. Unlike those on the Macintosh, a Windows menu remains on the screen even after the user releases the mouse button: it isn't necessary to hold down the button continuously in order to keep the menu visible. If the user rolls the mouse along the menu bar to another menu's title, the first menu will disappear and the second menu will pop up in its place, even without a further mouse click. The user must click the mouse a second time to make the menu go away: either by clicking in one of its items (choosing that item) or by clicking anywhere outside the menu (dismissing it without choosing an item).

Figure 9–1. A top-level menu

Figure 9–2. A pop-up menu

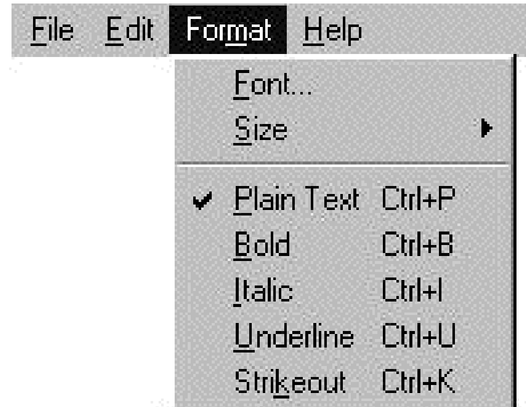


Figure 9–2 shows the three types of item can appear on a menu:

- A *command item* invokes an action of some sort by issuing a message to the window procedure of the window that owns the menu.
- A *pop-up item* is actually the title of another menu; choosing it with the mouse causes the associated menu to pop up on the screen.
- A *separator* is a horizontal line used to separate the menu's items into groups for easier readability. It is not an active item and does not respond to the mouse.

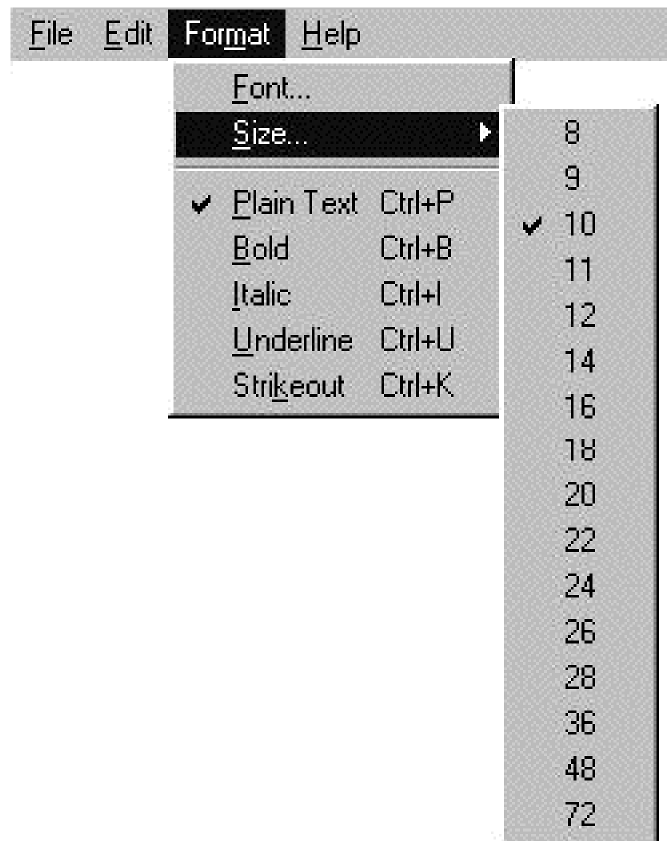
Only a pop-up menu can contain separators, but the other two types can appear on both top-level and pop-up menus. In the menu bar, pop-up items work just like the menu titles in the Macintosh menu bar: they display a pop-up menu that hangs down below the title, as shown in Figure 9–2. In pop-up menus, they work like hierarchical menus on the Macintosh, bringing up a submenu to the right of the original menu (Figure 9–3); such hierarchical submenus can be nested to any depth. As on the Macintosh, a right-pointing triangle symbol to the right of the item cues the user that the item invokes a submenu rather than a command. One difference from the Macintosh is that command items, too, can appear in both top-level and pop-up menus. Thus, although the menu bar ordinarily contains nothing but pop-up menu titles, it can also, in unusual cases, contain items that activate an immediate command instead of a menu.

Individual menu items can be placed in any of three states:

- An *enabled* item can be chosen with the mouse and will respond by invoking a command or displaying a submenu, as the case may be.
- A *disabled* item is displayed in the normal way, but does not respond to the mouse.
- A *grayed* item is displayed in gray instead of black and does not respond to the mouse.

By default, all menu items are normally enabled. As on the Macintosh, you should gray out any items that are temporarily inappropriate or inapplicable in a given situtation, such as a **Cut** or **Copy** command when there is no current selection or a **Save** command when the window doesn't contain any unsaved changes. Disabling is more appropriate for items that are intended to be permanently inactive, such as a title introducing a group of related items on the menu. Grayed items highlight in the normal way when the user points to them with the mouse, but cause no action to occur if the mouse is clicked; disabled items are completely inert and do not even highlight when the mouse rolls over them. The Windows function **EnableMenuItem** sets the state of a menu item enables, disables, or grays a menu item.

Figure 9–3. A hierarchical pop-up menu



As on the Macintosh, a menu item can be marked with a check mark, or *checked,* to show that it is currently in effect (see Figure 9–2). You can use the Windows function **CheckMenuItem** to check or uncheck an item and **GetMenuState** to find out whether it is currently checked or unchecked. The standard check mark symbol shown in the figure is used by default, but it is possible to substitute a custom symbol of your own with a Windows function called **SetMenuItemBitmaps**.

