Appendix

Windows API Functions

The Visual Basic language provides a rich set of functions, commands, and objects, but in many cases they don't meet all the needs of a professional programmer. Just to name a few shortcomings, Visual Basic doesn't allow you to retrieve system information—such as the name of the current user—and most Visual Basic controls expose only a fraction of the features that they potentially have.

Expert programmers have learned to overcome most of these limitations by directly calling one or more Windows API functions. In this book, I've resorted to API functions on many occasions, and it's time to give these functions the attention they deserve. In contrast to my practice in most other chapters in this book, however, I won't even try to exhaustively describe all you can do with this programming technique, for one simple reason: The Windows operating system exposes several thousand functions, and the number grows almost weekly.

Instead, I'll give you some ready-to-use routines that perform specific tasks and that remedy a few of the deficiencies of Visual Basic. You won't see much theory in these pages because there are many other good sources of information available, such as the Microsoft Developer Network (MSDN), a product that should always have a place on the desktop of any serious developer, regardless of his or her programming language.

A World of Messages

The Microsoft Windows operating system is heavily based on messages. For example, when the user closes a window, the operating system sends the window a WM_CLOSE message. When the user types a key, the window that has the focus receives a WM_CHAR message, and so on. (In this context, the term *window* refers to both top-level windows and child controls.) Messages can also be sent to a window or a control to affect its appearance or behavior or to retrieve the information it contains. For example, you can send the WM_SETTEXT message to most windows and controls to assign a string to their contents, and you can send the WM_GETTEXT message to read their current contents. By means of these messages, you can set or read the caption of a top-level window or set or read the *Text* property of a TextBox control, just to name a few common uses for this technique.

Broadly speaking, messages belong to one of two families: They're *control messages* or *notification messages*. Control messages are sent by an application to a window or a control to set or retrieve its contents, or to modify its behavior or appearance. Notification messages are sent by the operating system to windows or controls as the result of the actions users perform on them.

Visual Basic greatly simplifies the programming of Windows applications because it automatically translates most of these messages into properties, methods, and events. Instead of using WM_SETTEXT and WM_GETTEXT messages, Visual Basic programmers can reason in terms of *Caption* and *Text* properties. Nor do they have to worry about trapping WM_CLOSE messages sent to a form because the Visual Basic runtime automatically translates them into *Form_Unload* events. More generally, control messages map to properties and methods, whereas notification messages map to events.

Not all messages are processed in this way, though. For example, the TextBox control has built-in undo capabilities, but they aren't exposed as properties or methods by Visual Basic, and therefore they can't be accessed by "pure" Visual Basic code. (In this appendix, *pure Visual Basic* means code that doesn't rely on external API functions.) Here's another example: When the user moves a form, Windows sends the form a WM_MOVE message, but the Visual Basic runtime traps that message without raising an event. If your application needs to know when one of its windows moves, you're out of luck.

By using API functions, you can work around these limitations. In this section, I'll show you how you can send a control message to a window or a control to affect its appearance or behavior, while in the "Callback and Subclassing" section, I'll illustrate a more complex programming technique, called *window subclassing*, which lets you intercept the notification messages that Visual Basic doesn't translate to events.

Before you can use an API function, you must tell Visual Basic the name of the DLL that contains it and the type of each argument. You do this with a *Declare* statement, which must appear in the declaration section of a module. *Declare* statements must be declared as Private in all types of modules except BAS modules (which also accept Public *Declare* statements that are visible from the entire application). For additional information about the *Declare* statement, see the language documentation.

The main API function that you can use to send a message to a form or a control is *SendMessage*, whose *Declare* statement is this:

```
Private Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
    (ByVal hWnd As Long, ByVal wMsg As Long, _
    ByVal wParam As Long, lParam As Any) As Long
```

The *hWnd* argument is the handle of the window to which you're sending the message (it corresponds to the window's *hWnd* property), *wMsg* is the message number (usually expressed as a symbolic constant), and the meaning of the *wParam* and *lParam* values depend on the particular message you're sending. Notice that *lParam* is declared with the *As Any* clause so that you can pass virtually anything to this argument, including any simple data type or a UDT. To reduce the risk of accidentally sending invalid data, I've prepared a version of the *SendMessage* function, which accepts a Long number by value, and another version that expects a String passed by value. These are the so called type-safe *Declare* statements:

```
Private Declare Function SendMessageByVal Lib "user32" _
    Alias "SendMessageA" (ByVal hWnd As Long, ByVal wMsg As Long, _
    ByVal wParam As Long, Byval lParam As Long) As Long

Private Declare Function SendMessageString Lib "user32" _
    Alias "SendMessageA" ByVal hWnd As Long, ByVal wMsg As Long, _
    ByVal wParam As Long, ByVal lParam As String) As Long
```

Apart from such type-safe variants, the *Declare* functions used in this chapter, as well as the values of message symbolic constants, can be obtained by running the API Viewer utility that comes with Visual Basic. (See Figure A-1 on the following page.)

> **CAUTION** When working with API functions, you're in direct touch with the operating system and aren't using the safety net that Visual Basic offers. If you make an error in the declaration or execution of an API function, you're likely to get a General Protection Fault (GPF) or another fatal error that will immediately shut down the Visual Basic environment. For this reason, you should carefully double-check the *Declare* statements and the arguments you pass to an API function, and you should always save your code before running the project.

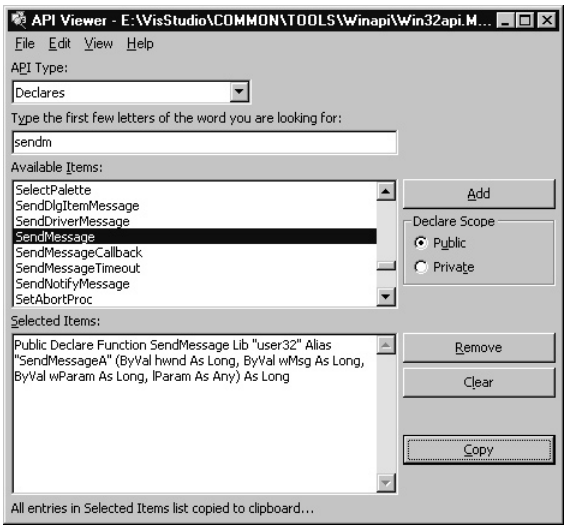


Figure A-1. *The API Viewer utility has been improved in Visual Basic 6 with the capability to set the scope of* Const *and* Type *directives and* Declare *statements.*

Multiline TextBox Controls

The *SendMessage* API function is very useful with multiline TextBox controls because only a small fraction of their features is exposed through standard properties and methods. For example, you can determine the number of lines in a multiline TextBox control by sending it an EM_GETLINECOUNT message:

```
LineCount = SendMessageByVal(Text1.hWnd, EM_GETLINECOUNT, 0, 0)
```

or you can use the EM_GETFIRSTVISIBLELINE message to determine which line is the first visible line. (Line numbers are zero-based.)

```
FirstVisibleLine = SendMessageByVal(Text1.hWnd, EM_GETFIRSTVISIBLELINE, 0, 0)
```

> **NOTE** All the examples shown in this appendix are available on the companion CD. To make the code more easily reusable, I've encapsulated all the examples in Function and Sub routines and stored them in BAS modules. Each module contains the declaration of the API functions used, as well as the *Const* directives that define all the necessary symbolic constants. On the CD, you'll also find demonstration programs that show all the routines in action. (See Figure A-2.)

