
C H A P T E R

10

Working with Menus

*Inside the Menuing
System*

*Using the Menu API
Functions with VB*

Examples

Function Reference



MOST VISUAL BASIC APPLICATIONS USE MENUS TO ALLOW THE USER TO execute program commands. This chapter will show you how to use the Windows API functions to add to the menu capabilities provided by Visual Basic. You will learn how to create custom checkmarks for checked menus, and how to use any bitmap as a menu entry in place of a string. You will also learn how to customize floating pop-up menus that can appear anywhere on the screen. The MenuLook sample program included in this chapter demonstrates these features and shows how you can use the Windows API functions to analyze the structure of an existing menu. Finally, this chapter will demonstrate some more advanced techniques for implementing customized control menus and context menus with the aid of subclassing techniques.

Inside the Menuing System

Before reviewing the Windows API functions that deal with menus, it is important for you to understand a bit about how menus work and how Visual Basic uses menus. The Windows API functions provide some powerful capabilities, but the Visual Basic environment imposes some strict requirements on their use in order to maintain compatibility.

How Windows Menus Work

Let's first examine how the Windows menu system works outside of Visual Basic. Then you'll see how Visual Basic interacts with menus.

A menu is one of the few objects appearing on the screen that is not a window. This means that there is no window handle to a menu, nor does a menu have a window function. The appearance, visibility, and position of menus are handled entirely by the Windows environment.

There are two types of menus: top level menus and pop-up menus. A top level menu appears as a horizontal bar and may be assigned to a window. Once assigned, it will appear as the menu bar for the window. Pop-up menus appear as needed and disappear as soon as a selection is made, the Escape key is pressed, or the mouse is clicked outside of the menu. Pop-up menus may appear as dropping down from the top level menu bar or another pop-up menu, or may appear anywhere on the screen under program control. They are often used as context menus—a pop-up menu that appears when you click on an object with the right mouse button.

Each entry in a menu has attributes as shown in Table 10.1.

In Windows, a menu is built by first creating a top level menu and the associated pop-up menus. In most cases, each of the top level menu entries has a pop-up menu attached. Each entry has attributes set as needed. If an entry does not have a pop-up menu attached, it is assigned a menu ID attribute. Once the menu is built, it is assigned to a window and appears as a menu bar. Menus may also be loaded as resources from an executable module.



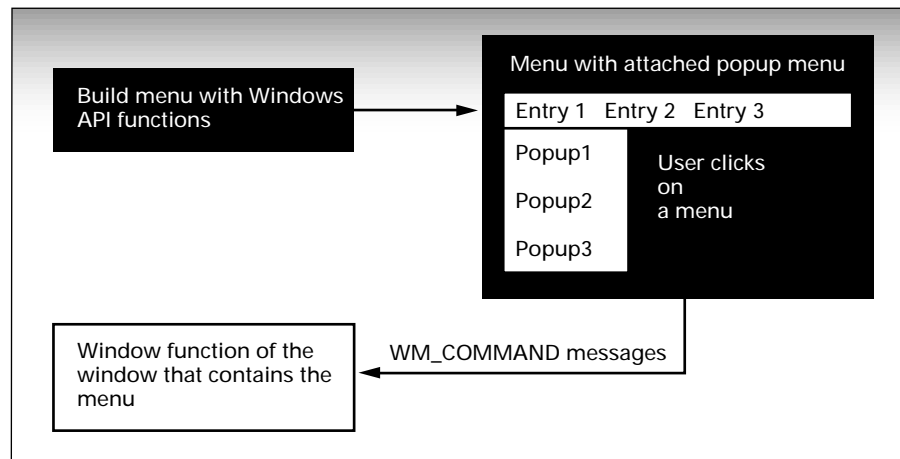
Table 10.1 Attributes of a Menu Entry

Attribute	Description
Bitmap	Bitmap to display instead of a string for a menu entry.
Checked	An entry can be checked or unchecked. A space to the left of the entry name (or string) displays a symbol to indicate if the entry is checked or unchecked. The default is nothing when unchecked and a checkmark when checked, but any bitmap can be defined for either state.
Unchecked	
Checkmark symbol	Bitmaps to use for the checked and unchecked state for a menu entry. The default is no mark for the unchecked state, and a checkmark for the checked state.
Enabled Disabled Grayed	If a menu entry is enabled, the user can click on that entry. When disabled, clicks on the entry are ignored. When grayed, the entry is disabled and appears in a gray color: it is not available to the user at that time.
Default (bold)	If a menu entry is the default entry, it appears in bold. By convention, when used with context menus, this is the same action that takes place for an object that is double clicked. Not supported on NT3.51.
Highlight	Top level menus only—the entry appears highlighted (inverted).
Menu ID (command)	Every entry other than separators and pop-up menus has a menu ID. This is a 16-bit integer that is sent to the window function of the menu's window when that entry is selected. Menu IDs need not be unique (that is, more than one entry may share the same menu ID).
MenuBreak MenuBarBreak	Specifies that an entry is the start of a new column (for pop-up menus) or line (for top level menus). With MenuBarBreak, a vertical separator line appears between the two columns.
OwnerDraw	Allows total customization of the appearance of a menu. Not supported by Visual Basic.
Pop-up	If an entry has a pop-up menu assigned, selecting the entry will cause that pop-up menu to appear. Pop-up entries generally have a string attribute assigned as well.
Position	The position of the entry in the menu. Positions are numbered from zero, with entry zero being the left entry (for top level menus) or top entry (for pop-up menus).
Separator	A special menu entry that appears as a separator line between entries. It is always disabled.
String (name)	The text string displayed for the entry. Also referred to as the name of the entry.

Consider what happens when the user clicks in a menu. Clicks inside of a disabled entry (such as a separator or disabled command) are ignored. Clicks inside of an entry that has a pop-up menu attached cause the pop-up menu to be displayed, from which the user may then select an entry.

When a user clicks on an enabled entry that does not have a pop-up menu attached, Windows sends a WM_COMMAND message to the window that contains the menu. This message contains the menu ID of the menu entry. Figure 10.1 illustrates the operation of menus under Windows.

Figure 10.1
Menu operation
under Windows



Each of the three entries in the pop-up menu has a menu ID that is sent as one of the parameters with the WM_COMMAND message. This allows the window function for the window owning the menu to determine which menu entry has been selected. The program can then take whatever action is appropriate for that menu selection.

Table 10.2 lists the API functions that deal with menus.

Table 10.2 Menu API Functions

Function	Description
AppendMenu	Adds an entry to a menu.
ChangeMenu	Obsolete: Use AppendMenu, InsertMenu, ModifyMenu, and RemoveMenu, or the new MenuItemInfo function.
CheckMenuItem	Checks or unchecks a menu entry.



Table 10.2 Menu API Functions (Continued)

Function	Description
CheckMenuRadioItem	Checks one of a group of menu entries, unchecking the others in the group. N/A on NT 3.51.
CreateMenu	Creates an empty top level menu.
CreatePopupMenu	Creates an empty pop-up menu.
DeleteMenu	Deletes a menu entry.
DestroyMenu	Destroys a menu.
DrawMenuBar	Updates (redraws) a menu.
EnableMenuItem	Enables, disables, or grays a menu entry.
GetMenu	Retrieves a handle to the menu for a window.
GetMenuCheckMarkDimensions	Determines the size of a menu checkmark symbol.
GetMenuContextHelpId	Retrieves a help context associated with a menu. N/A on NT 3.51.
GetMenuDefaultItem	Retrieves the default entry for a menu. N/A for NT 3.51.
GetMenuItemCount	Determines the number of entries in a menu.
GetMenuItemID	Determines the menu ID of a menu entry.
GetMenuItemInfo	New general purpose function for retrieving menu information. N/A on NT 3.51.
GetMenuItemRect	Retrieves the size and location of an entry in a menu.
GetMenuState	Retrieves information about the attributes of a menu entry.
GetMenuString	Retrieves the string (name) of a menu entry.
GetSubMenu	Retrieves a handle to the pop-up menu attached to a specified menu entry.
GetSystemMenu	Retrieves a handle to the system pop-up menu (referred to as the control menu in VB). This is the menu that appears when the ControlBox property for a form is set to TRUE.
HiliteMenuItem	Sets the highlight attribute for a top level menu entry.
InsertMenu	Inserts a menu entry into a menu.

Table 10.2 Menu API Functions (Continued)

Function	Description
InsertMenuItem	General purpose function for inserting menu entries. N/A on NT 3.51.
IsMenu	Determines if a handle is not a menu handle.
LoadMenu	Loads a menu resource.
LoadMenuIndirect	Creates a menu from a data structure.
MenuItemFromPoint	Tests whether there is a menu entry at the screen location specified.
ModifyMenu	Changes the attributes of a menu entry.
RemoveMenu	Removes a menu entry. If the entry has a pop-up menu attached, the pop-up is not destroyed (unlike the case with DeleteMenu).
SetMenu	Sets the menu for a window.
SetMenuContextHelpId	Sets a help context associated with a menu. N/A on NT 3.51.
SetMenuDefaultItem	Sets the default entry for a menu. N/A on NT 3.51.
SetMenuItemBitmaps	Sets the symbols to use to indicate the checked or unchecked state of a menu entry.
SetMenuItemInfo	New general purpose function for modifying menu entries. N/A on NT 3.51.
TrackPopupMenu	Brings up a pop-up menu anywhere on the screen.
TrackPopupMenuEx	Supports extended pop-up menu features. N/A on NT 3.51.

Standard versus Extended Menus

Windows 95 and NT 4.0 and later implement a number of changes to the traditional menu system while continuing to support the standard Win32 menu functions. These changes are summarized below:

- **Context help:** Allows each menu to have a help context identifier associated with it. This help context is not actually used by Windows, but can be easily retrieved by the application to implement online help.
- **Context Menu support:** A context menu is a pop-up menu that appears when an object such as a control is clicked. Any control can implement a



context menu, and context menu support is strongly recommended for controls or other user interface objects. Windows does provide some support in the operating system for context menus in the form of the `WM_CONTEXTMENU` message (see Chapter 16), but it is the responsibility of each control or the application itself to implement this capability. The `TrackPopupMenuEx` function extends the original `TrackPopupMenu` command by supporting pop-ups that use the right mouse button as well as the left. It also provides improved control over where the pop-up menu will appear.