One particularly important pop-up menu is the *system menu* (Figure 9–4), which appears when the user clicks the small program icon at the left of a window's title bar. The system menu typically lists commands for manipulating the window on the

screen, such as **`Move`**, **`Size`**, **`Minimize`**, **`Maximize`**, **`Restore`**, and **`Close`**, which are normally handled by the Windows system itself through its default window procedure. A program is free to intercept and respond to these commands itself, or to add further commands of its own to the system menu, but most programs just let the system handle all interactions with this menu in its own way.

Figure 9–4. The system menu

Not every pop-up menu has to be anchored to a pop-up item in the menu bar or another menu. It's also possible to create a *floating pop-up menu* that pops up right at the mouse location in response to some action by the user, typically a click of the right mouse button. The contents of the resulting menu can vary depending on the context, circumstances, and area of the screen in which the mouse is pressed: for example, many programs display the system menu as a floating pop-up when the user clicks the right button in a window's title bar.

As on the Macintosh, you can either build your menus "by hand," using system functions provided for the purpose, or load them from the disk as resources. To build a menu from scratch, you begin by calling either the Windows function **`CreateMenu`** to create an empty menu bar, or **`CreatePopupMenu`** to create an empty pop-up menu. Both functions return a result of type **`HMENU`**, a handle to a menu. You can then add items to the menu one by one, using either **`AppendMenu`** to add a new item at the end of the menu or **`InsertMenu`** to insert it before another specified item.

One of the parameters you supply to both **`AppendMenu`** and **`InsertMenu`** is a *menu item identifier,* a 16-bit integer that will uniquely identify the item in all communications between your program and the Windows system. Item identifiers are purely arbitrary numbers that you define for yourself: they aren't assigned automatically by the system, nor do they bear any necessary relation to the item's position on the menu. Listing 9–1 shows the item identifiers used by our WiniEdit program, taken from the header file **`WiniEdit Resources.h`**.

Listing 9–1. WiniEdit menu-item identifiers

```
#define Main_Menu                      1000

#define File_Menu                   0
#define    New_Item                    1001
#define    Open_Item                   1002
#define    Close_Item                  1003
#define    Save_Item                   1004
#define    SaveAs_Item                 1005
#define    Revert_Item                 1006
#define    Setup_Item                  1007
#define    Print_Item                  1008
#define    Exit_Item                   1009

#define Edit_Menu                   1
#define    Undo_Item                   1101
#define    Cut_Item                    1102
#define    Copy_Item                   1103
#define    Paste_Item                  1104
#define    Delete_Item                 1105
#define    SelectAll_Item              1106

#define Format_Menu                 2
#define    Format_Item                 1201
#define    Default_Item                1202
#define    Background_Item             1203

#define Help_Menu                   3
#define    Help_Item                   1301
#define    About_Item                  1302
```

Besides the item identifier, both **AppendMenu** and **InsertMenu** accept a menu handle identifying the menu to which the new item is to be added, along with a flag word describing the item's characteristics, as shown in Table 9–1. **InsertMenu** takes an additional parameter specifying where in the menu the new item is to be inserted. Finally, both functions accept a parameter defining the content of the new menu item, depending on its type: a character string for a text item, a bitmap for a graphical item, or a 32-bit data value to be passed to the drawing routine for a custom item.

Although you can certainly build all your menus by hand if you want to, it's generally a better idea to read them in as resources. A single *menu template* resource defines a top-level menu and all of its submenus, filling the roles of both **'MBAR'** and **'MENU'** resources on the Macintosh. I built all of WiniEdit's menus directly onscreen, using the interactive menu editor from the Visual C++ development environment. The editor produced the resource description shown in Listing 9–2, in the description language expected by the Visual C++ resource compiler. The resource compiler then compiled the description into a menu template that can be read into memory with the Windows function **LoadMenu**.

Table 9–1. Menu-item style options

Style name	Meaning
MF_STRING	Text item
MF_BITMAP	Graphical item
MF_OWNERDRAW	Custom item (drawn by application)
MF_POPUP	Pop-up item (invokes another menu)
MF_SEPARATOR	Separator line
MF_ENABLED	Item is enabled
MF_DISABLED	Item is disabled
MF_GRAYED	Item is grayed
MF_CHECKED	Item is checked
MF_UNCHECKED	Item is unchecked
MF_MENUBREAK	Item starts new line (in menu bar) or new column (in pop-up menu)
MF_MENUBARBREAK	Same as **MF_MENUBREAK**, but with vertical separator line between columns in pop-up menu

Once you've created a top-level menu, you have to associate it with a window. There are three ways of doing this:

- Make it the default menu for the window's class. You do this by storing the menu's resource name or ID into the **lpszMenuName** field of the **WNDCLASS** structure that you pass to **RegisterClass**. This is how WiniEdit does it, in its **InitProgram** routine:

```
resourceID = MAKEINTRESOURCE(Main_Menu);     // Convert resource ID to string
windowClass.lpszMenuName = resourceID;       // Set menu
```

- Specify it as a parameter at the time the window is created. To do this, you must first create the menu and obtain a handle to it, either by building it from scratch with **CreateMenu**, **AppendMenu**, and **InsertMenu** or by loading it from a template resource with **LoadMenu**. Once you have a handle to an existing menu, you can pass it as a parameter to **CreateWindow**; if this parameter is non-null, it overrides the default menu associated with the window's class.
- Associate it directly with an existing window with the **SetMenu** function.

You can obtain a handle to a window's top-level menu with **GetMenu**, to a pop-up menu with **GetSubMenu**, or to the system menu with **GetSystemMenu**. Destroying a window automatically destroys its menu (including all submenus) along with it, so there's no need to dispose of the menu explicitly. A menu not associated with any window, such as a floating pop-up, must be destroyed with **DestroyMenu** when you're through with it; otherwise it will continue to take up space in memory even after your program terminates.