The EM_LINESCROLL message enables you to programmatically scroll the contents of a TextBox control in four directions. You must pass the number of columns to scroll horizontally in *wParam* (positive values scroll right, negative values scroll left) and the number of lines to scroll vertically in *lParam* (positive values scroll down, negative values scroll up).

```
' Scroll one line down and (approximately) 4 characters to the right.
SendMessageByVal Text1.hWnd, EM_LINESCROLL, 4, 1
```

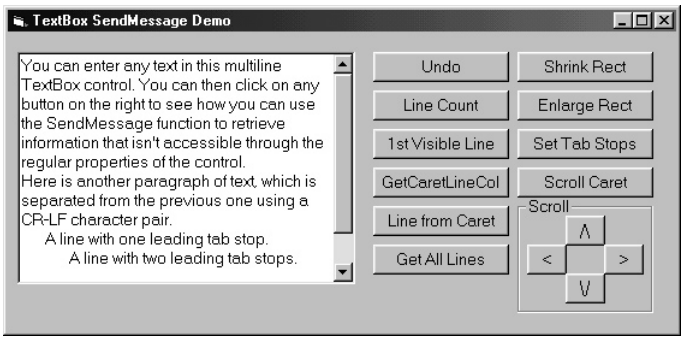


Figure A-2. *The program that demonstrates how to use the routines in the TextBox.bas module.*

Notice that the number of columns used for horizontal scrolling might not correspond to the actual number of characters scrolled if the TextBox control uses a nonfixed font. Moreover, horizontal scrolling doesn't work if the *ScrollBars* property is set to 2-Vertical. You can scroll the control's contents to ensure that the caret is visible using the EM_SCROLLCARET:

```
SendMessageByVal Text1.hWnd, EM_SCROLLCARET, 0, 0
```

One of the most annoying limitations of the standard TextBox control is that there's no way to find out how longer lines of text are split into multiple lines. Using the EM_FMTLINES message, you can ask the control to include the so-called *soft line breaks* in the string returned by its *Text* property. A soft line break is the point where the control splits a line because it's too long for the control's width. A soft line break is represented by the sequence CR-CR-LF. Hard line breaks, points at which the user has pressed the Enter key, are represented by the CR-LF sequence. When sending the EM_FMTLINES message, you must pass True in *wParam* to activate soft line breaks and False to disable them. I've prepared a routine that uses this feature to fill a String array with all the lines of text, as they appear in the control:

```
' Return an array with all the lines in the control.
' If the second optional argument is True, trailing CR-LFs are preserved.
Function GetAllLines(tb As TextBox, Optional KeepHardLineBreaks _
    As Boolean) As String()

    Dim result() As String, i As Long
    ' Activate soft line breaks.
    SendMessageByVal tb.hWnd, EM_FMTLINES, True, 0
    ' Retrieve all the lines in one operation. This operation leaves
    ' a trailing CR character for soft line breaks.
    result() = Split(tb.Text, vbCrLf)
    ' We need a loop to trim the residual CR characters. If the second
    ' argument is True, we manually add a CR-LF pair to all the lines that
    ' don't contain the residual CR char (they were hard line breaks).
```

(continued)

```
    For i = 0 To UBound(result)
        If Right$(result(i), 1) = vbCr Then
            result(i) = Left$(result(i), Len(result(i)) - 1)
        ElseIf KeepHardLineBreaks Then
            result(i) = result(i) & vbCrLf
        End If
    Next
    ' Deactivate soft line breaks.
    SendMessageByVal tb.hWnd, EM_FMTLINES, False, 0
    GetAllLines = result()
End Function
```

You can also retrieve one single line of text, using the EM_LINEINDEX message to determine where the line starts and the EM_LINELENGTH to determine its length. I've prepared a reusable routine that puts these two messages together:

```
Function GetLine(tb As TextBox, ByVal lineNum As Long) As String
    Dim charOffset As Long, lineLen As Long
    ' Retrieve the character offset of the first character of the line.
    charOffset = SendMessageByVal(tb.hWnd, EM_LINEINDEX, lineNum, 0)
    ' Now it's possible to retrieve the length of the line.
    lineLen = SendMessageByVal(tb.hWnd, EM_LINELENGTH, charOffset, 0)
    ' Extract the line text.
    GetLine = Mid$(tb.Text, charOffset + 1, lineLen)
End Function
```

The EM_LINEFROMCHAR message returns the number of the line given a character's offset; you can use this message and the EM_LINEINDEX message to determine the line and column coordinates of a character:

```
' Get the line and column coordinates of a given character.
' If charIndex is negative, it returns the coordinates of the caret.
Sub GetLineColumn(tb As TextBox, ByVal charIndex As Long, line As Long, _
    column As Long)
    ' Use the caret's offset if argument is negative.
    If charIndex < 0 Then charIndex = tb.SelStart
    ' Get the line number.
    line = SendMessageByVal(tb.hWnd, EM_LINEFROMCHAR, charIndex, 0)
    ' Get the column number by subtracting the line's start
    ' index from the character position.
    column = tb.SelStart - SendMessageByVal(tb.hWnd, EM_LINEINDEX, line, 0)
End Sub
```

Standard TextBox controls use their entire client area for editing. You can retrieve the dimension of such a formatting rectangle using the EM_GETRECT message, and you can use EM_SETRECT to modify its size as your needs dictate. In each instance, you need to include the definition of the RECT structure, which is also used by many other API functions:

```
Private Type RECT
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type
```

I've prepared two routines that encapsulate these messages:

```
' Get the formatting rectangle.
Sub GetRect(tb As TextBox, Left As Long, Top As Long, Right As Long, _
    Bottom As Long)
    Dim lpRect As RECT
    SendMessage tb.hWnd, EM_GETRECT, 0, lpRect
    Left = lpRect.Left: Top = lpRect.Top
    Right = lpRect.Right: Bottom = lpRect.Bottom
End Sub

' Set the formatting rectangle, and refresh the control.
Sub SetRect(tb As TextBox, ByVal Left As Long, ByVal Top As Long, _
    ByVal Right As Long, ByVal Bottom As Long)
    Dim lpRect As RECT
    lpRect.Left = Left: lpRect.Top = Top
    lpRect.Right = Right: lpRect.Bottom = Bottom
    SendMessage tb.hWnd, EM_SETRECT, 0, lpRect
End Sub
```

For example, see how you can shrink the formatting rectangle along its horizontal dimension:

```
Dim Left As Long, Top As Long, Right As Long, Bottom As Long
GetRect tb, Left, Top, Right, Bottom
Left = Left + 10: Right = Right - 10
SetRect tb, Left, Top, Right, Bottom
```

One last thing that you can do with multiline TextBox controls is to set their tab stop positions. By default, the tab stops in a TextBox control are set at 32 dialog units from one stop to the next, where each dialog unit is one-fourth the average character width. You can modify such default distances using the EM_SETTABSTOPS message, as follows:

```
' Set the tab stop distance to 20 dialog units
' (that is, 5 characters of average width).
SendMessage Text1.hWnd, EM_SETTABSTOPS, 1, 20
```

You can even control the position of each individual tab stop by passing this message an array of Long elements in *lParam* as well as the number of elements in the array in *wParam*. Here's an example:

```
Dim tabs(1 To 3) As Long
' Set three tab stops approximately at character positions 5, 8, and 15.
tabs(1) = 20: tabs(2) = 32: tabs(3) = 60
SendMessage Text1.hWnd, EM_SETTABSTOPS, 3, tabs(1)
```

Notice that you pass an array to an API function by passing its first element by reference.

ListBox Controls

Next to TextBox controls, ListBox and ComboBox are the intrinsic controls that benefit most from the *SendMessage* API function. In this section, I describe the messages you can send to a ListBox control. In some situations, you can send a similar message to the ComboBox control as well to get the same result, even if the numeric value of the message is different. For example, you can retrieve the height in pixels of an item in the list portion of these two controls by sending them the LB_GETITEMHEIGHT (if you're dealing with a ListBox control) or the CB_GETITEMHEIGHT (if you're dealing with a ComboBox control). I've encapsulated these two messages in a polymorphic routine that works with both types of controls. (See Figure A-3.)

```
' The result of this routine is in pixels.
Function GetItemHeight(ctrl As Control) As Long
    Dim uMsg As Long
    If TypeOf ctrl Is ListBox Then
        uMsg = LB_GETITEMHEIGHT
    ElseIf TypeOf ctrl Is ComboBox Then
        uMsg = CB_GETITEMHEIGHT
    Else
        Exit Function
    End If
    GetItemHeight = SendMessageByVal(ctrl.hwnd, uMsg, 0, 0)
End Function
```

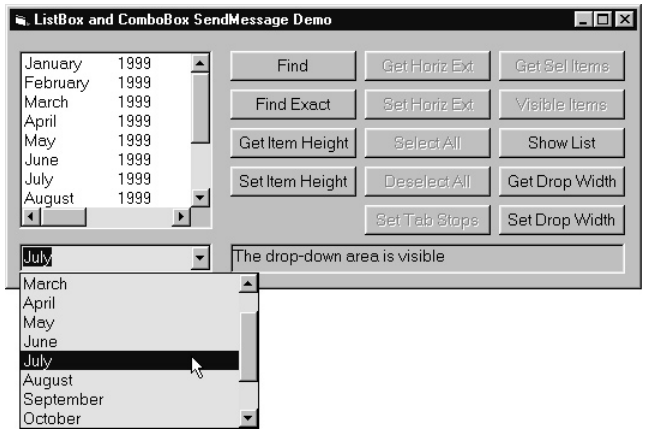


Figure A-3. *The demonstration program for using the* SendMessage *function with ListBox and ComboBox controls.*