- **Default Entry:** Windows allows one entry in a menu to be a default entry which appears in bold. This entry should represent the default command that is executed when the object associated with a pop-up context menu is double clicked.
- **New API functions.** Along with individual menu API commands, Windows 95 and NT 4.0 include the `GetMenuItemInfo`, `InsertMenuItem`, and `SetMenuItemInfo` functions that allow you to manipulate menu entries using the `MENITEMINFO` structures. Microsoft recommends using these functions instead of the standard menu function. At this time it is probably still too soon for most programmers to write off NT 3.51 and earlier. This book, therefore, advocates use of the standard menu functions, along with some of the newer functions that do not use the `MENITEMINFO` structure. Not only does this reduce the amount of code needed to support all three platforms, but it turns out that the standard functions are considerably easier to use under Visual Basic.

How Visual Basic Menus Work

The Visual Basic environment includes a sophisticated menu design window that is used to create menus for Visual Basic programs. When you design a menu using the Visual Basic menu design window, you are not actually creating a Windows menu—at least not directly. The VB menu designer actually creates an internal VB menu object. VB uses this object to create the actual menu using Windows API functions. VB sets some of the attributes of the menu according to the properties you specify. Other attributes are assigned based on an internal scheme. Table 10.3 shows how VB menu control properties correspond to the Windows menu attributes.

Some of the menu attributes have no equivalent in a VB property. Table 10.4 indicates the degree of VB compatibility of these attributes.

VB menu controls manipulate the actual menu in many ways. It is important to be aware of this situation when changing menus directly. The most important fact to keep in mind when modifying menus directly through API functions is that changes to the menus do not affect the VB menu controls.

Table 10.3 VB Menu Properties and Corresponding Attributes

Visual Basic Property	Windows Menu Attribute
Caption	String (name).
CtlName	No equivalent.
Index	No equivalent.
Tag	No equivalent.
Checked	Checked.
Enabled/Disabled	Enabled/Grayed. There is no VB equivalent to the Disabled but Ungrayed state.
Parent	No direct equivalent. This is the handle of the window that contains the menu.
Visible	Not an attribute. Visual Basic deletes any entries or menus that are not visible.
HelpContextID	HelpContext. However, the help context property is not compatible with the help context attribute.

Table 10.4 Menu Attributes without Corresponding VB Properties

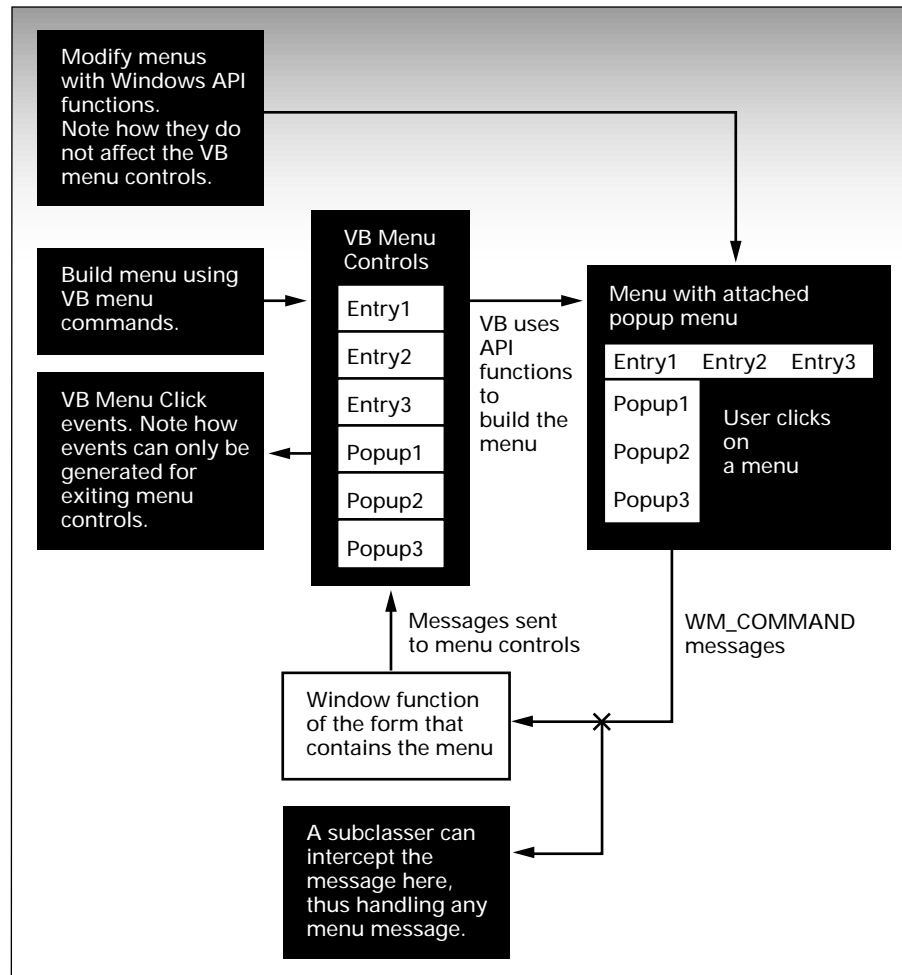
Windows Menu Attribute	Visual Basic Equivalent
Separator	Caption property = "-".
Bitmap	No equivalent.
Checkmark Symbol	No equivalent.
Menu ID	Chosen internally by VB. User cannot set.
MenuBreak, MenuBarBreak	No equivalent.
Pop-up	Level in the menu design window.
Position	Position in menu design window.

Figure 10.2 shows the flow of control in the Visual Basic menu system. Note how the VB menu controls set the structure of the menu, but that the menu has no corresponding arrow to set the contents of the VB menu controls. The Windows API functions bypass the VB menu controls and operate directly on the menu.



The impact of the structure of the menu system on the programmer depends on the attributes and properties in question.

Figure 10.2
Operation of menus
under Visual Basic



The Caption, Checked, and Enabled Properties

If you change the string attribute of a menu entry, the menu will display the new string but the VB Caption property for that menu entry will *not* reflect the new string. This applies to the Enabled and Checked properties as well. Note that if you use an API function to change the menu entry string or the enabled or checked state, the menu will work as you specify. It will display the new string



and it will be checked or enabled as you specify, so in this sense these API changes are compatible with VB. However, if you choose to use the API functions to change these attributes you must avoid using the equivalent VB property to read the current state of the menu (you may use it to set the state of the menu, as setting the property will immediately update the menu to the new state).

In general, it is recommended that you use the VB property to set these three attributes.

The Visible Property—Adding and Deleting Menu Entries

It is important to be aware that hiding a menu entry does not “hide” the entry—it actually deletes it from the menu. This affects the position attribute of all menu entries below the one that was hidden. By the same token, VB assumes that menu entries are where it expects them to be. If you start adding and deleting entries directly, you can cause Visual Basic to lose track of where menu entries are located, in which case changes to properties of a menu entry will actually affect the wrong menu entry.

As a result, you should not use API functions to add or delete menu entries on any menu in the VB menu structure. You may use these API functions if you are creating a custom menu that is compatible with the VB menu structure, as described in the next section.

The correct way to add or delete entries is to use a Visual Basic menu control array, then modify each menu entry as needed using properties or API functions.

The Menu ID Attribute and Ensuring Compatibility of Menu Entries to VB
Figure 10.2 shows how the WM_COMMAND messages from the menu are sent to the VB menu controls for processing. When a VB menu control receives a WM_COMMAND message that corresponds to its own menu ID, it generates a Click event.

Since a Visual Basic program cannot intercept windows messages directly, this imposes the requirement that every menu entry have a VB control. Note that VB controls only look at the menu ID for the WM_COMMAND message, so the reverse is not true—each VB control may have more than one menu entry.

From a practical point of view, this means that you may create pop-up menus at will, assign them to entries, or use them as tracked pop-up menus, and reassign menu IDs as you wish as long as you make certain that the menu ID for each entry has a corresponding VB menu control. When a menu structure created or modified with API functions meets this condition, it is referred to in this book as a *compatible* menu structure.

If, however, you are using a subclasser, you can define any menu ID that you wish, since the subclasser can intercept the WM_COMMAND message coming in to the window. If you do this, you must be extremely careful that any menu commands due to your custom menu are blocked before they



reach the destination window. Some Visual Basic controls are notoriously intolerant of unexpected `WM_COMMAND` or `WM_MENUSELECT` messages. You must also be extremely careful not to use a menu ID in this case that duplicates one that is already in use by a form or control. These issues are discussed further in the description of the SysMenu project later in this chapter. Subclassing is described further in Chapter 16.

Using the Menu API Functions with VB

This section describes a number of practical applications for the Windows menu API functions and how to safely use them with Visual Basic. Most of these examples are further illustrated in the MenuLook example program included in this chapter.

Creating Custom Checkmark Symbols

The space to the left of the menu entry text, or caption, is reserved for checkmarks. Normally, the space is empty when unchecked and displays a checkmark when the entry has been checked. Visual Basic is fully compatible with custom checkmarks; thus you could substitute another symbol such as a “+” or “-” for the standard “÷” symbol as needed by your application.

A custom checkmark must be the same size as the original checkmark. These dimensions may be obtained using the `GetMenuCheckMarkDimensions` API function. You may specify bitmaps for both the unchecked and checked state, and the bitmaps may be used on more than one menu entry. Note, however, that it is the programmer’s responsibility to destroy the bitmap when it is no longer needed or before the program terminates.

Using Bitmaps to Customize Menus

You may replace the name (caption) of any menu entry with the bitmap of your choice. The menu entry will automatically be sized to hold the bitmap and will display color bitmaps in full color depending on the characteristics of the display. This is useful for any case where you would like to use a picture in a menu entry instead of a text string—for example, you could allow your user to choose from among hatched brushes by displaying the available hatching patterns in the menu.