Listing 9–2. WiniEdit menu resource description

```
Main_Menu MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",                New_Item
        MENUITEM "&Open...\tCtrl+O",            Open_Item
        MENUITEM "&Close\tCtrl+W",              Close_Item
        MENUITEM SEPARATOR
        MENUITEM "&Save\tCtrl+S",               Save_Item
        MENUITEM "Save &As...\tCtrl+Alt+S",     SaveAs_Item
        MENUITEM "&Revert to Saved...\tCtrl+R", Revert_Item
        MENUITEM SEPARATOR
        MENUITEM "Page Set&up...\tCtrl+Alt+P",  Setup_Item
        MENUITEM "&Print...\tCtrl+P",           Print_Item
        MENUITEM SEPARATOR
        MENUITEM "E&xit\tCtrl+Q",               Exit_Item
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl+Z",               Undo_Item
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl+X",                Cut_Item
        MENUITEM "&Copy\tCtrl+C",               Copy_Item
        MENUITEM "&Paste\tCtrl+V",              Paste_Item
        MENUITEM "&Delete\tDelete",             Delete_Item
        MENUITEM SEPARATOR
        MENUITEM "Select &All\tCtrl+A",         SelectAll_Item
    END
    POPUP "For&mat"
    BEGIN
        MENUITEM "Text &Format...\tCtrl+F",     Format_Item
        MENUITEM "&Default Format\tCtrl+D",     Default_Item
        MENUITEM SEPARATOR
        MENUITEM "&Background Color...\tCtrl+B", Background_Item
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help\tCtrl+?",               Help_Item
        MENUITEM SEPARATOR
        MENUITEM "&About WiniEdit...",          About_Item
    END
END
```

Table 9–2 summarizes the menu-related messages that a window can receive. When the user clicks the mouse in the window's menu bar, Windows sends the message **WM_ENTERMENULOOP** to begin tracking the mouse for menu selection (roughly analogous to the Macintosh Toolbox's **MenuSelect** routine). This message allows you to get control at the beginning of the mouse-tracking loop in case you want to customize the menu's mouse-tracking behavior in some way. Theoretically, you could intercept this message and provide your own tracking loop; but in reality, you will almost certainly pass it to the default window procedure to track the mouse in the standard way. A companion message, **WM_EXITMENULOOP**, serves as a tail hook in case you want to do some kind of final customization after the button is released; again, most real-world programs simply ignore this message.

Table 9–2. Menu-related messages

Message type	Meaning
WM_INITMENU	Top-level menu becoming active
WM_INITMENUPOPUP	Pop-up menu about to be displayed
WM_MEASUREITEM	Find dimensions of custom menu item
WM_DRAWITEM	Draw custom menu item
WM_ENTERMENULOOP	Entering menu tracking loop
WM_EXITMENULOOP	Leaving menu tracking loop
WM_MENUCHAR	Unknown keyboard shortcut received
WM_MENUSELECT	Menu item selected
WM_COMMAND	Menu command chosen
WM_SYSCOMMAND	System menu command chosen

As the mouse rolls over each item in the menu, Windows sends a **WM_MENUSELECT** message for the item. The default window procedure responds by highlighting the item on the screen and, if it's a pop-up item, displaying the submenu associated with it. (Remember that the menu bar itself is the top-level menu, so even the first tier of pop-ups, the ones whose titles appear in the menu bar, are considered submenus.) You can intercept this message and customize the highlighting, if you like, or use the occasion to provide some other form of useful feedback such as a brief command description in a "status bar" at the bottom of your window.

Before displaying a pop-up menu, Windows sends you the message **WM_INITMENUPOPUP**. This gives you a chance to enable or gray out items on the menu according to the current context and circumstances, before they're presented to the user. The message's parameters identify the menu, the pop-up item that invoked it, and whether it's the system menu or one of your own. (A similar message, **WM_INITMENU**, allows you to modify the contents of the menu bar itself at the very beginning of the tracking loop, when the mouse button is first pressed in it —but it's hard to think of a convincing way to use it.)

Listing 9–3. Handle `WM_INITMENUPOPUP` message

```
VOID DoInitMenuPopup (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_INITMENUPOPUP message.

   {
      HMENU  theMenu   = HMENU(wParam);   // Handle to menu to be adjusted
      UINT   menuIndex = LOWORD(lParam);     // Relative position of menu in menu bar
      BOOL   isSystem  = HIWORD(lParam);     // Is it the system menu?


      if ( isSystem )                     // Is it the system menu?
         FixSystemMenu ();                   // Enable/disable system menu commands
      else
         switch ( menuIndex )
            {
               case File_Menu:
                  FixFileMenu (theMenu);   // Enable/disable File menu commands
                  break;

               case Edit_Menu:
                  FixEditMenu (theMenu);   // Enable/disable Edit menu commands
                  break;

               case Format_Menu:
                  FixFormatMenu (theMenu); // Enable/disable Format menu commands
                  break;

               case Help_Menu:
                  FixHelpMenu (theMenu);   // Enable/disable Help menu commands
                  break;

            }  /* end switch ( menuIndex ) */

   }  /* end DoInitMenuPopup */
```

The parameters to the **`WM_INITMENUPOPUP`** message identify the menu being displayed, the item that invoked it, and whether it’s the system menu or one of your own. WiniEdit’s routine for handling this message, **`DoInitMenuPopup`** (Listing 9–3), simply extracts the identifying information from the parameters and then dispatches on the item ID to a specialized routine for initializing that particular menu. Listing 9–4 shows one of these routines, **`FixEditMenu`**, for illustration. This routine asks the window’s edit control for the length of the text displayed in the window, the length of the current selection, and whether the last editing operation can be undone. It uses this information to decide whether to enable or gray the **`Undo`**, **`Cut`**, **`Copy`**, **`Delete`**, and **`Select All`** commands; for the **`Paste`** command, it checks for the availability of text data on the clipboard. Other WiniEdit routines perform the equivalent task of enabling and graying items, according to current conditions, on the **`File`**, **`Format`**, and **`Help`** menus.