You can also set a different height for the list items by using the LB_SETITEMHEIGHT or CB_SETITEMHEIGHT message. While the height of an item isn't valuable information in itself, it lets you evaluate the number of visible elements in a ListBox control, data that isn't exposed as a property of the Visual Basic control. You can evaluate the number of visible elements by dividing the height of the internal area of the control—also known as the *client area* of the control—by the height of each item. To retrieve the height of the client area, you need another API function, *GetClientRect*:

```
Private Declare Function GetClientRect Lib "user32" (ByVal hWnd As Long, _
    lpRect As RECT) As Long
```

This is the function that puts all the pieces together and returns the number of items in a ListBox control that are entirely visible:

```
Function VisibleItems(lb As ListBox) As Long
    Dim lpRect As RECT, itemHeight As Long
    ' Get client rectangle area.
    GetClientRect lb.hWnd, lpRect
    ' Get the height of each item.
    itemHeight = SendMessageByVal(lb.hWnd, LB_GETITEMHEIGHT, 0, 0)
    ' Do the division.
    VisibleItems = (lpRect.Bottom - lpRect.Top) \ itemHeight
End Function
```

You can use this information to determine whether the ListBox control has a companion vertical scroll bar control:

```
HasCompanionScrollBar = (Visibleitems(List1) < List1.ListCount)
```

Windows provides messages for quickly searching for a string among the items of a ListBox or ComboBox control. More precisely, there are two messages for each control, one that performs a search for a partial match—that is, the search is successful if the searched string appears at the beginning of an element in the list portion—and one that looks for exact matches. You pass the index of the element from which you start the search to *wParam* (–1 to start from the beginning), and the string being searched to *lParam* by value. The search isn't case sensitive. Here's a reusable routine that encapsulates the four messages and returns the index of the matching element or –1 if the search fails. Of course, you can reach the same result with a loop over the ListBox items, but the API approach is usually faster:

```
Function FindString(ctrl As Control, ByVal search As String, Optional _
    startIndex As Long = -1, Optional ExactMatch As Boolean) As Long
    Dim uMsg As Long
    If TypeOf ctrl Is ListBox Then
        uMsg = IIf(ExactMatch, LB_FINDSTRINGEXACT, LB_FINDSTRING)
```

(continued)

```
    ElseIf TypeOf ctrl Is ComboBox Then
        uMsg = IIf(ExactMatch, CB_FINDSTRINGEXACT, CB_FINDSTRING)
    Else
        Exit Function
    End If
    FindString = SendMessageString(ctrl.hwnd, uMsg, startIndex, search)
End Function
```

Because the search starts with the element after the *startIndex* position, you can easily create a loop that prints all the matching elements:

```
' Print all the elements that begin with the "J" character.
index = -1
Do
    index = FindString(List1, "J", index, False)
    If index = -1 Then Exit Do
    Print List1.List(index)
Loop
```

A ListBox control can display a horizontal scroll bar if its contents are wider than its client areas, but this is another capability that isn't exposed by the Visual Basic control. To make the horizontal scroll bar appear, you must tell the control that it contains elements that are wider than its client area. (See Figure A-3.) You do this using the LB_SETHORIZONTALEXTENT message, which expects a width in pixels in the *wParam* argument:

```
' Inform the ListBox control that its contents are 400 pixels wide.
' If the control is narrower, a horizontal scroll bar will appear.
SendMessageByVal List1.hwnd, LB_SETHORIZONTALEXTENT, 400, 0
```

You can add a lot of versatility to standard ListBox controls by setting the positions of their tab stops. The technique is similar to the one used for TextBox controls. If you add to that the ability to display a horizontal scroll bar, you see that the ListBox control becomes a cheap means for displaying tables—you don't have to resort to external ActiveX controls. All you have to do is set the tab stop position to a suitable distance and then add lines of tab-delimited elements, as in the following code:

```
' Create a 3-column table using a ListBox.
' The three columns hold 5, 20, and 25 characters of average width.
Dim tabs(1 To 2) As Long
tabs(1) = 20: tabs(2) = 100
SendMessage List1.hWnd, LB_SETTABSTOPS, 2, tabs(1)
' Add a horizontal scroll bar, if necessary.
SendMessageByVal List1.hwnd, LB_SETHORIZONTALEXTENT, 400, 0
List1.AddItem "1" & vbTab & "John" & vbTab & "Smith"
List1.AddItem "2" & vbTab & "Robert" & vbTab & "Doe"
```

You can learn how to use a few other ListBox messages by browsing the source code of the demonstration program provided on the companion CD.

ComboBox Controls

As I explained in the previous section, ComboBox and ListBox controls supports some common messages, even though the names and the values of the corresponding symbolic constants are different. For example, you can read and modify the height of items in the list portion using the CB_GETITEMHEIGHT and CB_SETITEMHEIGHT messages, and you can search items using the CB_FINDSTRINGEXACT and CB_FINDSTRING messages.

But the ComboBox control also supports other interesting messages. For example, you can programmatically open and close the list portion of a drop-down ComboBox control using the CB_SHOWDROPDOWN message:

```
' Open the list portion.
SendMessageByVal Combo1.hWnd, CB_SHOWDROPDOWN, True, 0
' Then close it.
SendMessageByVal Combo1.hWnd, CB_SHOWDROPDOWN, False, 0
```

and you can retrieve the current visibility state of the list portion using the CB_GETDROPPEDSTATE message:

```
If SendMessageByVal(Combo1.hWnd, CB_GETDROPPEDSTATE, 0, 0) Then
    ' The list portion is visible.
End If
```

One of the most useful messages for ComboBox controls is CB_SETDROPPEDWIDTH, which lets you set the width of the ComboBox drop-down list although values less than the control's width are ignored:

```
' Make the drop-down list 300 pixels wide.
SendMessageByVal cb.hwnd, CB_SETDROPPEDWIDTH, 300, 0
```

(See Figure A-3 for an example of a ComboBox whose drop-down list is wider than usual.)

Finally, you can use the CB_LIMITTEXT message to set a maximum number of characters for the control; this is similar to the *MaxLength* property for TextBox controls, which is missing in ComboBox controls:

```
' Set the maximum length of text in a ComboBox control to 20 characters.
SendMessageByVal Combo1.hWnd, CB_LIMITTEXT, 20, 0
```

SYSTEM FUNCTIONS

Many internal Windows values and parameters are beyond Visual Basic's capabilities, but they're just an API function call away. In this section, I show how you can retrieve some important system settings and how you can augment Visual Basic support for the mouse and the keyboard.

Windows Directories and Versions

Even though Visual Basic hides most of the complexities of the operating system, as well as the differences among the many Windows versions around, sometimes you must distinguish one from another—for example, to account for minor differences between Windows 9x and Windows NT. You can do this by examining the higher-order bit of the Long value returned by the *GetVersion* API function:

```
Private Declare Function GetVersion Lib "kernel32" () As Long

If GetVersion() And &H80000000 Then
    MsgBox "Running under Windows 95/98"
Else
    MsgBox "Running under Windows NT"
End If
```

If you need to determine the actual Windows version, you need the *GetVersionEx* API function, which returns information about the running operating system in a UDT:

```
Type OSVERSIONINFO
    dwOSVersionInfoSize As Long
    dwMajorVersion As Long
    dwMinorVersion As Long
    dwBuildNumber As Long
    dwPlatformId As Long
    szCSDVersion As String * 128
End Type

Private Declare Function GetVersionEx Lib "kernel32" Alias _
    "GetVersionExA" (lpVersionInformation As OSVERSIONINFO) As Long

Dim os As OSVERSIONINFO, ver As String
' The function expects the UDT size in the UDT's first element.
os.dwOSVersionInfoSize = Len(os)
GetVersionEx os
ver = os.dwMajorVersion & "." & Right$("0" & Format$(os.dwMinorVersion), 2)
Print "Windows Version = " & ver
Print "Windows Build Number = " & os.dwBuildNumber
```

Windows 95 returns a version number 4.00, and Windows 98 returns version 4.10. (See Figure A-4.) You can use the build number to identify different service packs.