One subtle use of this capability is to employ different fonts or text styles in menus. All you need to do is draw text onto a bitmap using the desired font or style. This bitmap may then be used as a menu entry.

Use the `ModifyMenu` API function to change a menu entry into a bitmap. Note that this will not affect the `Caption` property for the menu entry, which will continue to return the previous caption. Setting the `Caption` property will, however, remove the bitmap and replace it with the specified string.



It is important that menu entry bitmaps be preserved during the existence of the menu. VB modifies the bitmap accessed by the Image property of controls, so it is not appropriate to use the bitmap returned by the Image property of a control as a menu entry bitmap. One may, however, use the Picture property as long as the bitmap is left unchanged while it is in use. The MenuLook application solves this problem by drawing into a picture control that has the AutoRedraw property set to TRUE, and then making a copy of the image bitmap.

Bitmaps may be shared by menu entries, and should be deleted when the application exits in order to release the associated Windows resources. Chapter 9 discusses bitmaps in greater depth.

It is also possible to implement owner draw menus with the aid of a subclasser. An owner draw menu allows you to completely customize the size and contents of a menu entry using any Windows graphics command. You can see a demonstration of owner draw menus by running the SpyDemo demonstration application that is included on the CD-ROM.

Tracked Pop-up Menus

Visual Basic provides direct support for floating pop-up menus to appear anywhere on the screen using the PopupMenu command. The TrackPopupMenu API function can also be used to create pop-up menus in cases where customization is required. The only requirement is that this pop-up menu contain entries that have menu IDs that correspond to Visual Basic menu controls. One way to do this is to create a pop-up menu using the VB menu design tools, and simply hide it by setting the caption property of the top level menu for that pop-up to "" (note that this is *not* the same as setting the Visible property to zero, an action that destroys the menu). You can then use the GetSubMenu command to obtain a handle to the pop-up menu to use with TrackPopupMenu. Alternatively, you may create a new pop-up menu that is compatible with the VB menu structure.

The MenuLook application shows how to hide a top level entry and use its pop-up menu for a tracked pop-up. Note that even when a top level caption is set to the NULL string, it takes up space on the menu bar, so it is advisable to disable the menu entry as well.

Tracked pop-up menus are often used under Windows 95 and NT 4.0 to implement context menus—a small pop-up menu that appears when you right click on a control or form. The documentation for Visual Basic describes how to implement pop-up menus by using the PopupMenu command in response to the MouseDown or MouseUp event. This is fine in most cases; however, it can lead to problems with those controls that by default provide their own context menus. For example, the text box brings up its own context menu, so you would actually see two context menus if you took this approach—the standard text box context menu followed by your context menu.



The SysMenu sample program shows how you can use subclassing to detect the WM_CONTEXTMENU message going to a text box to override the default context menu and completely replace it with your own.

Creating a Pool of VB Menu Controls

If you expect to do a lot of menu customization or use many tracked pop-up menus, you will probably need a pool of menu controls to use in order to obtain menu IDs that will generate click events. The easiest way to do this is as follows:

1. Set the last entry on your top level menu to have no caption and set the Enabled property to zero.
2. Create under this entry a pop-up menu with a single entry by creating an entry indented one level in the menu design window. Make this entry a control array by setting the value of the Index property to 0.

Now, any time you need a VB menu control to use in another menu entry, simply create a new entry in the control array using the Visual Basic Load command. Because the top level entry for the control array is effectively hidden, you need not worry about the user seeing all of the menu entries you are creating in the control array. Meanwhile, new entries you create may use those menu IDs safely and will trigger Click events in the control array when selected.

Menus, System Menus, and Subclassing

Subclassing is a technique which allows you to intercept Windows messages going to a form. This technique can be used to detect the WM_COMMAND Windows message directly, eliminating the need to ensure compatibility with a Visual Basic menu structure when using menu API functions. It also allows you to intercept the WM_SYSCOMMAND message which makes it practical to customize an application's system menu. Refer to the Message Handling section in Chapter 16 for more information on subclassing and the tools required to use this powerful technique.

Obtaining Information about the VB Menu Structure

It is remarkably easy to obtain information about the existing VB menu structure. The GetMenu API function retrieves a menu handle to the top level menu for a form. The GetSubMenu function can then be used to obtain handles to the pop-up menus that it contains.

Additional API functions may be used to retrieve values of the various menu properties. The MenuLook example program shows how you may obtain a complete description of the menu structure.

Examples

This chapter includes two examples: MenuLook, which demonstrates how to customize menus and how to analyze a menu structure, and SysMenu, which demonstrates how to modify the system menu and use context menus.

MenuLook—A Menu Structure Viewer

MenuLook is a program that allows you to view the structure of the Visual Basic menu and shows how a menu may be modified. It also demonstrates many of the techniques for modifying menus described in this chapter.

This sample was ported from the 16-bit MenuLook program included in the original Visual Basic Programmer's Guide to the Windows API (16-bit edition), and has been implemented to run on both 16-bit VB4 and 32-bit VB platforms. This listing includes comments relating to some of the porting issues that came up. In addition, this program demonstrates some of the subtle issues relating to using strings inside of structures. These issues are discussed briefly here, but covered in detail in Chapter 15.

Using MenuLook

The MenuLook screen shown in Figure 10.3 demonstrates a variety of menu control techniques. The List1 list box is loaded with an analysis of the existing menu structure when the Analyze command button is selected.

The menu analysis shows the handle of each menu and a description of each menu entry. When an entry is a sub-menu, its handle is displayed along with its name. Handles are always displayed in hexadecimal; all other numbers are in decimal. Each sub-menu is analyzed in turn, with each menu entry shown along with its menu ID.

Figure 10.3
The MenuLook
program in action



Listing 10.1 Header for file MENULOOK.FRM

```
VERSION 5.00
Begin VB.Form Menulook
    Appearance      = fl 'Flat
    BackColor       = &H8f8f8f8f8f5&
    Caption         = "Menulook demo program"
    ClientHeight    = 412
    ClientLeft      = 1195
    ClientTop       = 177
    ClientWidth     = 7365
    BeginProperty Font
        name         = "MS Sans Serif"
        charset      = fl
        weight       = 7
        size        = 8.25
        underline    = fl 'False
        italic       = fl 'False
        strikethrough = fl 'False
    EndProperty
    ForeColor       = &H8f8f8f8f8f8&
    Height          = 471
    Left           = 1135
    LinkMode        = 1 'Source
    LinkTopic       = "Form1"
    ScaleHeight     = 412
    ScaleWidth      = 7365
    Top            = 114
    Width          = 7485
    Begin VB.ListBox List1
        Appearance    = fl 'Flat
        Height        = 1785
        Left         = 24
        TabIndex     = fl
        Top          = 216
        Width        = 6855
    End
    Begin VB.PictureBox Picture2
        Appearance    = fl 'Flat
        AutoRedraw    = -1 'True
        BackColor     = &H8f8f8f8f8f5&
        ForeColor     = &H8f8f8f8f8f8&
        Height        = 375
        Left         = 444
        ScaleHeight   = 345
        ScaleWidth    = 1315
        TabIndex     = 6
        Top          = 168
        Width        = 1335
    End
End
```



File **MENULOOK.FRX** is the binary file for this form. It contains the definition of the **Picture** property for control **Picture1**. You may remove the line

```
Picture = MENULOOK.FRX:fffffl
```

in which case you should load a small bitmap into the control's **Picture** property in Visual Basic design mode. The sample program uses the bitmap **ARGYLE.BMP**, which is provided with Windows.

```
Begin VB.PictureBox Picture1
    Appearance      = fl 'Flat
    BackColor       = &H8ffffff5&
    ForeColor       = &H8ffffff8&
    Height          = 495
    Left            = 372fl
    Picture         = "MENULOOK.frx":fffffl
    ScaleHeight     = 465
    ScaleWidth      = 465
    TabIndex        = 5
    Top             = 156fl
    Width           = 495
End
Begin VB.CommandButton CmdAnalyze
    Appearance      = fl 'Flat
    BackColor       = &H8ffffff5&
    Caption         = "Analyze"
    Height          = 495
    Left            = 6ffffl
    TabIndex        = 1
    Top             = 114fl
    Width           = 1215
End
Begin VB.CommandButton CmdTrack
    Appearance      = fl 'Flat
    BackColor       = &H8ffffff5&
    Caption         = "TrackPop-up"
    Height          = 495
    Left            = 6ffffl
    TabIndex        = 2
    Top             = 6fffl
    Width           = 1215
End
Begin VB.CommandButton CmdEntry2Chk
    Appearance      = fl 'Flat
    BackColor       = &H8ffffff5&
    Caption         = "Entry2-Check"
    Height          = 495
    Left            = 456fl
    TabIndex        = 8
    Top             = 6fffl
    Width           = 1335
```

```
End
Begin VB.CommandButton CmdAddBitmap
    Appearance      = fl 'Flat
    BackColor       = &H8f8f8f8f8f5&
    Caption         = "Add Bitmap"
    Height          = 495
    Left            = 6f8f8f
    TabIndex        = 4
    Top             = 6f8f
    Width           = 1215
End
Begin VB.CommandButton CmdAddRandom
    Appearance      = fl 'Flat
    BackColor       = &H8f8f8f8f8f5&
    Caption         = "AddRandom"
    Height          = 495
    Left            = 456f8f
    TabIndex        = 7
    Top             = 6f8f
    Width           = 1335
End
Begin VB.Label Label1
    Appearance      = fl 'Flat
    BackColor       = &H8f8f8f8f8f5&
    BorderStyle     = 1 'Fixed Single
    ForeColor       = &H8f8f8f8f8f8&
    Height          = 255
    Left            = 24f8f
    TabIndex        = 3
    Top             = 168f8f
    Width           = 3255
End
Begin VB.Menu MenuTop
    Caption         = "Menu1"
    HelpContextID   = 5f8f8f
    Begin VB.Menu MenuEntry1
        Caption      = "Entry1"
    End
    Begin VB.Menu MenuEntry2
        Caption      = "Entry2"
    End
    Begin VB.Menu MenuEntry3
        Caption      = "-"
    End
    Begin VB.Menu MenuArray1
        Caption      = "Array1"
        Index        = fl
    End
    Begin VB.Menu MenuArray1B
        Caption      = "Array1B"
        Index        = 1
    End
End
```



```

Begin VB.Menu MenuSub-menu1
    Caption      = "Sub-menu1"
    Begin VB.Menu MenuSub1Entry1
        Caption    = "Sub1Entry1"
    End
    Begin VB.Menu MenuSub1Entry2
        Caption    = "Sub1Entry2"
    End
End
End
Begin VB.Menu MenuFloat
    Caption      = "Floating"
    Begin VB.Menu MenuFloat1
        Caption    = "Float1"
        Index      = fl
    End
    Begin VB.Menu MenuFloat1
        Caption    = "Float2"
        Index      = 1
    End
End
Begin VB.Menu MenuRandomTop
    Caption      = "Random"
    Begin VB.Menu MenuArt
        Caption    = "Art1"
        Index      = fl
    End
End
End
End

```

MenuLook Listings

Module MENULOOK.BAS, shown in Listing 10.2, contains the constant, variable, and function declarations used by the program. Listing 10.3 shows the code for MENULOOK.FRM.