Listing 9–4. Enable/disable `Edit` menu commands

```
VOID FixEditMenu (HMENU theMenu)

   // Enable/disable Edit menu commands.

   {
      BOOL  canUndo;                                              // Can editor support Undo command?
      LONG  selStart;                                          // Character position at start of selection
      LONG  selEnd;                                            // Character position at end of selection
      BOOL  canPaste;                                          // Is there text on the clipboard?
      LONG  textLength;                                           // Number of characters in document


      canUndo = SendMessage(TheEditor, EM_CANUNDO, 0, 0);         // Is Undo available?
      if ( canUndo )
         EnableMenuItem (theMenu, Undo_Item, MF_ENABLED);         // Enable Undo command
      else
         EnableMenuItem (theMenu, Undo_Item, MF_GRAYED);          // Gray out Undo command


      SendMessage ( TheEditor, EM_GETSEL,                         // Get selection range
                    WPARAM(&selStart), LPARAM(&selEnd) );
      if ( selStart != selEnd )                                   // Is there a selection?
         {
            EnableMenuItem (theMenu, Cut_Item,    MF_ENABLED);      // Enable Cut command
            EnableMenuItem (theMenu, Copy_Item,   MF_ENABLED);      // Enable Copy command
            EnableMenuItem (theMenu, Delete_Item, MF_ENABLED);      // Enable Delete command

         }  /* end if ( selStart == selEnd ) */
      else
         {
            EnableMenuItem (theMenu, Cut_Item,    MF_GRAYED);     // Gray out Cut command
            EnableMenuItem (theMenu, Copy_Item,   MF_GRAYED);     // Gray out Copy command
            EnableMenuItem (theMenu, Delete_Item, MF_GRAYED);     // Gray out Delete command

         }  /* end else */


      canPaste = IsClipboardFormatAvailable(CF_TEXT);             // Does clipboard contain text?
      if ( canPaste )
         EnableMenuItem (theMenu, Paste_Item, MF_ENABLED);        // Enable Paste command
      else
         EnableMenuItem (theMenu, Paste_Item, MF_GRAYED);         // Gray out Paste command


      textLength = GetWindowTextLength(TheEditor);                // Get length of text
      if ( textLength > 0 )                                    // Any text in document?
         EnableMenuItem (theMenu, SelectAll_Item, MF_ENABLED);      // Enable Select All command
      else
         EnableMenuItem (theMenu, SelectAll_Item, MF_GRAYED);     // Gray out Select All command


   }  /* end FixEditMenu */
```

Two more menu messages, **WM_MEASUREITEM** and **WM_DRAWITEM**, are used for drawing custom menu items. (You may recall from Chapter 7 that these same two messages are also used for custom controls.) Most typical menus consist entirely of text items, which Windows already knows how to draw. Graphical items also present no problem, since they carry their own bitmaps with them. If you want to use custom items in your menus, however, then of course it's up to you to draw them yourself. Before drawing such an item for the first time, Windows sends you the message **WM_MEASUREITEM**, asking you to supply the item's dimensions so it knows how much space to reserve on the menu. Then, each time the item has to be drawn, Windows sends you a **WM_DRAWITEM** message. One of the parameters to this message is a data structure containing all the information needed to draw the item, including a handle to a device context. Your window procedure must respond to the message by drawing the item in the given context. Customizing menu items is an unusual thing for a program to do, and we won't spend any time on it here; if you're interested, you can find more information in the *Win32 Programmer's Reference.*

Listing 9–5. Handle WM_COMMAND message

```
VOID DoCommand (HWND thisWindow, WPARAM wParam, LPARAM lParam)

   // Handle WM_COMMAND message.

   {
      UINT  notifyCode = HIWORD(wParam);                    // Notification code from child control
      UINT  itemID     = LOWORD(wParam);                    // Item ID of message originator
      HWND  theControl = HWND(lParam);                      // Handle to control sending notification


      switch ( notifyCode )
         {
            case 0:
            case 1:
               DoMenuCommand (itemID);                      // Handle menu command
               break;

            default:
               DoNotification (itemID, theControl, notifyCode);  // Handle control notification
               break;

         }  /* end switch ( notifyCode ) */

   }  /* end DoCommand */
```

Of course, the most important menu message of all is **WM_COMMAND**, which tells you that the user has chosen a command item from one of your menus. Much of the real work of a program gets done in response to this message. (Another message, **WM_SYSCOMMAND**, reports that an item has been chosen from the system menu, rather than one of your own. You'll usually just want to let the default window procedure handle this one; normally, the only reason for processing it yourself would be if you've added one or more commands of your own to the system menu.)