All tips and tricks collections show how you can retrieve the path to the main Windows and System directories, which are often useful for locating other files that might interest you. These functions are helpful for another reason as well: They show you how to receive strings from an API function. In general, no API function directly

returns a string; instead, all the functions that return a string value to the calling program require that you create a receiving string buffer—typically, a string filled with spaces or null characters—and you pass it to the routine. Most of the time, you must pass the buffer's length in another argument so that the API function doesn't accidentally write in the buffer more characters than allowed. For example, this is the declaration of the *GetWindowsDirectory* API function:

```
Private Declare Function GetWindowsDirectory Lib "kernel32" Alias _
    "GetWindowsDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
```

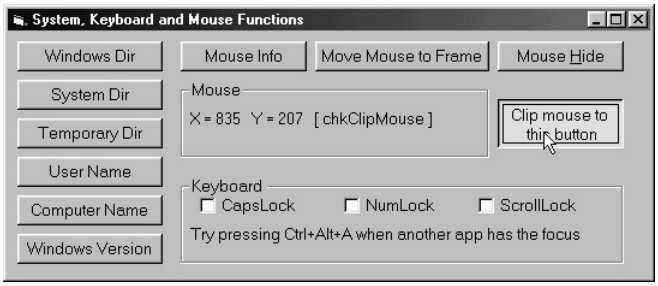


Figure A-4. *The sample program demonstrates several system, keyboard, and mouse API functions.*

You use this function by allocating a large-enough buffer, and then you pass it to the function. The return value of the function is the actual number of characters in the result string, and you can use this value to trim off characters in excess:

```
Dim buffer As String, length As Integer
buffer = Space$(512)
length = GetWindowsDirectory(buffer, Len(buffer))
Print "Windows Directory = " & Left$(buffer, length)
```

You can use the same method to determine the path of the Windows\System directory, using the *GetSystemDirectory* API function:

```
Private Declare Function GetSystemDirectory Lib "kernel32" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long

Dim buffer As String, length As Integer
buffer = Space$(512)
length = GetSystemDirectory(buffer, Len(buffer))
Print "System Directory = " & Left$(buffer, length)
```

The *GetTempPath* API function uses a similar syntax—although the order of arguments is reversed—and returns a valid directory name for storing temporary files, including a trailing backslash character (such as C:\WINDOWS\TEMP\):

```
Private Declare Function GetTempPath Lib "kernel32" Alias "GetTempPathA" _
    (ByVal nBufferLength As Long, ByVal lpBuffer As String) As Long

Dim buffer As String, length As Integer
buffer = Space$(512)
length = GetTempPath (Len(buffer), buffer)
Print "Temporary Directory = " & Left$(buffer, length)
```

The *GetUserName* function returns the name of the user currently logged in. At first glance, this function appears to use the same syntax as the functions I've just described. The documentation reveals, however, that it doesn't return the length of the result but just a zero value to indicate a failure or 1 to indicate the success of the operation. In this situation, you must extract the result from the buffer by searching for the Null character that all API functions append to result strings:

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
    "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long

Dim buffer As String * 512, length As Long
If GetUserName buffer, Len(buffer) Then
    ' Search the trailing Null character.
    length = InStr(buffer, vbNullChar) - 1
    Print "User Name = " & Left$(buffer, length)
Else
    Print "GetUserName function failed"
End If
```

The *GetComputerName* API function, which retrieves the name of the computer that's executing the program, uses yet another method: You must pass the length of the buffer in a *ByRef* argument. On exit from the function, this argument holds the length of the result:

```
Private Declare Function GetComputerName Lib "kernel32" Alias _
    "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long) As Long

Dim buffer As String * 512, length As Long
length = Len(buffer)
If GetComputerName(buffer, length) Then
    ' Returns nonzero if successful, and modifies the length argument
    MsgBox "Computer Name = " & Left$(buffer, length)
End If
```

The Keyboard

Visual Basic's keyboard events let you know exactly which keys are pressed and when. At times, however, it's useful to determine whether a given key is pressed even when you're not inside a keyboard event procedure. The pure Visual Basic solution

is to store the value of the pressed key in a module-level or global variable, but it's a solution that negatively impacts the reusability of the code. Fortunately, you can easily retrieve the current state of a given key using the *GetAsyncKeyState* function:

```
Private Declare Function GetAsyncKeyState Lib "user32" _
    (ByVal vKey As Long) As Integer
```

This function accepts a virtual key code and returns an Integer value whose high-order bit is set if the corresponding key is pressed. You can use all the Visual Basic vbKey*xxxx* symbolic constants as arguments to this function. For example, you can determine whether any of the shift keys is being pressed using this code:

```
Dim msg As String
If GetAsyncKeyState(vbKeyShift) And &H8000 Then msg = msg & "SHIFT "
If GetAsyncKeyState(vbKeyControl) And &H8000 Then msg = msg & "CTRL "
If GetAsyncKeyState(vbKeyMenu) And &H8000 Then msg = msg & "ALT "
' lblKeyboard is a Label control that displays the shift key states.
lblKeyboard.Caption = msg
```

An interesting characteristic of the *GetAsynchKeyState* function is that it works even if the application doesn't have the input focus. This capability lets you build a Visual Basic program that reacts to hot keys even if users press them while they're working with another application. To use this API function to trap hot keys, you need to add some code into a Timer control's *Timer* event procedure and set the Timer's *Interval* property to a small-enough value—for example, 200 milliseconds:

```
' Detect the Ctrl+Alt+A key combination.
Private Sub Timer1_Timer()
    If GetAsyncKeyState(vbKeyA) And &H8000 Then
        If GetAsyncKeyState(vbKeyControl) And &H8000 Then
            If GetAsyncKeyState(vbKeyMenu) And &H8000 Then
                ' Process the Ctrl+Alt+A hot key here.
            End If
        End If
    End If
End Sub
```

You can streamline your code by taking advantage of the following reusable routine, which can test the state of up to three keys:

```
Function KeysPressed(KeyCode1 As KeyCodeConstants, Optional KeyCode2 As _
    KeyCodeConstants, Optional KeyCode3 As KeyCodeConstants) As Boolean
    If GetAsyncKeyState(KeyCode1) >= 0 Then Exit Function
    If KeyCode2 = 0 Then KeysPressed = True: Exit Function
    If GetAsyncKeyState(KeyCode2) >= 0 Then Exit Function
    If KeyCode3 = 0 Then KeysPressed = True: Exit Function
    If GetAsyncKeyState(KeyCode3) >= 0 Then Exit Function
    KeysPressed = True
End Function
```

The three arguments are declared as KeyCodeConstant (an enumerated type defined in the Visual Basic runtime library) so that IntelliSense automatically helps you write the code for this function. See how you can rewrite the previous example that traps the Ctrl+Alt+A hot key:

```
If KeysPressed(vbKeyA, vbKeyMenu, vbKeyControl) Then
    ' Process the Ctrl+Alt+A hot key here.
End If
```

You can also modify the current state of a key, say, to programmatically change the state of the CapsLock, NumLock, and ScrollLock keys. For an example of this technique, see the "Toggling the State of Lock Keys" section in Chapter 10.

The Mouse

The support Visual Basic offers to mouse programming is defective in a few areas. As is true for the keyboard and its event procedures, you can derive a few bits of information about the mouse's position and the state of its buttons only inside a *MouseDown*, *MouseUp*, or *MouseMove* event procedure, which makes the creation of reusable routines in BAS modules a difficult task. Even more annoying, mouse events are raised only for the control under the mouse cursor, which forces you to write a lot of code just to find out where the mouse is in any given moment. Fortunately, querying the mouse through an API function is really simple.

To begin with, you don't need a special function to retrieve the state of mouse buttons because you can use the *GetAsyncKeyState* function with the special vbKeyLButton, vbKeyRButton, and vbKeyMButton symbolic constants. Here's a routine that returns the current state of mouse buttons in the same bit-coded format as the *Button* parameter received by *Mouse*xxxx event procedures:

```
Function MouseButton() As Integer
    If GetAsyncKeyState(vbKeyLButton) < 0 Then
        MouseButton = 1
    End If
    If GetAsyncKeyState(vbKeyRButton) < 0 Then
        MouseButton = MouseButton Or 2
    End If
    If GetAsyncKeyState(vbKeyMButton) < 0 Then
        MouseButton = MouseButton Or 4
    End If
End Function
```

The Windows API includes a function for reading the position of the mouse cursor:

```
Private Type POINTAPI
    X As Long
    Y As Long
End Type
```

```
Private Declare Function GetCursorPos Lib "user32" (lpPoint As POINTAPI) _
    As Long
```

In both cases, the coordinates are in pixels and relative to the screen:

```
' Display current mouse screen coordinates in pixels using a Label control.
Dim lpPoint As POINTAPI
GetCursorPos lpPoint
lblMouseState = "X = " & lpPoint.X & "   Y = " & lpPoint.Y
```

To convert screen coordinates to a pair of coordinates relative to the client area of a window—that is, the area of a window inside its border—you can use the *ScreenToClient* API function:

```
Private Declare Function ScreenToClient Lib "user32" (ByVal hWnd As Long, _
    lpPoint As POINTAPI) As Long

' Display mouse screen coordinates relative to current form.
Dim lpPoint As POINTAPI
GetCursorPos lpPoint
ScreenToClient Me.hWnd, lpPoint
lblMouseState = "X = " & lpPoint.X & "   Y = " & lpPoint.Y
```

The *SetCursorPos* API function lets you move the mouse cursor anywhere on the screen, something that you can't do with standard Visual Basic code:

```
Private Declare Function SetCursorPos Lib "user32" (ByVal X As Long, _
    ByVal Y As Long) As Long
```

When you use this function, you often need to convert from client coordinates to screen coordinates, which you do with the *ClientToScreen* API function. The following code snippet moves the mouse cursor to the center of a push button:

```
Private Declare Function ClientToScreen Lib "user32" (ByVal hWnd As Long, _
    lpPoint As POINTAPI) As Long

' Get the coordinates (in pixels) of the center of the Command1 button.
' The coordinates are relative to the button's client area.
Dim lpPoint As POINTAPI
lpPoint.X = ScaleX(Command1.Width / 2, vbTwips, vbPixels)
lpPoint.Y = ScaleY(Command1.Height / 2, vbTwips, vbPixels)
' Convert to screen coordinates.
ClientToScreen Command1.hWnd, lpPoint
' Move the mouse cursor to that point.
SetCursorPos lpPoint.X, lpPoint.Y
```

In some circumstances, for example, during drag-and-drop operations, you might want to prevent the user from moving the mouse outside a given region. You can achieve this behavior by setting up a rectangular *clipping area* with the *ClipCursor* API function. You'll often need to clip the mouse cursor to a given window, which

you can do by retrieving the window's client area rectangle with the *GetClientRect* API function and convert the result to screen coordinates. The following routine does everything for you:

```
Private Declare Function ClipCursor Lib "user32" (lpRect As Any) As Long

Sub ClipMouseToWindow(ByVal hWnd As Long)
    Dim lpPoint As POINTAPI, lpRect As RECT
    ' Retrieve the coordinates of the upper-left corner of the window.
    ClientToScreen hWnd, lpPoint
    ' Get the client screen rectangle.
    GetClientRect hWnd, lpRect
    ' Manually convert the rectangle to screen coordinates.
    lpRect.Left = lpRect.Left + lpPoint.X
    lpRect.Top = lpRect.Top + lpPoint.Y
    lpRect.Right = lpRect.Right + lpPoint.X
    lpRect.Bottom = lpRect.Bottom + lpPoint.Y
    ' Enforce the clipping.
    ClipCursor lpRect
End Sub
```

Here's an example that uses the previous routine and then cancels the clipping effect:

```
' Clip the mouse cursor to the current form's client area.
ClipMouseToWindow Me.hWnd
...
' When you don't need the clipping any longer. (Don't forget this!)
ClipCursor ByVal 0&
```

(Remember that a window automatically loses the mouse capture if it executes a *MsgBox* or *InputBox* statement.) Windows normally sends mouse messages to the window under the cursor. The only exception to this rule occurs when the user presses a mouse button on a window and then drags the mouse cursor outside it. In this situation, the window continues to receive mouse messages until the button is released. But sometimes it's convenient to receive mouse notifications even when the mouse is outside the window's boundaries.

Consider the following situation: You want to provide the user with a visual clue when the mouse cursor enters the area of a control—for example, by changing the control's background color. You can achieve this effect simply by changing the control's *BackColor* property in its *MouseMove* event because this event fires as soon as the mouse cursor hovers over the control. Unluckily, Visual Basic doesn't fire an event in a control when the mouse cursor exits its client area, so you don't know when to restore the original background color. Using pure Visual Basic, you're forced to write code inside the *MouseMove* events of the forms and of all the other controls on the form's surface, or you must have a Timer that periodically monitors where the mouse is. By no means is this an elegant or efficient solution.

A better approach would be to capture the mouse when the cursor enters the control's client area, using the *SetCapture* API function. When a form or a control captures the mouse, it receives mouse messages until the user clicks outside the form or the control or until the mouse capture is explicitly relinquished through a *ReleaseCapture* API function. This technique permits you to solve the problem by writing code in one single procedure:

```
' Add these declarations to a BAS module.
Private Declare Function SetCapture Lib "user32" (ByVal hWnd As Long) _
    As Long
Private Declare Function ReleaseCapture Lib "user32" () As Long
Private Declare Function GetCapture Lib "user32" () As Long

' Change the BackColor of Frame1 control to yellow when the mouse enters
' the control's client area, and restore it when the mouse leaves it.
Private Sub Frame1_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    ' Set the mouse capture unless the control already has it.
    ' (The GetCapture API function returns the handle of the window that
    ' has the mouse capture.)
    If GetCapture <> Frame1.hWnd Then
        SetCapture Frame1.hWnd
        Frame1.BackColor = vbYellow
    ElseIf X < 0 Or Y < 0 Or X > Frame1.Width Or Y > Frame1.Height Then
        ' If the mouse cursor is outside the Frame's client area, release
        ' the mouse capture and restore the BackColor property.
        ReleaseCapture
        Frame1.BackColor = vbButtonFace
    End If
End Sub
```

You can see this technique in action in the demonstration program shown in Figure A-4. Anytime the user moves the mouse onto or away from the topmost Frame control, the control's background color changes.

The *WindowsFromPoint* API function often comes in handy when you're working with the mouse because it returns the handle of the window at given screen coordinates:

```
Private Declare Function WindowFromPointAPI Lib "user32" Alias _
    "WindowFromPoint" (ByVal xPoint As Long, ByVal yPoint As Long) As Long
```

This routine returns the handle of the window under the mouse cursor:

```
Function WindowFromMouse() As Long
    Dim lpPoint As POINTAPI
    GetCursorPos lpPoint
    WindowFromMouse = WindowFromPoint(lpPoint.X, lpPoint.Y)
End Function
```

For example, you can quickly determine from within a form module which control is under the mouse cursor using the following approach:

```
Dim handle As Long, ctrl As Control
On Error Resume Next
handle = WindowFromMouse()
For Each ctrl In Me.Controls
    If ctrl.hWnd <> handle Then
        ' Not on this control, or hWnd property isn't supported.
    Else
        ' For simplicity's sake, this routine doesn't account for elements
        ' of control arrays.
        Print "Mouse is over control " & ctrl.Name
        Exit For
    End If
Next
```

For more information, see the source code of the demonstration application on the companion CD.

THE WINDOWS REGISTRY

The Windows Registry is the area where the operating system and most applications store their configuration values. You must be able to read as well as to write data into the Registry in order to build flexible applications that adapt themselves to their environment.

Visual Basic Built-In Functions

Unfortunately, the support for the Registry offered by Visual Basic leaves much to be desired and is limited to the following four commands and functions:

```
' Save a value.
SaveSetting AppName, Section, Key, Setting
' Read a value. (The Default argument is optional.)
value = GetSetting(AppName, Section, Key, Default)
' Return a list of settings and their values.
values = GetAllSettings(AppName, Section)
' Delete a value. (Section and Key arguments are optional.)
DeleteSetting AppName, Section, Key
```

These four commands can't read and write to an arbitrary area in the Registry but are limited to the *HKEY_CURRENT_USER\Software\VB and VBA Program Settings* subtree of the Registry. For example, you can use the *SaveSetting* function to store the initial position and size of the main form in the MyInvoicePrg application:

```
SaveSetting "MyInvoicePrg", "frmMain", "Left", frmMain.Left
SaveSetting "MyInvoicePrg", "frmMain", "Top", frmMain.Top
```