Listing 10.2 MENULOOK.BAS

```

Attribute VB_Name = "MENULOOK1"
Option Explicit

' MenuLook sample program
' Copyright (c) 1992-1997, Desaware

'-----

'      Public Constants

```

This file includes the 16- and 32-bit declarations for the following types from the file API32.TXT:

RECT, POINTAPI and BITMAP

It also includes the 32-bit declarations for OSVERSIONINFO and MENUITEMINFO. Note that MENUITEMINFO has been modified from the original WIN32API.TXT file that comes with Visual Basic for reasons that will become clear.

This file also includes constant declarations from the file API32.TXT that begin with the following prefixes: MIIM_, MF_, MFS_, MFT_, TPM_, VER_, and the standard trinary raster operations (such as SRCCOPY, SRCPAINT, and so forth).

```
Public Const WM_USER = &H4f1f1f

Public Const LB_RESETCONTENT = (WM_USER + 5)

' Bitmap to use for checkmark on entry one menu

' Port: Convert to longs
Public NewCheck&

Public BMHandles&(32)

Public FloatBitmap&

#If Win32 Then
Declare Function GetLastError Lib "kernel32" () As Long
Declare Function lstrlen Lib "kernel32" Alias "lstrlenA" (ByVal lpstring As _
String) As Long
Declare Function GetMenuItemInfo Lib "user32" Alias "GetMenuItemInfoA" (ByVal _
hMenu As Long, ByVal un As Long, ByVal b As Long, lpMenuItemInfo As _
MENUITEMINFO) As Long
Declare Function GetVersionEx Lib "kernel32" Alias "GetVersionExA" _
(lpVersionInformation As OSVERSIONINFO) As Long
Declare Function CreateBitmapIndirect Lib "gdi32" (lpBitmap As BITMAP) As Long
Declare Function LoadMenu Lib "user32" Alias "LoadMenuA" (ByVal hInstance As _
Long, ByVal lpstring As String) As Long
Declare Function LoadMenuIndirect Lib "user32" Alias "LoadMenuIndirectA" (ByVal _
lpMenuTemplate As Long) As Long
Declare Function GetMenu Lib "user32" (ByVal hwnd As Long) As Long
Declare Function SetMenu Lib "user32" (ByVal hwnd As Long, ByVal hMenu As Long) _
As Long
Declare Function HiliteMenuItem Lib "user32" (ByVal hwnd As Long, ByVal hMenu _
```



```

As Long, ByVal wIDHiliteItem As Long, ByVal wHilite As Long) As Long
Declare Function GetMenuString Lib "user32" Alias "GetMenuStringA" (ByVal hMenu _
As Long, ByVal wIDItem As Long, ByVal lpstring As String, ByVal nMaxCount _
As Long, ByVal wFlag As Long) As Long
Declare Function GetMenuState Lib "user32" (ByVal hMenu As Long, ByVal wID As _
Long, ByVal wFlags As Long) As Long
Declare Function DrawMenuBar Lib "user32" (ByVal hwnd As Long) As Long
Declare Function GetSystemMenu Lib "user32" (ByVal hwnd As Long, ByVal bRevert _
As Long) As Long
Declare Function CreateMenu Lib "user32" () As Long
Declare Function CreatePop-upMenu Lib "user32" () As Long
Declare Function DestroyMenu Lib "user32" (ByVal hMenu As Long) As Long
Declare Function CheckMenuItem Lib "user32" (ByVal hMenu As Long, ByVal _
wIDCheckItem As Long, ByVal wCheck As Long) As Long
Declare Function EnableMenuItem Lib "user32" (ByVal hMenu As Long, ByVal _
wIDEnableItem As Long, ByVal wEnable As Long) As Long
Declare Function GetSubMenu Lib "user32" (ByVal hMenu As Long, ByVal nPos As _
Long) As Long
Declare Function GetMenuItemID Lib "user32" (ByVal hMenu As Long, ByVal nPos As _
Long) As Long
Declare Function GetMenuItemCount Lib "user32" (ByVal hMenu As Long) As Long

Declare Function InsertMenu Lib "user32" Alias "InsertMenuA" (ByVal hMenu As _
Long, ByVal nPosition As Long, ByVal wFlags As Long, ByVal wIDNewItem As Long, _
ByVal lpNewItem As String) As Long
Declare Function AppendMenu Lib "user32" Alias "AppendMenuA" (ByVal hMenu As _
Long, ByVal wFlags As Long, ByVal wIDNewItem As Long, ByVal lpNewItem As _
String) As Long
Declare Function AppendMenuBynum Lib "user32" Alias "AppendMenuA" (ByVal hMenu _
As Long, ByVal wFlags As Long, ByVal wIDNewItem As Long, ByVal lpNewItem As _
Long) As Long
Declare Function ModifyMenu Lib "user32" Alias "ModifyMenuA" (ByVal hMenu As _
Long, ByVal nPosition As Long, ByVal wFlags As Long, ByVal wIDNewItem As Long, _
ByVal lpstring As String) As Long
Declare Function RemoveMenu Lib "user32" (ByVal hMenu As Long, ByVal nPosition _
As Long, ByVal wFlags As Long) As Long
Declare Function DeleteMenu Lib "user32" (ByVal hMenu As Long, ByVal nPosition _
As Long, ByVal wFlags As Long) As Long
Declare Function SetMenuItemBitmaps Lib "user32" (ByVal hMenu As Long, ByVal _
nPosition As Long, ByVal wFlags As Long, ByVal hBitmapUnchecked As Long, ByVal _
hBitmapChecked As Long) As Long