Listing 9–6. Handle menu command

```
VOID DoMenuCommand (UINT itemID)

   // Handle menu command.

   {
      switch ( itemID )
         {

            /* File menu */

               case New_Item:
                  DoNew ();                    // Handle New command
                  break;

               case Open_Item:
                  DoOpen ();                   // Handle Open... command
                  break;

               case Close_Item:
                  DoClose ();                  // Handle Close command
                  break;

               case Save_Item:
                  DoSave ();                   // Handle Save command
                  break;

               case SaveAs_Item:
                  DoSaveAs ();                 // Handle Save As... command
                  break;

               case Revert_Item:
                  DoRevert ();                 // Handle Revert to Saved... command
                  break;

               case Setup_Item:
                  DoSetup ();                  // Handle Page Setup... command
                  break;

               case Print_Item:
                  DoPrint ();                  // Handle Print... command
                  break;

               case Exit_Item:
                  DoExit ();                   // Handle Exit command
                  break;

            /* Edit menu */

               case Undo_Item:
                  DoUndo ();                   // Handle Undo command
                  break;
```

Listing 9–6. Handle menu command (*continued*)

```
            case Cut_Item:
               DoCut ();                   // Handle Cut command
               break;

            case Copy_Item:
               DoCopy ();                  // Handle Copy command
               break;

            case Paste_Item:
               DoPaste ();                 // Handle Paste command
               break;

            case Delete_Item:
               DoDelete ();                // Handle Delete command
               break;

            case SelectAll_Item:
               DoSelectAll ();             // Handle Select All command
               break;

         /* Format menu */

            case Format_Item:
               DoFormat ();                // Handle Text Format... command
               break;

            case Default_Item:
               DoDefaultFormat ();         // Handle Default Format command
               break;

            case Background_Item:
               DoBackground ();            // Handle Background Color... command
               break;

         /* Help menu */

            case Help_Item:
               DoHelp ();                  // Handle Help command
               break;

            case About_Item:
               DoAbout ();                 // Handle About WiniEdit... command
               break;

            default:
               MessageBeep (MB_OK);           // Error: control should never
               break;                      //    reach this point

         }  /* end switch ( itemID ) */

   }  /* end DoMenuCommand */
```

We've already encountered the **WM_COMMAND** message in Chapter 7, where it was used to report events affecting a control to the control's parent window. Recall that, when the message comes from a control, the high-order word of the **wParam** parameter contains a notification code identifying the nature of the event being reported. If the notification code is **0**, the message is from a menu item instead of a control. In this case, the low-order word of **wParam** gives the item identifier for the item that was chosen; **lParam** is irrelevant and is ignored. (A notification code of **1** denotes a menu item invoked from the keyboard rather than with the mouse; we'll be discussing this case in the next section.)

When WiniEdit's window procedure, **DoMessage**, receives a **WM_COMMAND** message, it calls the **DoCommand** routine shown in Listing 9–5. This routine simply unpacks the identifying information from the message parameters, examines the notification code to see if it's a menu command or a control notification, and calls the applicable WiniEdit routine, either **DoMenuCommand** or **DoNotification**. **DoMenuCommand** (Listing 9–6) is nothing but one big **switch** statement that relays control, in turn, to a set of even more specialized routines that handle each specific menu command. Notice that the identifiers used here to designate the various menu items (**New_Item**, **Open_Item**, and so on) are the same ones we saw earlier in Listing 9–1. These are constants defined by the program itself in the header file **WiniEdit Resources.h**. I associated them with the menu items when I built the menu onscreen with the Visual C++ interactive menu editor; the editor then incorporated them into the menu's resource description, as you can see by referring back to Listing 9–2.

As on the Macintosh, you can provide keyboard shortcuts to allow the user to invoke menu items from the keyboard rather than with the mouse. In fact, Windows allows two different kinds of keyboard shortcut: mnemonics and accelerators. Although they both serve the same basic purpose, the two have evolved separately, side by side, for different historical reasons. A good Windows program should support both.

Mnemonics

Right from the start, the Macintosh was designed with the mouse in mind. The mouse (or trackball or other equivalent pointing device) is an integral, indispensable part of the system: no Macintosh has ever been sold without one, or ever will be. In the IBM/DOS/Intel world, the mouse has always been regarded as an optional peripheral device, a gimmicky afterthought on what was essentially a keyboard- and character-based system. So Windows programs have always been expected to accommodate their mouseless users by providing a keyboard alternative for every mouse action.

In keeping with that principle, Windows defines a standard keyboard interface for choosing items from a menu:

- The Alt key toggles menu selection mode on or off. When first pressed, it enters menu mode and highlights the first item in the menu bar (normally a menu title); once in menu mode, pressing the Alt key escapes from it.
- The combination Alt-space displays the system menu.
- The left and right arrow keys cycle the highlight from item to item within the menu bar (including the system menu).
- The Enter key chooses the currently highlighted item, either activating a command or displaying a pop-up menu at the next level of the menu hierarchy.
- The up and down arrows cycle among the items within a pop-up menu.
- The Esc key backs up to the previous level of the menu hierarchy, or exits menu mode from the top level.

All of this standard behavior is implemented automatically by the default window procedure, so it takes no special effort on your part to support it. What you can do, however, is define *mnemonics* for your menus and menu items, allowing the user to choose them directly by typing a character from the keyboard while in menu mode, rather than laboriously cycling to them with the arrow keys. For example, if **`F`** is the mnemonic for the **`File`** menu and **`O`** is the mnemonic for the **`Open`**`...` item within that menu, then the user can invoke the **`Open`**`...` command by typing Alt to enter menu mode, then **`F`** for **`File`** and **`O`** for **`Open`**`....`

You define a mnemonic for a menu item by including an ampersand character (**`&`**) in the item's text, either in your menu template resource or in the text string that you pass as a parameter to **`AppendMenu`** or **`InsertMenu`**. The character immediately following the ampersand will become the item's mnemonic. Windows will automatically underline the character when displaying that item on the menu, as a cue to the user, and will invoke the item when the user types the character in menu mode. In Listing 9–2, for instance, you'll notice that WiniEdit's menu template defines a pop-up menu named **`&Edit`** containing items named **`&Undo`**, **`Cu&t`**, **`&Copy`**, **`&Paste`**, **`&Delete`**, and **`Select &All`**. Windows will display the items with the indicated characters underlined (**`Edit`**, **`Undo`**, **`Cut`**, **`Copy`**, **`Paste`**, **`Delete`**, **`Select All`**), and will interpret these characters as mnemonics when typed in menu mode.