```

```
Declare Function GetMenuCheckMarkDimensions Lib "user32" () As Long
Declare Function TrackPopupMenu Lib "user32" (ByVal hMenu As Long, ByVal _
wFlags As Long, ByVal x As Long, ByVal y As Long, ByVal nReserved As Long, _
ByVal hwnd As Long, lprc As RECT) As Long
Declare Function TrackPopupMenuBynum Lib "user32" Alias "TrackPopupMenu" _
(ByVal hMenu As Long, ByVal wFlags As Long, ByVal x As Long, ByVal y As Long, _
ByVal nReserved As Long, ByVal hwnd As Long, ByVal lprc As Long) As Long
Declare Function GetCursorPos Lib "user32" (lpPoint As POINTAPI) As Long
Declare Function DeleteObject Lib "gdi32" (ByVal hObject As Long) As Long
Declare Function ModifyMenuBynum Lib "user32" Alias "ModifyMenuA" (ByVal hMenu _
As Long, ByVal nPosition As Long, ByVal wFlags As Long, ByVal wIDNewItem As _
Long, ByVal lpstring As Long) As Long

Declare Function GetObjectAPI Lib "gdi32" Alias "GetObjectA" (ByVal hObject As _
Long, ByVal nCount As Long, lpObject As Any) As Long
Declare Function CreateCompatibleDC Lib "gdi32" (ByVal hdc As Long) As Long
Declare Function SelectObject Lib "gdi32" (ByVal hdc As Long, ByVal hObject As _
Long) As Long
Declare Function BitBlt Lib "gdi32" (ByVal hDestDC As Long, ByVal x As Long, _
ByVal y As Long, ByVal nWidth As Long, ByVal nHeight As Long, ByVal hSrcDC As _
Long, ByVal xSrc As Long, ByVal ySrc As Long, ByVal dwRop As Long) As Long
Declare Function DeleteDC Lib "gdi32" (ByVal hdc As Long) As Long
Declare Function CreateSolidBrush Lib "gdi32" (ByVal crColor As Long) As Long
Declare Function Rectangle Lib "gdi32" (ByVal hdc As Long, ByVal X1 As Long, _
ByVal Y1 As Long, ByVal X2 As Long, ByVal Y2 As Long) As Long
Declare Function GetMenuContextHelpId Lib "user32" (ByVal hMenu As Long) As Long
Declare Function GetMenuDefaultItem Lib "user32" (ByVal hMenu As Long, ByVal _
fByPos As Long, ByVal gmdiFlags As Long) As Long
Declare Function SetMenuContextHelpId Lib "user32" (ByVal hMenu As Long, ByVal _
dw As Long) As Long
Declare Function SetMenuDefaultItem Lib "user32" (ByVal hMenu As Long, ByVal _
uItem As Long, ByVal fByPos As Long) As Long
Declare Function SetMenuItemInfo Lib "user32" Alias "SetMenuItemInfoA" (ByVal _
hMenu As Long, ByVal un As Long, ByVal bool As Long, lpcMenuItemInfo As _
MENUITEMINFO) As Long

#Else
```



```

Declare Function GetVersion& Lib "Kernel" ()
Declare Function DeleteDC% Lib "GDI" (ByVal hdc%)
Declare Function BitBlt% Lib "GDI" (ByVal hDestDC%, ByVal x%, ByVal y%, ByVal _
nWidth%, ByVal nHeight%, ByVal hSrcDC%, ByVal xSrc%, ByVal ySrc%, ByVal dwRop&)
Declare Function SelectObject% Lib "GDI" (ByVal hdc%, ByVal hObject%)
Declare Function CreateCompatibleDC% Lib "GDI" (ByVal hdc%)
Declare Function CreateBitmapIndirect% Lib "GDI" (lpBitmap As BITMAP)
Declare Function GetObjectAPI% Lib "GDI" Alias "GetObject" (ByVal hObject%, _
ByVal nCount%, lpObject As Any)
Declare Function DeleteObject% Lib "GDI" (ByVal hObject%)
Declare Sub GetCursorPos Lib "User" (lpPoint As POINTAPI)
Declare Function LoadMenu% Lib "User" (ByVal hInstance%, ByVal lpstring$)
Declare Function GetMenu% Lib "User" (ByVal hwnd%)
Declare Function GetMenuCheckMarkDimensions& Lib "User" ()
Declare Function GetMenuItemCount% Lib "User" (ByVal hMenu%)
Declare Function GetMenuItemID% Lib "User" (ByVal hMenu%, ByVal nPos%)
Declare Function GetMenuState% Lib "User" (ByVal hMenu%, ByVal wID%, ByVal _
wFlags%)
Declare Function GetMenuString% Lib "User" (ByVal hMenu%, ByVal wIDItem%, ByVal _
lpstring$, ByVal nMaxCount%, ByVal wFlag%)
Declare Function GetSubMenu% Lib "User" (ByVal hMenu%, ByVal nPos%)
Declare Function GetSystemMenu% Lib "User" (ByVal hwnd%, ByVal bRevert%)
Declare Function HiliteMenuItem% Lib "User" (ByVal hwnd%, ByVal hMenu%, ByVal _
wIDHiliteItem%, ByVal wHilite%)
Declare Function InsertMenu% Lib "User" (ByVal hMenu%, ByVal nPosition%, ByVal _
wFlags%, ByVal wIDNewItem%, ByVal lpNewItem As Any)
Declare Function InsertMenuBynum% Lib "User" Alias "InsertMenu" (ByVal hMenu%, _
ByVal nPosition%, ByVal wFlags%, ByVal wIDNewItem%, ByVal lpNewItem&)
Declare Function InsertMenuBystring% Lib "User" Alias "InsertMenu" (ByVal _
hMenu%, ByVal nPosition%, ByVal wFlags%, ByVal wIDNewItem%, ByVal lpNewItem$)
Declare Function IsMenu% Lib "User" (ByVal hMenu%)
Declare Function ModifyMenu% Lib "User" (ByVal hMenu%, ByVal nPosition%, ByVal _
wFlags%, ByVal wIDNewItem%, ByVal lpstring As Any)
Declare Function ModifyMenuBynum% Lib "User" Alias "ModifyMenu" (ByVal hMenu%, _
ByVal nPosition%, ByVal wFlags%, ByVal wIDNewItem%, ByVal lpstring&)
Declare Function ModifyMenuBystring% Lib "User" Alias "ModifyMenu" (ByVal _
hMenu%, ByVal nPosition%, ByVal wFlags%, ByVal wIDNewItem%, ByVal lpstring$)
Declare Function RemoveMenu% Lib "User" (ByVal hMenu%, ByVal nPosition%, ByVal _
wFlags%)
Declare Function SetMenu% Lib "User" (ByVal hwnd%, ByVal hMenu%)
Declare Function SetMenuItemBitmaps% Lib "User" (ByVal hMenu%, ByVal _
nPosition%, ByVal wFlags%, ByVal hBitmapUnchecked%, ByVal hBitmapChecked%)

```

```

Declare Function TrackPop-upMenu% Lib "User" (ByVal hMenu%, ByVal wFlags%, _
ByVal x%, ByVal y%, ByVal nReserved%, ByVal hwnd%, lpRect As Any)
Declare Function TrackPop-upMenuBynum% Lib "User" Alias "TrackPop-upMenu" _
(ByVal hMenu%, ByVal wFlags%, ByVal x%, ByVal y%, ByVal nReserved%, ByVal _
hwnd%, ByVal lpRect&)
Declare Function CreateSolidBrush% Lib "GDI" (ByVal crColor&)
Declare Function Rectangle% Lib "GDI" (ByVal hdc%, ByVal X1%, ByVal Y1%, ByVal _
X2%, ByVal Y2%)

#End If

```

On load, the application determines if it is running in the Windows 95 environment and saves this information in the `IsWindows95` public variable. This is used to select between two possible approaches to analyzing a menu hierarchy. On Unload, the program deletes any bitmap objects that it has created.

Listing 10.3 MENULOOK.FRM

```

Option Explicit
Dim IsWindows95 As Boolean

Private Sub Form_Load()
#If Win32 Then
    Dim os As OSVERSIONINFO
    Dim di&
    Print "Click anywhere on the form to"
    Print "bring up a tracked pop-up menu"
    Randomize
    os.dwOSVersionInfoSize = Len(os)
    di = GetVersionEx(os)
    If os.dwPlatformId = VER_PLATFORM_WIN32_WINDOWS Then IsWindows95 = True
#End If
' We never let IsWindows95 get set to True in 16 bits, because
' it is only used for Win32 specific APIs
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Dim x%, di&
    ' If a new check bitmap was set, we need to destroy it
    ' otherwise the resources used by the bitmap will not
    ' be freed.
    If NewCheck <> 0 Then
        di = DeleteObject(NewCheck)
    End If
    ' The same applies to the random menu bitmaps

```




Listing 10.3 MENULOOK.FRM (Continued)

```

    For x% = fl To 32
        If BMHandles(x%) <> fl Then
            di = DeleteObject(BMHandles(x%))
        End If
    Next x%
    If FloatBitmap <> 0 Then
        Call DeleteObject(FloatBitmap)
    End If

End Sub

```

The Label1 control is used to indicate which menu has been clicked. This lets you verify that a menu entry is, in fact, still generating a Visual Basic click event even after it has been changed into a bitmap or customized in some other fashion. All of the menu click events are displayed in this control. Some of the click events also toggle the check state of the entry.

```

' Let the system know this menu has been clicked
,

Private Sub MenuArray1_Click(Index As Integer)
    Label1.Caption = "Array1(" + Str$(Index) + ") selected."

End Sub

,

' Let the system know this menu has been clicked'
,

Private Sub MenuArt_Click(Index As Integer)
    Label1.Caption = "Menu Random Art (" + Str$(Index) + ") selected."

End Sub

,

' Let the system know this menu has been clicked'
,

Private Sub MenuEntry1_Click()
    ' Check or uncheck the menu each time it is clicked
    Label1.Caption = "Entry1 selected."
    MenuEntry1.Checked = Not MenuEntry1.Checked
End Sub

,

' Let the system know this menu has been clicked'
,

Private Sub MenuEntry2_Click()
    ' Check or uncheck the menu each time it is clicked
    If MenuEntry2.Checked Then MenuEntry2.Checked = fl Else MenuEntry2.Checked = -1

```

```
        Labell.Caption = "Entry2 selected."

End Sub

'
' Let the system know this menu has been clicked'
'
Private Sub MenuFloat_Click()
    Labell.Caption = "Floating selected."

End Sub

'
' Let the system know this menu has been clicked'
'
Private Sub MenuFloat1_Click(Index As Integer)
    Labell.Caption = "Float1(" + Str$(Index) + ") selected."
End Sub

'
' Let the system know this menu has been clicked'
'
Private Sub MenuRandomTop_Click()
    Labell.Caption = "Menu Random selected."
End Sub

'
' Let the system know this menu has been clicked'
'
Private Sub MenuSub1Entry1_Click()
    Labell.Caption = "Sub1Entry1 selected."
End Sub

'
' Let the system know this menu has been clicked'
'
Private Sub MenuSub1Entry2_Click()
    Labell.Caption = "Sub1Entry2 selected."
End Sub

'
' Let the system know this menu has been clicked'
'
Private Sub MenuSub-menu1_Click()
    Labell.Caption = "Sub-menu1 selected."
End Sub

'
' Let the system know this menu has been clicked'
'
Private Sub MenuTop_Click()
```



```
Label1.Caption = "MenuTop selected."
End Sub
```

The `CmdEntry2Chk_Click` function demonstrates several important techniques. First, it shows how to obtain the handle to the forms top level menu and how to obtain one of the pop-up menus that belong to the top level menu entries. This technique is repeated throughout the application.

It is critical, when setting a bitmap into a menu entry, that the bitmap not change or be destroyed while it is in use by the menu. This function obtains a new bitmap for use as a checkmark by calling the `GetNewCheck` function. In this case the unchecked state is left at its default (nothing shown); however, it is possible to set a checkmark for both the checked and unchecked states. A handle to the bitmap is stored for destruction when the form is unloaded.

```
,
' Create a box checkmark bitmap for the Entry2 menu
,
Private Sub CmdEntry2Chk_Click()
#If Win32 Then
    Dim topmenuhnd&
    Dim floatmenu&
    Dim NewCheck&
#Else
    Dim NewCheck%
    Dim topmenuhnd%
    Dim floatmenu%
#End If
Dim oldbkcolor&
Dim di&

' Get the new checkmark bitmap
NewCheck = GetNewCheck()
' Get a handle to the top level menu
topmenuhnd = GetMenu(Menulook.hwnd)
' Get a handle to the first popup
floatmenu = GetSubMenu(topmenuhnd, fl)
' And set the new check bitmap for the first (entry1) menu item
di = SetMenuItemBitmaps(floatmenu, 1, MF_BYPOSITION, fl, NewCheck)
' Check the entry
MenuEntry2.Checked = -1
' Remind the user to look at it.
MsgBox "Look at the Menu1 - Entry2 menu"
End Sub
```

The `GetNewCheck` function creates a device-dependent bitmap using the same techniques demonstrated in Chapter 9. It starts by determining the standard checkmark dimensions to use. It shrinks this rectangle by two pixels on a side for aesthetic reasons—there is no Windows requirement to reduce the size of the bitmap. It draws the image that it will use on the `Picture2` control.