> In defining your mnemonics, be careful not to confuse your user by using the same character to stand for two different items on the same menu. It's your responsibility to make sure your mnemonics are unique and unambiguous—as WiniEdit does, for instance, by using **`T`** as the mnemonic for **`Cut`**, to distinguish it from **`C`** for **`Copy`**. (It's OK to use the same mnemonic character on two different menus, or at different levels of the menu hierarchy, but not twice on the same menu.) If you do make this mistake, Windows will utter no complaint; it will just cycle from one of the conflicting items to the next each time the user types the ambiguous mnemonic character.

Accelerators

The second form of keyboard shortcut, *accelerators,* are more like the ones we're used to on the Macintosh. Instead of entering a special "menu mode" with the Alt key, an accelerator is simply a modeless keystroke that the user can type at any time, such as Ctrl-`S` for `Save` or Ctrl-`P` for `Print`. It's just a convenient alternative to using the mouse, not a substitute for the benefit of users who don't have a mouse in the first place.

Listing 9–7. WiniEdit accelerator table description

```
Accel_ID ACCELERATORS PRELOAD DISCARDABLE
BEGIN
    "A",            SelectAll_Item,        VIRTKEY, CONTROL
    "B",            Background_Item,       VIRTKEY, CONTROL
    "C",            Copy_Item,             VIRTKEY, CONTROL
    "D",            Default_Item,          VIRTKEY, CONTROL
    "F",            Format_Item,           VIRTKEY, CONTROL
    "N",            New_Item,              VIRTKEY, CONTROL
    "O",            Open_Item,             VIRTKEY, CONTROL
    "P",            Print_Item,            VIRTKEY, CONTROL
    "P",            Setup_Item,            VIRTKEY, CONTROL, ALT
    "Q",            Exit_Item,             VIRTKEY, CONTROL
    "R",            Revert_Item,           VIRTKEY, CONTROL
    "S",            Save_Item,             VIRTKEY, CONTROL
    "S",            SaveAs_Item,           VIRTKEY, CONTROL, ALT
    "V",            Paste_Item,            VIRTKEY, CONTROL
    VK_BACK,        Undo_Item,             VIRTKEY, ALT
    VK_DELETE,      Cut_Item,              VIRTKEY, SHIFT
    VK_F1,          Help_Item,             VIRTKEY
    VK_F2,          Cut_Item,              VIRTKEY
    VK_F3,          Copy_Item,             VIRTKEY
    VK_F4,          Paste_Item,            VIRTKEY
    VK_HELP,        Help_Item,             VIRTKEY, CONTROL
    VK_HELP,        Help_Item,             VIRTKEY, SHIFT, CONTROL
    VK_INSERT,      Copy_Item,             VIRTKEY, CONTROL
    VK_INSERT,      Paste_Item,            VIRTKEY, SHIFT
    "W",            Close_Item,            VIRTKEY, CONTROL
    "X",            Cut_Item,              VIRTKEY, CONTROL
    "Z",            Undo_Item,             VIRTKEY, CONTROL
END
```

You define your program's accelerators with a data structure called an *accelerator table.* Each entry in the table associates a keystroke with an item identifier, normally that of an item on one of your menus. You can either build the accelerator table from scratch with the Windows function `CreateAcceleratorTable` or (more commonly) define it as a resource and read it in at program initialization time with `LoadAccelerators`. Listing 9–7 shows the definition of WiniEdit's accelerator table, taken from the resource description file `WiniEdit.rc`. Like other resources that we've looked at, this one was built onscreen with the Visual C++ interactive resource editor and written out as a text-based resource description, then in turn compiled by the resource compiler into a resource of type `RT_ACCELERATOR` in the program's executable file.

Listing 9–8. Initialize accelerator table

```
VOID InitAccelerators (VOID)

   // Initialize accelerator table.

   {
      CHAR *resourceID;                                        // Resource ID in string form


      resourceID = MAKEINTRESOURCE(Accel_ID);                  // Convert resource ID
      AccelTable = LoadAccelerators(ThisInstance, resourceID);  // Load accelerator table

   }  /* end InitAccelerators */
```

Listing 9–9. WiniEdit main program loop

```
VOID MainLoop (VOID)

   // Execute one pass of main program loop.

   {
      MSG   theMessage;                               // Next message to process
      BOOL  translated;                               // Was message translated as a keyboard accelerator?


      ContinueFlag = GetMessage(&theMessage, NULL, 0, 0);  // Get next message

      translated = TranslateAccelerator (TheWindow, AccelTable, &theMessage);
                                                    // Check for keyboard accelerator

      if ( !translated )                            // Was the message an accelerator?
         {
            TranslateMessage (&theMessage);           // Convert virtual keys to characters
            DispatchMessage  (&theMessage);           // Send message to window procedure

         }  /* end if ( !translated ) */

   }  /* end MainLoop */
```

The WiniEdit routine **`InitAccelerators`** (Listing 9–8), part of WiniEdit's one-time initialization sequence, reads in the accelerator table resource and stores its handle in a global variable, **`AccelTable`**. Later, as the program's message loop (Listing 9–9) retrieves messages from the message queue, it passes each one to the Windows function **`TranslateAccelerator`**, along with the global accelerator table handle. If the message is a keystroke (message type **`WM_KEYDOWN`**), **`TranslateAccelerator`** looks it up in the accelerator table. If it finds an accelerator for that keystroke, it generates an equivalent **`WM_COMMAND`** message and dispatches it directly to the program's window procedure (unless it's the identifier of a disabled or grayed menu item). The **`WM_COMMAND`** message carries a notification code of **`1`** to identify it as an accelerator command, along with the item identifier associated with the given

keystroke in the accelerator table. (In Listing 9–7, for instance, if the keystroke were Ctrl-**A**, the corresponding item identifier would be **SelectAll_Item**.) **TranslateAccelerator** returns a boolean result to let the calling program know whether the incoming message was translated; if the result is **TRUE**, WiniEdit's message loop skips processing the raw keyboard message, knowing that an equivalent **WM_COMMAND** message has been dispatched in its place.