The `CreateBitmapIndirect` function then creates a bitmap, and `CreateCompatibleDC` creates a memory device context with which it can be accessed. The copy is accomplished with a simple `BitBlt` operation, after which the device context is deleted. The function returns the bitmap handle, which should be destroyed by the application before it terminates.

```
' Create a custom bitmap checkmark and return a
' handle to that bitmap.
,
Private Function GetNewCheck&()
    Dim bm As BITMAP
    Dim pt As POINTS
    #If Win32 Then
        Dim newbm&
        Dim tdc&, oldbm&
        Dim br&, oldbrush&
    #Else
        Dim newbm%
        Dim tdc%, oldbm%
        Dim br%, oldbrush%
    #End If
    Dim markdims&
    Dim di&

    ' Find out how big the checkmark should be.
    markdims& = GetMenuCheckMarkDimensions()
    agDWORDto2Integers markdims&, pt.x, pt.y

    ' And create a magenta brush (the checkmark will be
    ' a magenta filled rectangle
    br = CreateSolidBrush(QBColor(5))

    Picture2.Cls
    oldbrush = SelectObject(Picture2.hdc, br)
    ' Draw the rectangle.
    di = Rectangle(Picture2.hdc, 2, 2, pt.x - 2, pt.y - 2)
    di = SelectObject(Picture2.hdc, oldbrush)
    di = DeleteObject(br) ' Dump the brush.

    ' Create a compatible bitmap of the right size
    ' was: di% = GetObjectAPI(Picture2.Image, 14, agGetAddressForObject&(bm))
    di = GetObjectAPI(Picture2.Image, Len(bm), bm)
    bm.bmBits = fl
    bm.bmWidth = pt.x
    bm.bmHeight = pt.y
    newbm = CreateBitmapIndirect(bm)
    ' And create a memory device context to use
    tdc = CreateCompatibleDC(Picture2.hdc)
    oldbm = SelectObject(tdc, newbm)
    ' Copy in the new checkmark
    di = BitBlt(tdc, fl, fl, pt.x, pt.y, Picture2.hdc, fl, fl, SRCCOPY)
```



```

oldbm = SelectObject(tdc, oldbm)
di = DeleteDC(tdc)
GetNewCheck = newbm

```

End Function

The `CmdAddBitmap_Click` function sets a bitmap into the entry for a menu. In the past, this example has used the `Picture1` control's `Picture` property because that bitmap remained unchanged through the life of the program. On some operating system configurations, this no longer seems to be the case. For this reason, the example has been modified for this edition to use the `CopyPictureImage` function to make a copy of the bitmap.

This function creates a menu to modify by using standard Visual Basic menu control array techniques. This ensures that a VB menu control is assigned to the entry that is to be modified. The `GetMenuItemID` API function is used to obtain the menu ID for the new menu entry. This menu ID is used later when calling the `ModifyMenu` command to make sure that all `WM_COMMAND` messages from the modified menu entry will continue to go to the VB menu control.

```

' This command adds the bitmap that is in the Picture1
' control to the Floating menu.
'
Private Sub CmdAddBitmap_Click()
#If Win32 Then
    Dim topmenuhnd&
    Dim floatmenu&
    Dim menuid&
#Else
    Dim topmenuhnd%
    Dim floatmenu%
    Dim menuid%
#End If
    Dim di&

    ' Get a handle to the top level menu
    topmenuhnd = GetMenu(Menulook.hwnd)
    ' And get a handle to the Floating popup menu.
    floatmenu = GetSubMenu(topmenuhnd, 1)
    ' If 3rd (bitmap) entry is already loaded, exit now
    ' (we only load the bitmap once)
    If GetMenuItemCount(floatmenu) >= 3 Then Exit Sub

    ' First, add a menu entry under VB - this provides us
    ' with a menu entry that can be replaced with a bitmap,
    ' but whose menu ID will be the same (so it will work
    ' properly)
    Load MenuFloat1(2)
    ' Now get the ID of that entry
    menuid = GetMenuItemID(floatmenu, 2)

```

```
If FloatBitmap = 0 Then FloatBitmap = CopyPictureImage(Picture1)
' And replace it with a bitmap.
di = ModifyMenuBynum(floatmenu, 2, MF_BITMAP Or MF_BYPOSITION, menuid, _
FloatBitmap)
```

End Sub

This function combines the techniques shown by the **CmdAddBitmap_Click** and **CmdEntry2Chk_Click** functions to create custom bitmap menu entries. For this program, a number of random rectangles are drawn into an empty bitmap (the **Picture2**) control for use as a menu entry bitmap. Note how a copy of the picture control's bitmap is used. The bitmap is stored in a global array for destruction when the form is unloaded.

```
,
' This command creates a random bitmap and loads it into
' the Random popup menu.
,
Private Sub CmdAddRandom_Click()
#If Win32 Then
    Dim topmenuhnd&
    Dim floatmenu&
    Dim menuid&
    Dim newmenupos&
    Dim newbm&
#Else
    Dim topmenuhnd%
    Dim floatmenu%
    Dim menuid%
    Dim newmenupos%
    Dim newbm%
#End If
    Dim pw!, ph!
    Dim x%, di&

    ' First get the width and height of the picture2 control
    pw! = Picture2.ScaleWidth
    ph! = Picture2.ScaleHeight
    ' Get a handle to the Random popup menu
    topmenuhnd = GetMenu(Menulook.hwnd)
    floatmenu = GetSubMenu(topmenuhnd, 2)
    ' Find out how many menu items are already in the popup
    newmenupos = GetMenuItemCount(floatmenu)

    ' Load a VB menu entry at that position
    Load MenuArt(newmenupos)
    ' And get the MenuID for that entry
    menuid = GetMenuItemID(floatmenu, newmenupos)
```



```

' Draw some stuff on picture2
Picture2.Cls
For x% = fl To 5 ' Random rectangles
    Picture2.Line (Rnd * pw!, Rnd * ph!)-(Rnd * pw!, Rnd * ph!), _
        QBColor(CInt(Rnd * 15)), B
Next x%
' Get a bitmap that is a copy of the picture2 control
newbm = CopyPictureImage(Picture2)
For x% = fl To 32
    If BMHandles(x%) = fl Then Exit For
Next x%
If x% = 32 Then ' No room to store the bitmap handle
    di = DeleteObject(newbm)
    Exit Sub
End If
BMHandles(x%) = newbm
' And place that bitmap in the menu
di = ModifyMenuBynum(floatmenu, newmenupos, MF_BITMAP Or MF_BYPOSITION, _
    menuid, newbm)

End Sub

```

When using a bitmap in a menu entry it is important that the bitmap not be destroyed while the menu is using it. The MenuLook program uses the Picture2 control as a source for images. However, we cannot simply use the bitmap property returned by the Image property of the control—that bitmap changes as the application executes. This function makes a copy of a bitmap contained in a picture control. The techniques shown here are based on the information in Chapter 9.

```

'
' This function makes a copy of the Image property
' of the specified image control and returns a handle to that bitmap
'
Private Function CopyPictureImage(SourceImage As PictureBox) As Long
    Dim bm As BITMAP
    #If Win32 Then
        Dim newbm&
        Dim tdc&, oldbm&
    #Else
        Dim newbm%
        Dim tdc%, oldbm%
    #End If
    Dim di&

    ' First get the information about the image bitmap
    di = GetObjectAPI(SourceImage.Image, Len(bm), bm)
    bm.bmBits = 0
    ' Create a new bitmap with the same structure and size
    ' of the image bitmap
    newbm = CreateBitmapIndirect(bm)

```

```

' Create a temporary memory device context to use
tdc = CreateCompatibleDC(SourceImage.hdc)
' Select in the newly created bitmap
oldbm = SelectObject(tdc, newbm)

' Now copy the bitmap from the persistent bitmap in
' picture 2 (note that picture2 has AutoRedraw set TRUE
di = BitBlt(tdc, 0, 0, bm.bmWidth, bm.bmHeight, SourceImage.hdc, 0, 0, _
SRCCOPY)
' Select out the bitmap and delete the memory DC
oldbm = SelectObject(tdc, oldbm)
di = DeleteDC(tdc)

' And return the new bitmap
CopyPictureImage = newbm
End Function

```

The `CmdTrack_Click` function demonstrates two techniques. First it shows how to temporarily hide a menu by setting the caption to the empty string. In practice you would probably want to disable the menu entry as well, and to position it to the right of all other menus. Even when hidden, the menu takes up space. Keep in mind that you cannot hide the menu by setting the menu control's `Visible` property to `FALSE`—that causes the menu entry and associated pop-up to be destroyed.

Second, this function demonstrates how to bring up a tracked pop-up menu at the cursor position.

```

'
' Hides the Floating menu entry in the caption,
' and turns it into a tracked popup menu
'
Private Sub CmdTrack_Click()
#If Win32 Then ' Port to correct handle type
Dim topmenuhnd%
Dim floatmenu%
#Else
Dim topmenuhnd%
Dim floatmenu%
#End If
Dim oldcap$, di%
Dim pt As POINTAPI

' Get a handle to the popup menu
topmenuhnd = GetMenu(Menulook.hwnd)
floatmenu = GetSubMenu(topmenuhnd, 1)

' Hide the menu entry by clearing the string
' temporarily. Note, don't make it invisible or the
' menu will go away!

```




```

        oldcap$ = MenuFloat.Caption
        MenuFloat.Caption = ""
        ' Find where the mouse cursor is and place the
        ' popup at that point
        GetCursorPos pt
        di = TrackPopupMenuBynum(floatmenu, TPM_CENTERALIGN, pt.x, pt.y, fl, _
        Menulook.hwnd, fl&)
        ' Restore the original popup name
        MenuFloat.Caption = oldcap$
    End Sub

    '
    ' Clicking anywhere on the form triggers the CmdTrack command
    ' button.
    '

    Private Sub Form_MouseUp(Button As Integer, Shift As Integer, x As Single, y As _
    Single)
        ' Clicking anywhere on the form triggers the CmdTrack button
        CmdTrack.value = -1
    End Sub

```

The `CmdAnalyze_Click` function obtains a handle to the top level menu for a form, then calls the `ViewMenu` command to display the structure of the menu in the `List1` control. `ViewMenu` is recursive (meaning it can make additional calls to itself to analyze sub-menus), so this single call will cause the entire menu structure to be displayed.

```

    '
    ' Get the form's menu and analyze it.
    '

    Private Sub CmdAnalyze_Click()
    #If Win32 Then
        Dim menuhnd&
    #Else
        Dim menuhnd%
    #End If

        ' Clear the listbox
        ' dl& = SendMessage(List1.hwnd, LB_RESETCONTENT, fl, fl&)
        List1.Clear
        ' Get a handle to the top level menu for this window
        menuhnd = GetMenu(Menulook.hwnd)
        ' And analyze it
        ViewMenu menuhnd
    End Sub

```

The `ViewMenu` function has the ability to add to the `List1` list box a description of the menu whose handle is passed to it as a parameter. It shows how to obtain the number of menu entries in a menu, and how to obtain information about each menu entry including the menu ID and state information.

This function makes a list of all pop-up menus attached to this menu, and after completing the description of each entry proceeds to recursively call itself for each pop-up sub-menu.

This function uses two different techniques for analyzing a menu. Under Win16 and Windows NT, it uses the standard menu API commands. Under Windows 95 it uses the extended menu API commands that work with the MENUITEMINFO structure. It is important to stress that Windows 95 is fully compatible with the standard menu API command set and that absolutely no benefit is gained in this example by using the new functions—on the contrary, it requires substantially more sophisticated string handling than is needed by the standard functions. The only reason that it is used in this example is because this is a sample program—and a good example of some of the techniques involved in working with complex structures. Also, keep in mind that the new functions, while only demonstrated under this example with Windows 95, work equally well on NT 4.0 and later.

The portions of this function that discuss the use of a byte array to retrieve string information from the MENUITEMINFO structure are examined in detail in Chapter 15 under the section titled “Dynamic Strings In Structures.” If you wish to pursue this subject further, you might want to skip ahead to read that section before continuing.

```

''
'   This function loads into list1 an analysis of the
'   specified menu
'
Private Sub ViewMenu(ByVal menuhnd&)
Dim menulen&
Dim di&
#If Win32 Then
Dim thismenu&
Dim menuid&
Dim currentpopup&
Dim menuinfo As MENUITEMINFO
#Else
Dim thismenu%
Dim menuid%
Dim currentpopup%
#End If
Dim db%
Dim menuflags%
Dim flagstring$
Dim menustring(128) As Byte
Dim menustring2 As String * 128
Dim context&

' This routine can analyze up to 32 popup sub-menus
' for each menu. We keep track of them here so that
' we can recursively analyze the popups after we

```

```

analyze the main menu.
Dim trackpopups&(32)

currentpopup = fl

List1.AddItem "Analysis of Menu handle# " & Hex$(menuhnd)

' Find out how many entries are in the menu.
menulen = GetMenuItemCount(menuhnd)
List1.AddItem "# of entries = " & Str$(menulen)

#If Win32 Then
context& = GetMenuContextHelpId(menuhnd)
If context& > fl Then List1.AddItem "Help Context ID = " & Str$(context&)
#End If

If IsWindows95 Then
    #If Win32 Then
        menuinfo.cbSize = Len(menuinfo)

        For thismenu = fl To menulen - 1
            ' cch field is reset each time
            menuinfo.cch = 127
            menuinfo.dwTypeData = agGetAddressForObject(menustring(fl))
            menuinfo.fMask = MIIM_DATA Or MIIM_ID Or MIIM_STATE Or _
                MIIM_SUB-MENU Or MIIM_TYPE Or MIIM_CHECKMARKS

            ' Get the ID for this menu
            ' It's a command ID, -1 for a popup, fl for a separator
            di = GetMenuItemInfo(menuhnd, thismenu, True, menuinfo)
            If di = fl Then
                List1.AddItem "Entry #" + Str$(thismenu) + _
                    " is unaccessable via GetMenuItemInfo"
            Else
                With menuinfo
                    ' Obtain all information about a menu
                    If .hSub-menu <> fl Then
                        ' Save it in the list of popups
                        trackpopups&(currentpopup) = thismenu
                        currentpopup = currentpopup + 1
                        ' Why do we use left$?
                        List1.AddItem "Entry #" + Str$(thismenu) + _
                            " is a sub-menu. Handle = " + Hex$(.hSub-menu) + _
                            " is " + Left$(StrConv(menustring, vbUnicode), .cch)
                        List1.AddItem "Flags: " & GetFlagStringNew$(.fType, _
                            .fState)
                    End If
                End With
            End If
        Next thismenu
    End If
End If

```

```

        If .fType = MFT_STRING Then
            List1.AddItem "Entry #" + Str$(thismenu) + _
                " is a string = " + Left$(StrConv(menustring, _
                    vbUnicode), .cch)
            List1.AddItem "Flags:" _
                + GetFlagStringNew$(.fType, .fState)
        Else
            ' Menu type is a bitmap, or otherwise not _
            supported by this application
            List1.AddItem "Entry #" & Str$(thismenu) & _
                "has flags: " + GetFlagStringNew$(.fType, _
                    .fState)
        End If

    End If

End With

End If

Next thismenu
#End If
Else
    For thismenu = fl To menulen - 1
        ' Get the ID for this menu
        ' It's a command ID, -1 for a popup, fl for a separator
        menuid = GetMenuItemID(menuhnd, thismenu)
        Select Case menuid
            Case fl ' It's a separator
                List1.AddItem "Entry #" + Str$(thismenu) + "is a separator"
            Case -1 ' It's a popup menu
                ' Save it in the list of popups
                trackpopups&(currentpopup) = thismenu
                currentpopup = currentpopup + 1
                ' And report that it's here
                db = GetMenuString(menuhnd, thismenu, menustring2, 127, _
                    MF_BYPOSITION)
                menuflags = GetMenuState(menuhnd, thismenu, MF_BYPOSITION)
                List1.AddItem "Entry #" + Str$(thismenu) + _
                    " is a sub-menu.
                Handle = " + Hex$(GetSubMenu(menuhnd, thismenu)) _
                    + " is " + Left$(menustring2, db)
                List1.AddItem "Flags: " + GetFlagString$(menuflags)

            Case Else ' A regular entry
                db = GetMenuString(menuhnd, menuid, menustring2, 127, _
                    MF_BYCOMMAND)
                List1.AddItem "Entry #" + Str$(thismenu) + " cmd = " + _
                    Str$(menuid) + " is " + Left$(menustring2, db)
                menuflags = GetMenuState(menuhnd, menuid, MF_BYCOMMAND)
                List1.AddItem "Flags: " + GetFlagString$(menuflags)
        End Select
    End For
End If

```



```

        Next thismenu
    End If
    If currentpopup > fl Then ' At least one popup was found
        List1.AddItem "Sub menus:"
        For thismenu = fl To currentpopup - 1
            menuid = trackpopups&(thismenu)
            ' Recursively analyze the popup menu.
            ViewMenu GetSubMenu(menuhnd, menuid)
        Next thismenu
    End If

```

```
End Sub
```

The `GetFlagString` and `GetFlagStringNew` functions obtain a list of the flags that are set for a menu entry. The two different techniques correspond to the standard and extended menu functions.

```

'
' Gets a string containing a description of the flags
' set for this menu item.
'
Private Function GetFlagString$(menuflags%)
    Dim f$
    If (menuflags% And MF_CHECKED) <> fl Then
        f$ = f$ + "Checked "
    Else
        f$ = f$ + "Unchecked "
    End If
    If (menuflags% And MF_DISABLED) <> fl Then
        f$ = f$ + "Disabled "
    Else
        f$ = f$ + "Enabled "
    End If
    If (menuflags% And MF_GRAYED) <> fl Then f$ = f$ + "Grayed "
    If (menuflags% And MF_BITMAP) <> fl Then f$ = f$ + "Bitmap "
    If (menuflags% And MF_MENUBARBREAK) <> fl Then f$ = f$ + "Bar-break "
    If (menuflags% And MF_MENUBREAK) <> fl Then f$ = f$ + "Break "
    If (menuflags% And MF_SEPARATOR) <> fl Then f$ = f$ + "Seperator "
    GetFlagString$ = f$
End Function

```

```

Public Function GetFlagStringNew$(ByVal fType&, ByVal fState&)
    Dim f$
    If (fState And MFS_CHECKED) <> fl Then
        f$ = f$ + "Checked "
    Else
        f$ = f$ + "Unchecked "
    End If
    If (fState And MFS_DISABLED) <> fl Then

```

```

        f$ = f$ + "Disabled "
    Else
        f$ = f$ + "Enabled "
    End If
    If (fState And MFS_GRAYED) <> fl Then f$ = f$ + "Grayed "
    If (fType And MFT_BITMAP) <> fl Then f$ = f$ + "Bitmap "
    If (fType And MFT_MENUBARBREAK) <> fl Then f$ = f$ + "Bar-break "
    If (fType And MFT_RADIOCHECK) <> fl Then f$ = f$ + "Radio "
    If (fType And MFT_MENUBREAK) <> fl Then f$ = f$ + "Break "
    If (fType And MFT_SEPARATOR) <> fl Then f$ = f$ + "Separator "
    GetFlagStringNew$ = f$

End Function

```

Suggestions for Further Practice

Here are a number of suggestions for ways to improve the MenuLook program.

- In the example, the custom menu example always draws bitmaps with a white background into the menu. Modify the example to determine the background color for menus and use it as the background color for the bitmap. (Hint: Look at the GetSysColor function in Chapter 6, and the PatBlt or drawing functions in Chapter 9.)
- Try implementing the new radio menu capability under Windows 95 or NT 4.0.
- Modify the MenuLook program so that it will analyze the current system menu.