Although they resemble the keyboard shortcuts on the Macintosh, Windows accelerators are actually more flexible in a number of ways. For one thing, whereas Macintosh keyboard shortcuts are limited to combinations involving the Command key, Windows lets you define an accelerator for any keystroke you like. An accelerator can use any combination of the modifier keys Ctrl, Alt, and Shift, or it can use no modifiers at all. The WiniEdit accelerator table in Listing 9–7, for instance, defines the unmodified function keys **F1** to **F4** as accelerators for the **Help**, **Cut**, **Copy**, and **Paste** commands, respectively. In an application that doesn't require direct text entry from the keyboard, you could even use plain, unmodified letter keys as accelerators: **S** for **Save** instead of Ctrl-**S**, for example. Notice also that, unlike the Macintosh Toolbox, Windows lets you define more than one accelerator for the same operation. WiniEdit, for instance, provides three different accelerators for the **Cut** command: Shift-Delete (the Windows standard), Ctrl-**X** (the Macintosh standard, with the Ctrl key substituting for the Macintosh Command key), and **F2** (also common on the Macintosh).

Table 9–3 lists some accelerators that are considered standard in the Windows/DOS world. They aren't particularly logical or easy to remember, but Windows users expect them to work, so you should make sure your program supports them. The *system accelerators* listed in Table 9–4, on the other hand, are implemented automatically by the default window procedure; you don't need to anything special to support them, just make sure your own accelerators don't conflict with them. The Macintosh convention of using the first four keys on the bottom row of the keyboard for the standard editing commands (Ctrl-**Z** for **Undo**, Ctrl-**X** for **Cut**, Ctrl-**C** for **Copy**, Ctrl-**V** for **Paste**) also appears to be gaining acceptance in Windows, so you should probably support them, too. Many Macintosh programs also use the first four function keys (**F1–F4**) as shortcuts for these same four editing commands. Unfortunately, as shown in Table 9–4, **F1** is used almost universally in Windows as a help key, so it isn't available for the **Undo** command; but you can still use **F2**, **F3**, and **F4** for **Cut**, **Copy**, and **Paste**, as we've already seen that WiniEdit does.

Table 9–3. Standard Windows accelerators

Keystroke	Command equivalent
Alt-Backspace	**Undo**
Shift-Delete	**Cut**
Ctrl-Insert	**Copy**
Shift-Insert	**Paste**

Finally, notice that Windows accelerators, unlike their Macintosh counterparts, are not directly tied to the program's menu structure. On the Macintosh, by definition,

every Command-key shortcut stands for a menu item. You can’t define an operation

that's strictly keyboard-based and not also available by selecting from a menu with the mouse. (You could implement such an operation by hand, of course, but there's no way to do it directly through the Toolbox.) Windows doesn't have this restriction: the item identifiers in the accelerator table are simply integer constants that you define for yourself in your program's resource header file. Although it's common to use the same identifier for both an accelerator and a menu item, there's no law that this must be so. All the accelerator table says is that a certain keystroke typed by the user should be converted into a **`WM_COMMAND`** message with a certain value for its item identifier; the interpretation of that identifier is strictly up to your window procedure. So if you want to define an item identifier that corresponds to an accelerator keystroke only and has no corresponding menu item, you're free to do so. Whether this is a desirable user-interface feature can be debated either way; but the option is available if you want it.

Table 9–4. System accelerators

Keystroke	Meaning
Alt-Esc	Switch to next application
Shift-Alt-Esc	Switch to previous application
Alt-Tab	Cycle to next application
Shift-Alt-Tab	Cycle to previous application
Ctrl-Esc	Open task bar's Start menu
Alt-space	Open main window's system menu
Alt-hyphen	Open subwindow's system menu (Multiple Document Interface)
Alt-**F4**	Close main window
Ctrl-**F4**	Close subwindow (Multiple Document Interface)
F13	Copy snapshot of screen to clipboard
Alt-**F13**	Copy snapshot of active window to clipboard
F1	Help

Unlike the Macintosh Toolbox, Windows does *not* automatically incorporate keyboard accelerators into a menu when displaying it on the screen. If you want your accelerators to appear on the menu, you have to include them explicitly as part of each item's text. You can see an example of this in Listing 9–2, where, for instance, the text of the **`Cut`** menu item is defined as

Cu&t\tCtrl+X

As we saw in the last section, the ampersand (**`&`**) introduces the item's Alt-key mnemonic; the ampersand itself will not appear on the menu, but the character following it (in this case, **`t`**) will be underlined as the mnemonic. The item's accelerator, **`Ctrl+X`**, will be displayed next to it on the menu; the **`\t`** sets it off with a tab character to space it out to the right edge of the menu. The complete menu item will thus look like this:

Cut   Ctrl+X

Windows will automatically calculate the tab width so that all the menu's contents align neatly on the screen. Note that the accepted onscreen convention for combination keystrokes in Windows is to separate the modifier key from the printable character with a plus sign (**+**), not a hyphen as on the Macintosh (and as I've been using in this book): **`Ctrl+X`**, not **`Ctrl-X`**.

When you create an accelerator table in the normal way, by reading it in as a resource with **`LoadAccelerators`**, you don't need to worry about disposing of it: Windows will destroy it for you automatically when your program terminates. However, if you build it from scratch with **`CreateAcceleratorTable`**, you have to destroy it explicitly with **`DestroyAcceleratorTable`** before exiting from your program, or it will live on in memory after you're gone, like the Cheshire cat's grin.

Table 9–5 summarizes some common Windows functions relating to menus and accelerators. We've already discussed some of them in this chapter; you can learn about the rest in the *Win32 Programmer's Reference.*

Table 9–5. Common menu functions

Function	Mac counterpart	Purpose
CreateMenu	——————	Create top-level menu
CreatePopupMenu	**NewMenu**	Create pop-up menu
LoadMenu	**GetNewMBar**, **GetMenu**	Load menu from template resource
LoadMenuIndirect	——————	Create menu from template in memory
DestroyMenu	**DisposeMenu**	Destroy menu
AppendMenu	**AppendMenu**	Add item at end of menu
InsertMenu	**InsertMenu**, **InsMenuItem**	Add item anywhere in menu
DeleteMenu	**DeleteMenu**, **DelMenuItem**	Delete item and destroy associated menu, if any
RemoveMenu	**DeleteMenu**, **DelMenuItem**	Remove item without destroying associated menu, if any
SetMenu	**SetMenuBar**	Set window’s menu bar
GetMenu	**GetMenuBar**	Get handle to window’s menu bar
GetSubMenu	**GetMHandle**	Get handle to pop-up menu
GetSystemMenu	——————	Get handle to system menu
IsMenu	——————	Is handle a menu handle?
GetMenuItemCount	**CountMItems**	Get number of items in menu
GetMenuItemID	——————	Get item identifier by position
GetMenuString	**GetItem**	Get text of menu item
GetMenuState	——————	Get menu item attributes
ModifyMenu	**SetItem**, **EnableItem**, **DisableItem**, **CheckItem**, **SetItemIcon**	Change text, appearance, or attributes of menu item
EnableMenuItem	**EnableItem**, **DisableItem**	Enable, disable, or gray menu item
CheckMenuItem	**CheckItem**	Check menu item
SetMenuItemBitmaps	**SetItemMark**	Set bitmap representing check mark
GetMenuCheckMarkDimensions	——————	Get bitmap dimensions for check mark
DrawMenuBar	**DrawMenuBar**	Redraw menu bar
HiliteMenuItem	**HighlightMenu**	Highlight menu item on screen
TrackPopupMenu	**MenuSelect**	Track in menu
CreateAcceleratorTable	——————	Create accelerator table in memory

LoadAccelerators	—————	Load accelerator table from resource
DestroyAcceleratorTable	—————	Destroy accelerator table
CopyAcceleratorTable	—————	Copy accelerator table
TranslateAccelerator	**MenuKey**	Translate accelerator to menu command

Macintosh	Windows
• The Macintosh user can issue commands to a program by choosing menu items with the mouse.	• The Windows user can issue commands to a program by choosing menu items with the mouse.
• A Macintosh menu item can either invoke a command or pop up a submenu.	• A Windows menu item can either invoke a command or pop up a submenu.
• Macintosh menu items can be disabled, preventing them from responding to the mouse.	• Windows menu items can be disabled or grayed, preventing them from responding to the mouse.
• Macintosh menu items can be marked with a check mark or other symbol to show that they are currently in effect.	• Windows menu items can be marked with a check mark or other symbol to show that they are currently in effect.
• Macintosh menus and menu bars can be built from scratch or read in as resources.	• Windows menus and menu bars can be built from scratch or read in as resources.
• A Macintosh program can define keyboard shortcuts to allow the user to invoke menu items from the keyboard.	• A Windows program can define keyboard mnemonics and accelerators to allow the user to invoke menu items from the keyboard.

...Only Different

Macintosh	Windows
• The Macintosh has one global menu bar at the top of the screen.	• A Windows menu bar belongs to a single window.
• The Macintosh menu bar is a separate entity from the menus it contains.	• A Windows menu bar is simply a menu that's displayed horizontally instead of vertically.
• Every item in the Macintosh menu bar is a menu title.	• The Windows menu bar can contain command items as well as menu titles.
• Macintosh menus disappear when the user releases the mouse button.	• Windows menus remain visible until the user clicks the mouse button a second time.
• The Macintosh has an Apple menu containing desk accessories and other frequently used items.	• Windows has a system menu containing standard commands for manipulating the window on the screen.

- Macintosh menu items are identified by their sequential position within the menu.
- Windows menu items are identified by an arbitrary ID number assigned by the program.

- When the user clicks the mouse in the menu bar, a Macintosh program must explicitly call a Toolbox tracking routine, **`MenuSelect`**, to track the mouse.
- When the user clicks the mouse in the menu bar, a Windows program can leave the job of tracking the mouse to the default window procedure.

- A Macintosh program obtains the menu item chosen by the user as a direct function result returned by **`MenuSelect`**.
- A Windows program obtains the menu item chosen by via a **`WM_COMMAND`** message sent through the normal message mechanism.

- A Macintosh program must either enable and disable its menu items incrementally as conditions change, or explicitly update the state of its menus before calling **`MenuSelect`**.
- A Windows program receives a message, **`WM_INITMENUPOPUP`**, telling it when to update the state of a menu.

- Macintosh keyboard shortcuts are defined as part of the menu structure itself.
- Windows accelerators are defined in a separate accelerator table that can be read in as an independent resource.

- Macintosh keyboard shortcuts are always invoked by key combinations involving the Command key.
- Windows accelerators can be invoked by any key combination at all.

- A Macintosh keyboard shortcut must invoke an existing menu item.
- A Windows accelerator can invoke any operation at all, whether it exists as an independent menu item or not.

- A Macintosh menu item can have at most one keyboard shortcut.
- A Windows menu item can have any number of accelerators.