SysMenu—A System Menu and Context Menu Demonstration

The SysMenu program demonstrates how you can modify the system menu of a Visual Basic application and how you can implement true Context menu support (as compared to the partial support offered by Visual Basic).

There is a catch, though: Both of these techniques require that you use subclassing. Subclassing was not supported under Visual Basic 4.0 and thus required use of third party subclassing controls. It is possible to subclass a window within Visual Basic 5.0 using the SetWindowLong function. However, despite the directions given in the VB5 documentation, you should never subclass a window using this technique within a VB application. It will work—but you will run into severe problems debugging your application, for as soon as it enters break mode or the Stop command is invoked, the code that runs the window function stops running as well. Thus this technique can interfere with the normal operation of your system. You should always subclass windows in an



application using an in-process DLL component (which can be written using VB5). This book includes a fully functional demonstration version of a subclassing OLE control `dwsbc32d.ocx`, which is one of two controls included in Desaware's Spyworks package (the other is written in VB and includes complete source code). This control will allow you to run the sample programs and learn how to take advantage of the capabilities that subclassing provides; but if you wish to do so in your own applications you will need to purchase a subclassing control that is licensed for distribution or write one yourself.

Using SysMenu

The SysMenu screen is shown in Figure 10.5. As you can see, a new menu entry was added to the system menu. You can enter any text into the text box (which is partially obscured by the dropped menu in the figure) and click on the "Add To System Menu" button to insert a new entry into the system menu.

Figure 10.5
SysMenu screen
showing modified
system menu



Context menus were discussed earlier in the chapter text. When you click the right mouse button on the form, a `WM_CONTEXTMENU` message is sent to the control that you clicked on. A control has the option of processing the message itself, or allowing it to be sent on to the container form. Most controls do not process the message themselves, but under Windows 95 and NT 4.0 the text control handles its own context menu. The `dwsbc32d.ocx` control uses subclassing to detect the `WM_CONTEXTMENU` message for both the form and the text control, thus allowing it to handle the default context menu for controls that do not have their own, and to override the context menu for the text control.

Project Description

The SysMenu project includes three files. `SYSTEMENU.FRM` is the only form used in the program. This program also uses `MENULOOK.BAS` which contains the constant type, variable, and function declarations. `APIGID32.BAS` contains the declarations for the `APIGID32.DLL` dynamic link library included with this book. Listing 10.4 contains the listing for `SYSTEMENU.FRM`. `MENULOOK.BAS` was listed in the previous section for the `MenuLook.vbp` example.

Listing 10.4 SYSMENU.FRM

```
VERSION5.frl
Begin VB.Form frmSysMenu
    Caption           = "Change System Menu & Context Menus"
    ClientHeight      = 156fl
    ClientLeft        = 1fl95
    ClientTop         = 1515
    ClientWidth       = 579fl
    Height            = 1965
    Left              = 1fl35
    LinkTopic         = "Form1"
    ScaleHeight       = 156fl
    ScaleWidth        = 579fl
    Top               = 117fl
    Width             = 591fl
    Begin VB.PictureBox picTarget
        Height         = 495
        Left           = 72fl
        ScaleHeight    = 465
        ScaleWidth     = 885
        TabIndex       = 3
        Top            = 84fl
        Visible        = fl 'False
        Width          = 915
    End
    Begin VB.CheckBox chkContext
        Caption        = "Replace Context Menu"
        Height         = 255
        Left           = 27flfl
        TabIndex       = 2
        Top            = 96fl
        Width          = 2115
    End
    Begin VB.CommandButton cmdAddSystem
        Caption        = "Add To System Menu"
        Height         = 495
        Left           = 27flfl
        TabIndex       = 1
        Top            = 3flfl
        Width          = 1995
    End
    Begin VB.TextBox txtMenu
        Height         = 315
        Left           = 66fl
        TabIndex       = fl
        Text           = "NewMenu"
        Top            = 36fl
        Width          = 1815
    End
End
```



The `SubClass1` control `Messages` property contains a list of the messages that this control should intercept. You should set this to the `WM_SYSCOMMAND`, `WM_COMMAND`, and `WM_CONTEXTMENU` messages.

```
Begin DwsbcLib.SubClass SubClass1
    Left           = 514fl
    Top            = 48fl
    _Version       = 262144
    _ExtentX       = 847
    _ExtentY       = 847
    _StockProps    = fl
    CtlParam       = "frmSysMenu"
    Persist        = fl
    RegMessage1    = ""
    RegMessage2    = ""
    RegMessage3    = ""
    RegMessage4    = ""
    RegMessage5    = ""
    Detect          = fl
    Messages       = "SYSMENU.frx":fffffl
End
End
Attribute VB_Name = "frmSysMenu"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = TrueAttribute VB_Exposed = False
```

The `NewContextMenu` variable will hold a handle to a custom-created pop-up menu. This menu uses menu command IDs that start with `&H2000`. This number was chosen arbitrarily—as long as the menu ID is above `WM_USER` (`&H400`), it should not interfere with the normal behavior of the control.

```
Option Explicit
Dim CurrentID&
Dim NewContextMenu&

Const WM_CONTEXTMENU = &H7B
Const WM_SYSCOMMAND = &H112
Const WM_MENUBASE = &H2ffffl
Const WM_COMMAND = &H111

Const SCOFFSET = 2ffffl
```

During form load a new pop-up menu is created which has three entries. The control was set to subclass the form by setting the `CtlParam` parameter to `frmSysMenu`, the name of the form. The `AddHwnd` property is used to specify additional windows to subclass.

```
Private Sub Form_Load()
    Dim di&
```

```
NewContextMenu = CreatePopupMenu()
di = AppendMenu(NewContextMenu, MF_STRING, WM_MENUBASE, "Entry 1")
di = AppendMenu(NewContextMenu, MF_STRING, WM_MENUBASE + 1, "Entry 2")
di = AppendMenu(NewContextMenu, MF_STRING, WM_MENUBASE + 2, "Entry 3")
SubClass1.AddHwnd = txtMenu.hwnd
SubClass1.AddHwnd = picTarget.hwnd
End Sub
```

Destroy the menu when the application is unloaded.

```
Private Sub Form_Unload(Cancel As Integer)
    If NewContextMenu Then Call DestroyMenu(NewContextMenu)
End Sub
```

Note the use of the **GetSystemMenu** API to retrieve the system menu.

```
Private Sub cmdAddSystem_Click()
    Dim sm&, di&
    If Len(txtMenu.Text) = 0 Then
        MsgBox "Must specify menu text"
        Exit Sub
    End If
    sm& = GetSystemMenu(hwnd, False)
    di& = AppendMenu(sm, MF_STRING, SCOFFSET + CurrentID, txtMenu.Text)
    CurrentID = CurrentID + 1
End Sub
```

This particular application was set up so that one `dwsbc32d.ocx` control performs all of the subclassing. This means that the `WndMessage` event receives messages from every window that is being subclassed. You can use the `hwnd` property to determine which control a message is destined for.

The `WM_SYSCOMMAND` message is sent by the control menu messages. If the message number, which is in the `wp` parameter, is not one of the ones that we added, the function exits and the default operation for the message takes place. If it is a message from one of the menu entries we added, a message box is displayed indicating its receipt. The `nofdef` property is then set to tell Windows that it need not process the message any further.

The `WM_CONTEXTMENU` message is sent when a context menu is requested. When this message is received, the function uses the `TrackPopupMenu` function to show a pop-up menu. Messages from this pop-up menu are directed to a picture box. This is necessary, as these messages can interfere with the behavior of other controls under Visual Basic (leading to fatal exceptions). Even so, you must intercept and block the `WM_COMMAND` messages coming into the picture control. Other messages (such as `WM_MENUSELECT`) are sent to a control while the pop-up menu is displayed. These, too, can cause problems with other controls, an issue that we avoid by using a picture control.

```
,
' This event is triggered for every WM_SYSCOMMAND message
```



```

Private Sub SubClass1_WndMessage(hwnd As Long, msg As Long, wp As Long, lp As _
Long, retval As Long, nodef As Integer)
    Dim sm&, di&
    Dim usestring$
    Dim usex%, usey%
    Select Case msg
        Case WM_COMMAND
            ' We only care about WM_COMMAND messages to picture control
            If hwnd <> picTarget.hwnd Then Exit Sub
            Call agDWORDto2Integers(wp, usex, usey)
            MsgBox "Received command # " & Hex$(usex)
            nodef = True
        Case WM_SYSCOMMAND
            ' If it's not one of the ones we added, just exit
            If wp < SCOFFSET Or wp >= (SCOFFSET + CurrentID) Then Exit Sub
            ' Get the text for this menu entry
            sm& = GetSystemMenu(hwnd, False)
            usestring$ = String$(128, fl)
            di = GetMenuString(sm, wp, usestring, 127, MF_BYCOMMAND)
            MsgBox Left$(usestring, di), vbOKOnly, "System Menu Clicked is:"
            nodef = True
        Case WM_CONTEXTMENU
            ' Only trap the context menu if requested
            If chkContext.value = fl Then Exit Sub
            ' Get the location of the mouse click
            Call agDWORDto2Integers(lp, usex, usey)
            Call TrackPop-upMenuBynum(NewContextMenu, TPM_LEFTALIGN Or _
            TPM_RIGHTBUTTON, _
            usex, usey, fl, picTarget.hwnd, fl)
            nodef = True ' Don't let control get the message!
    End Select
End Sub

```

Suggestions for Further Practice

This sample illustrates one of the simplest applications of subclassing. Here is an experiment or two for you to try:

- What happens if you set the nodef property to TRUE to prevent the default processing of standard system menu messages? Could you prevent a window from being closed? Could you change the meaning of one of the standard messages?
- Chapter 6 showed how you can obtain the window handle for any window in the system. Can you add a system menu entry to another application and have it trigger an event in your application? (Hint: Yes, this is possible with the dwsbc32d.ocx control, though not with subclassing techniques written using VB5.)