```
SaveSetting "MyInvoicePrg", "frmMain", "Width", frmMain.Width
SaveSetting "MyInvoicePrg", "frmMain", "Height", frmMain.Height
```

You can see the result of this sequence of statements in Figure A-5.

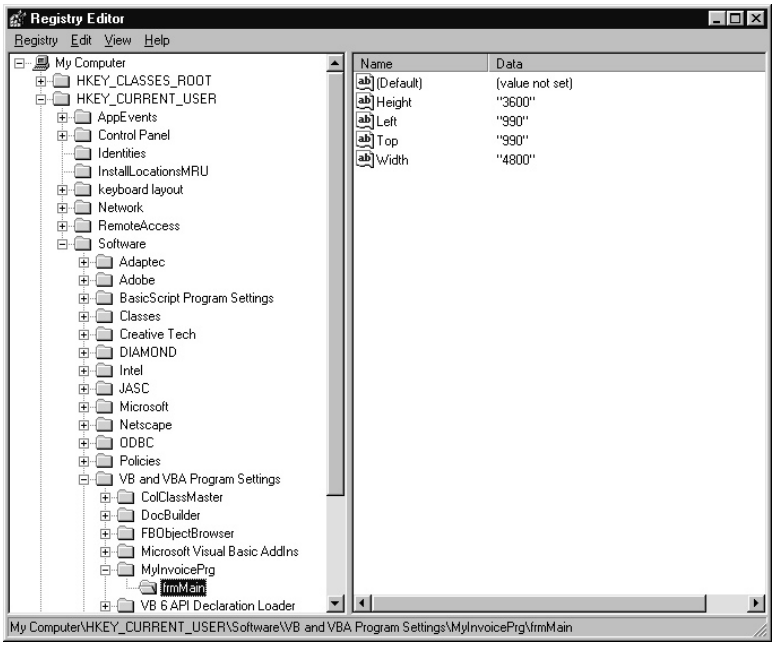


Figure A-5. *All Visual Basic Registry functions read and write values in the* HKEY_CURRENT_USER\Software\VB and VBA Program Settings *subtree.*

You can then read back these settings using the *GetSetting* function:

```
' Use the Move method to avoid multiple Resize and Paint events.
frmMain.Move GetSetting("MyInvoicePrg", "frmMain", "Left", "1000"), _
    GetSetting("MyInvoicePrg", "frmMain", "Top", "800"), _
    GetSetting("MyInvoicePrg", "frmMain", "Width", "5000"), _
    GetSetting("MyInvoicePrg", "frmMain", "Height", "4000")
```

If the specified key doesn't exist, the *GetSetting* function either returns the values passed to the *Default* argument or it returns an empty string if that argument is omitted. *GetAllSettings* returns a two-dimensional array, which contains all the keys and values under a given section:

```
Dim values As Variant, i As Long
values = GetAllSettings("MyInvoicePrg", "frmMain")
' Each row holds two items, the key name and the key value.
For i = 0 To UBound(settings)
    Print "Key =" & values(i, 0) & "  Value = " & values(i, 1)
Next
```

The last function of the group, *DeleteSetting*, can delete an individual key, or it can delete all the keys under a given section if you omit its last argument:

```
' Delete the "Left" key for the frmMain form.
DeleteSetting "MyInvoicePrg", "frmMain", "Left"
' Delete all the settings for the frmMain form.
DeleteSetting "MyInvoicePrg", "frmMain"
```

The demonstration program shown in Figure A-6 demonstrates how you can use the Visual Basic built-in Registry functions to save and to restore form settings.

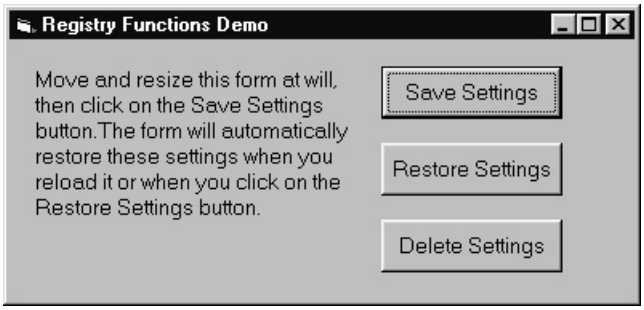


Figure A-6. *The demonstration program contains reusable routines for saving and restoring form settings to the Registry.*

The API Functions

While the Visual Basic built-in functions are barely versatile enough for saving and restoring program configuration values, they entirely lack the functionality for accessing any region of the Registry, which you must have to read some important settings of the operating system. Luckily, the Windows API contains all the functions you need to perform this task.

> **WARNING** You must be very careful when you play with the Registry in this way because you might corrupt the installation of other applications or the operating system itself, and you might even be forced to reinstall them. But in general, you can't do much harm if you simply read values in the Registry and don't write to it. To reduce risks, however, you might want to back up your system Registry so that you have a copy to restore if something goes wrong.

Predefined keys

Before starting to play with API functions, you must have a broad idea of how the Registry is arranged. The system Registry is a hierarchical structure that consists of keys, subkeys, and values. More precisely, the Registry has a number of predefined top-level keys, which I've summarized in Table A-1.

Key	*Value*	*Description*
HKEY_CLASSES_ROOT	&H80000000	The subtree that contains all the information about COM components installed on the machine. (It's actually a subtree of the HKEY _LOCAL_MACHINE key but also appears as a top-level key.)
HKEY_CURRENT_USER	&H80000001	The subtree that contains the preferences for the current user. (It's actually a subtree of the HKEY_USERS key but also appears as a top-level key.)
HKEY_LOCAL_MACHINE	&H80000002	The subtree that contains information about the physical configuration of the computer, including installed hardware and software.
HKEY_USERS	&H80000003	The subtree that contains the default user configuration and also contains information about the current user.
HKEY_PERFORMANCE_DATA	&H80000004	The subtree that collects performance data; data is actually stored outside the Registry, but appears to be part of it. (It's available only in Windows NT.)
HKEY_CURRENT_CONFIG	&H80000005	The subtree that contains data about the current configuration. (It corresponds to a subtree of the HKEY_LOCAL_MACHINE key but also appears as a top-level key.)
HKEY_DYN_DATA	&H80000006	The subtree that collects performance data; this portion of the Registry is reinitialized at each reboot. (It's available only in Windows 95 and 98.)

Table A-1. *The predefined Registry keys.*

Each Registry key has a name, which is a string of up to 260 printable characters that can't include backslash characters (\) or wildcards (? and *). Names beginning with a period are reserved. Each key can contain subkeys and values. In Windows 3.1, a key could hold only one unnamed value, while 32-bit platforms allow an unlimited number of values. (But unnamed values, called *default values*, are maintained for backward compatibility.)

> **NOTE** In general, Windows 9x and Windows NT differ in how they deal with the Registry. In Windows NT, you must account for additional security issues, and in general you have no guarantee that you can open an existing Registry key or value. In this section, I stayed clear of such details and focused on those functions that behave the same way for all the Windows platforms. For this reason, I've sometimes used "old" Registry functions instead of newer ones, which you recognize by the *Ex* suffix in their names, a suffix that stands for "extended."

Working with keys

Navigating the Registry is similar to exploring a directory tree: To reach a given file, you must open the directory that contains it. Likewise, you reach a Registry subkey from another open key at a higher level in the Registry hierarchy. You must open a key before reading its subkeys and its values, and to do that you must supply the handle of another open key in the Registry. After you've worked with a key, you must close it, as you do with files. The only keys that are always open and that don't need to be closed are the top-level keys listed in Table A-1. You open a key with the *RegOpenKeyEx* API function:

```
Declare Function RegOpenKeyEx Lib "advapi32.dll" Alias "RegOpenKeyExA" _
    (ByVal hKey As Long, ByVal lpSubKey As String, ByVal ulOptions As _
    Long, ByVal samDesired As Long, phkResult As Long) As Long
```

hKey is the handle of an open key and can be one of the values listed in Table A-1 or the handle of a key that you've opened previously. *lpSubKey* is the path from the *hKey* key to the key that you want to open. *ulOptions* is a reserved argument and must be 0. *samDesired* is the type of access you want for the key that you want to open and is a symbolic constant, such as KEY_READ, KEY_WRITE, or KEY_ALL_ACCESS. Finally, *phkResult* is a Long variable passed by reference, which receives the handle of the key opened by the function if the operation is successful. You can test the success of the open operation by looking at the return value of the *RegOpenKeyEx* function: A zero value means that the operation succeeded, and any non-zero value is an error code. This behavior is common to all the Registry API functions, so you can easily set up a function that tests the success state of any call. (See the MSDN documentation for the list of error codes.)

As I mentioned earlier, you must close any open key as soon as you don't need it any longer, which you do with the *RegCloseKey* API function. This function takes the handle of the key to be closed as its only argument, and returns 0 if the operation is successful:

```
Declare Function RegCloseKey Lib "advapi32.dll" (ByVal hKey As Long) _
    As Long
```

Frequently, the presence of a subkey is enough to store significant data in a key. For example, if the machine has a math coprocessor, Windows creates the following key:

```
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\FloatingPointProcessor
```

so you can test the presence of the coprocessor using this routine:

```
' Assumes that all symbolic constants are correctly declared elsewhere.
Function MathProcessor() As Boolean
    Dim hKey As Long, Key As String
    Key = "HARDWARE\DESCRIPTION\System\FloatingPointProcessor"
    If RegOpenKeyEx(HKEY_LOCAL_MACHINE, Key, 0, KEY_READ, hKey) = 0 Then
```

```
        ' If the open operation succeeded, the key exists.
        MathProcessor = True
        ' Important: close the key before exiting.
        RegCloseKey hKey
    End If
End Function
```

As you might expect, the Registry API includes a function for creating new keys, but its syntax is overly complex:

```
Declare Function RegCreateKeyEx Lib "advapi32.dll" Alias "RegCreateKeyExA"_
    (ByVal hKey As Long, ByVal lpSubKey As String, ByVal Reserved As Long,_
    ByVal lpClass As Long, ByVal dwOptions As Long, _
    ByVal samDesired As Long, ByVal lpSecurityAttributes As Long, _
    phkResult As Long, lpdwDisposition As Long) As Long
```

Most of the arguments have the same names and syntax as those that I've already described for the *RegOpenKeyEx* function, and I won't describe most of the new arguments because they constitute a topic too advanced for this context. You can pass a Long variable to the *lpdwDisposition* argument, and when the function returns you can test the contents in this variable. The value REG_CREATED_NEW_KEY (1) means that the key didn't exist and has been created and opened by this function, whereas the value REG_OPENED_EXISTING_KEY (2) means that the key already existed and the function just opened it without altering the Registry in any way. To reduce the confusion, I use the following routine, which creates a key if necessary and returns True if the key already existed:

```
Function CreateRegistryKey(ByVal hKey As Long, ByVal KeyName As String) _
    As Boolean
    Dim handle As Long, disp As Long
    If RegCreateKeyEx(hKey, KeyName, 0, 0, 0, 0, 0, handle, disp) Then
        Err.Raise 1001, , "Unable to create the Registry key"
    Else
        ' Return True if the key already existed.
        If disp = REG_OPENED_EXISTING_KEY Then CreateRegistryKey = True
        ' Close the key.
        RegCloseKey handle
    End If
End Function
```

The following code snippet shows how you can use the *CreateRegistryKey* function to create a key with the name of your company under the key HKEY_CURRENT_USER\Software, which contains another key with the name of your application. This is the approach followed by most commercial applications, including all those by Microsoft and other leading software companies:

```
CreateRegistryKey HKEY_CURRENT_USER, "Software\YourCompany"
CreateRegistryKey HKEY_CURRENT_USER, "Software\YourCompany\YourApplication"
```

> **NOTE** The *CreateRegistryKey* function, like all other Registry routines provided on the companion CD, always closes a key before exiting. This approach makes them "safe," but it also imposes a slight performance penalty because each call opens and closes a key that you might have to reopen immediately afterwards, as in the preceding example. You can't always have it all.

Finally, you can delete a key from the Registry, using the *RegDeleteKey* API function:

```
Declare Function RegDeleteKey Lib "advapi32.dll" Alias "RegDeleteKeyA" _
    (ByVal hKey As Long, ByVal lpSubKey As String) As Long
```

Under Windows 95 and 98, this function deletes a key and all its subkeys, whereas under Windows NT you get an error if the key being deleted contains other keys. For this reason, you should manually delete all the subkeys first:

```
' Delete the keys created in the previous example.
RegDeleteKey HKEY_CURRENT_USER, "Software\YourCompany\YourApplication"
RegDeleteKey HKEY_CURRENT_USER, "Software\YourCompany"
```

Working with values

In many cases, a Registry key contains one or more values, so you must learn how to read these values. To do so, you need the *RegQueryValueEx* API function:

```
Declare Function RegQueryValueEx Lib "advapi32.dll" Alias _
    "RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
    ByVal lpReserved As Long, lpType As Long, lpData As Any, _
    lpcbData As Long) As Long
```

hKey is the handle of the open key that contains the value. *lpValueName* is the name of the value you want to read. (Use an empty string for the default value.) *lpReserved* must be zero. *lpType* is the type of the key. *lpData* is a pointer to a buffer that will receive the data. *lpcbData* is a Long variable passed by reference; on entry it has to contain the size in bytes of the buffer, and on exit it contains the number of bytes actually stored in the buffer. Most Registry values you'll want to read are of type REG_DWORD (a Long value), REG_SZ (a null-terminated string), or REG_BINARY (array of bytes).

The Visual Basic environment stores some of its configuration settings as values under the following key:

```
HKEY_CURRENT_USER\Software\Microsoft\VBA\Microsoft Visual Basic
```

You can read the FontHeight value to retrieve the size of the font used for the code editor, whereas the FontFace value holds the name of the font. Because the former value is a Long number and the latter is a string, you need two different coding techniques for them. Reading a Long value is simpler because you just pass a Long vari-

able by reference to *lpData* and pass its length in bytes (which is 4 bytes) in *lpcbData*. To retrieve a string value, on the other hand, you must prepare a buffer and pass it by value, and when the function returns you must strip the excess characters:

```
Dim KeyName As String, handle As Long
Dim FontHeight As Long, FontFace As String, FontFaceLen As Long

KeyName = "Software\Microsoft\VBA\Microsoft Visual Basic"
If RegOpenKeyEx(HKEY_CURRENT_USER, KeyName, 0, KEY_READ, handle) Then
    MsgBox "Unable to open the specified Registry key"
Else
    ' Read the "FontHeight" value.
    If RegQueryValueEx(handle, "FontHeight", 0, REG_DWORD, FontHeight, 4) _
        = 0 Then
        Print "Face Height = " & FontHeight
    End If

    ' Read the "FontFace" value.
    FontFaceLen = 128                   ' Prepare the receiving buffer.
    FontFace = Space$(FontFaceLen)
    ' Notice that FontFace is passed using ByVal.
    If RegQueryValueEx(handle, "FontFace", 0, REG_SZ, ByVal FontFace, _
        FontFaceLen) = 0 Then
        ' Trim excess characters, including the trailing Null char.
        FontFace = Left$(FontFace, FontFaceLen - 1)
        Print "Face Name = " & FontFace
    End If
    ' Close the Registry key.
    RegCloseKey handle
End If
```

Because you need to read Registry values often, I've prepared a reusable function that performs all the necessary operations and returns the value in a Variant. You can also specify a default value, which you can use if the specified key or value doesn't exist. This tactic is similar to what you do with the Visual Basic intrinsic *GetSetting* function.

```
Function GetRegistryValue(ByVal hKey As Long, ByVal KeyName As String, _
    ByVal ValueName As String, ByVal KeyType As Integer, _
    Optional DefaultValue As Variant = Empty) As Variant

    Dim handle As Long, resLong As Long
    Dim resString As String, length As Long
    Dim resBinary() As Byte
    ' Prepare the default result.
    GetRegistryValue = DefaultValue
    ' Open the key, exit if not found.
    If RegOpenKeyEx(hKey, KeyName, 0, KEY_READ, handle) Then Exit Function
```

(continued)

```
    Select Case KeyType
        Case REG_DWORD
            ' Read the value, use the default if not found.
            If RegQueryValueEx(handle, ValueName, 0, REG_DWORD, _
                resLong, 4) = 0 Then
                GetRegistryValue = resLong
            End If
        Case REG_SZ
            length = 1024: resString = Space$(length)
            If RegQueryValueEx(handle, ValueName, 0, REG_SZ, _
                ByVal resString, length) = 0 Then
                ' If value is found, trim excess characters.
                GetRegistryValue = Left$(resString, length - 1)
            End If
        Case REG_BINARY
            length = 4096
            ReDim resBinary(length - 1) As Byte
            If RegQueryValueEx(handle, ValueName, 0, REG_BINARY, _
                resBinary(0), length) = 0 Then
                ReDim Preserve resBinary(length - 1) As Byte
                GetRegistryValue = resBinary()
            End If
        Case Else
            Err.Raise 1001, , "Unsupported value type"
    End Select
    RegCloseKey handle
End Function
```

To create a new Registry value or to modify the data of an existing value, you use the *RegSetValueEx* API function:

```
Declare Function RegSetValueEx Lib "advapi32.dll" Alias "RegSetValueExA" _
    (ByVal hKey As Long, ByVal lpValueName As String, _
    ByVal Reserved As Long, ByVal dwType As Long, lpData As Any, _
    ByVal cbData As Long) As Long
```

Let's see how we can add a LastLogin value in the key HKEY_CURRENT_USER\Software\YourCompany\YourApplication, that we created in the previous section:

```
Dim handle As Long, strValue As String
' Open the key, check whether any error occurred.
If RegOpenKeyEx(HKEY_CURRENT_USER, "Software\YourCompany\YourApplication",_
    0, KEY_WRITE, handle) Then
    MsgBox "Unable to open the key."
Else
    ' We want to add a "LastLogin" value of type string.
    strValue = FormatDateTime(Now)
    ' Strings must be passed using ByVal.
    RegSetValueEx handle, "LastLogin", 0, REG_SZ, ByVal strValue, _
```

```
        Len(strValue)
    ' Don't forget to close the key.
    RegCloseKey handle
End If
```

On the companion CD, you'll find the source code of the *SetRegistryValue* function, which automatically uses the correct syntax according to the type of value you're creating. Finally, by using the *RegDeleteValue* API function, you can delete a value under a key that you opened previously:

```
Declare Function RegDeleteValue Lib "advapi32.dll" Alias "RegDeleteValueA"_
    (ByVal hKey As Long, ByVal lpValueName As String) As Long
```

Enumerating keys and values

When you're exploring the Registry, you often need to enumerate all the keys or all the values under a key. The function you use to enumerate keys is *RegEnumKey*:

```
Private Declare Function RegEnumKey Lib "advapi32.dll" _
    Alias "RegEnumKeyA" (ByVal hKey As Long, ByVal dwIndex As Long, _
    ByVal lpName As String, ByVal cbName As Long) As Long
```

You must pass the handle of an open Registry key in the *hKey* argument, and then you repeatedly call this function, passing increasing index values in *dwIndex*. The *lpName* argument must be a string buffer of at least 260 characters (the maximum length for a key name), and *lpcbName* is the length of the buffer. When you exit the routine, the buffer contains a Null-terminated string, so you have to strip all the excess characters. To simplify your job, I've prepared a function that iterates on all the subkeys of a given key and returns a String array that contains the names of all the subkeys:

```
Function EnumRegistryKeys(ByVal hKey As Long, ByVal KeyName As String) _
    As String()
    Dim handle As Long, index As Long, length As Long
    ReDim result(0 To 100) As String

    ' Open the key, exit if not found.
    If Len(Keyname) Then
        If RegOpenKeyEx(hKey, KeyName, 0, KEY_READ, handle) Then
            Exit Function
        End If
        ' Subsequent functions use hKey.
        hKey = handle
    End If

    For index = 0 To 999999
        ' Make room in the array.
        If index > UBound(result) Then
            ReDim Preserve result(index + 99) As String
```

(continued)

```
            End If
            length = 260                    ' Max length for a key name.
            result(index) = Space$(length)
            If RegEnumKey(hKey, index, result(index), length) Then Exit For
            ' Trim excess characters.
            result(index) = Left$(result(index), InStr(result(index), _
                vbNullChar) - 1)
        Next

        ' Close the key if it was actually opened.
        If handle Then RegCloseKey handle
        ' Trim unused items in the array, and return the results to the caller.
        ReDim Preserve result(index - 1) As String
        EnumRegistryKeys = result()
    End Function
```

Thanks to the *EnumRegistryKey* function, it's simple to dig a lot of useful information out of the Registry. For example, see how easy it is to fill a ListBox control with the names of all the components registered on the machine under the HKEY_CLASS_ROOT key:

```
Dim keys() As String, i As Long
keys() = EnumRegistryKeys(HKEY_CLASSES_ROOT, "")
List1.Clear
For i = LBound(keys) To UBound(keys)
    List1.AddItem keys(i)
Next
```

The companion CD includes a demonstration program (shown in Figure A-7) that displays the list of installed COM components as well as their CLSIDs and the DLL or EXE file that contains each one of them. You can easily expand this first version to create your own utilities that track anomalies in the Registry. For example, you can list all the DLL and EXE files that aren't in the locations listed in the Registry. (COM raises an error when you try to instantiate such components.)

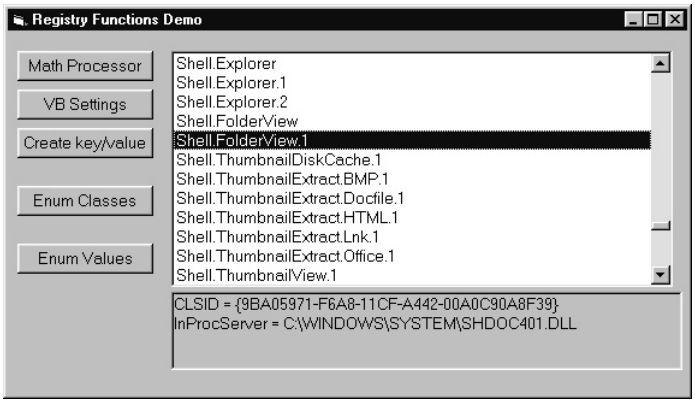


Figure A-7. *You can use Registry API routines to list all the components installed on your machine, with their CLSIDs and the locations of their executable files.*

The Windows API also exposes a function for enumerating all the values under a given open key:

```
Declare Function RegEnumValue Lib "advapi32.dll" Alias "RegEnumValueA" _
    (ByVal hKey As Long, ByVal dwIndex As Long, ByVal lpValueName As _
    String, lpcbValueName As Long, ByVal lpReserved As Long, _
    lpType As Long, lpData As Any, lpcbData As Long) As Long
```

This function returns the type of each value in the *lpType* variable and the contents of the value in *lpData*. The difficulty is that you don't know in advance what the type of the value is, and therefore you don't know the kind of variable—Long, String, or Byte array—you should pass in *lpData*. The solution to this problem is to pass a Byte array and then move the result into a Long variable using the *CopyMemory* API routine or into a String variable using the VBA *StrConv* function. On the companion CD, you'll find the complete source of the *EnumRegistryValues* routine, which encapsulates all these details and returns a two-dimensional array of Variants containing all the values' names and data. For example, you can use this routine to retrieve all the Microsoft Visual Basic configuration values:

```
Dim values() As Variant, i As Long
values() = EnumRegistryValues(HKEY_CURRENT_USER, _
    "Software\Microsoft\VBA\Microsoft Visual Basic")
For i = LBound(values, 2) To UBound(values, 2)
    ' Row 0 holds the value's name, row 1 holds its value.
    List1.AddItem values(0, i) & " = " & values(1, i)
Next
```

CALLBACK AND SUBCLASSING

As you probably remember from the "A World of Messages" section near the beginning of this appendix, Windows deals with two types of messages: control messages and notification messages. Although sending a control message is just a matter of using the *SendMessage* API function, you'll see that intercepting a notification message is much more difficult and requires that you adopt an advanced programming technique known as *window subclassing*. But to understand how this technique works, you need to know what the *AddressOf* keyword does and how you can use it to set up a callback procedure.

Callback Techniques

Callback and subclassing capabilities are relatively new to Visual Basic in that they weren't possible until version 5. What made these techniques available to Visual Basic programmers was the introduction of the new *AddressOf* keyword under Visual Basic 5. This keyword can be used as a prefix for the name of a routine defined in a BAS module, and evaluates to the 32-bit address of the first statement of that routine.

System timers

To show this keyword in action, I'll show you how you can create a timer without a Timer control. Such a timer might be useful, for example, when you want to periodically execute a piece of code located in a BAS module, and you don't want to add a form to the application just to get a pulse at regular intervals. Setting up a system timer requires only a couple of API functions:

```
Declare Function SetTimer Lib "user32" (ByVal hWnd As Long, ByVal nIDEvent_
    As Long, ByVal uElapse As Long, ByVal lpTimerFunc As Long) As Long

Declare Function KillTimer Lib "user32" (ByVal hWnd As Long, _
    ByVal nIDEvent As Long) As Long
```

For our purposes, we can ignore the first two arguments to the *SetTimer* function and just pass the *uElapse* value (which corresponds to the *Interval* property of a Timer control) and the *lpTimerFunc* value (which is the address of a routine in our Visual Basic program). This routine is known as the *callback procedure* because it's meant to be called from Windows and not from the code in our application. The *SetTimer* function returns the ID of the timer being created or 0 in case of error:

```
Dim timerID As Long
' Create a timer that sends a notification every 500 milliseconds.
timerID = SetTimer(0, 0, 500, AddressOf Timer_CBK)
```

You need the return value when it's time to destroy the timer, a step that you absolutely must perform before closing the application if you don't want the program to crash:

```
' Destroy the timer created previously.
KillTimer 0, timerID
```

Let's see now how to build the *Timer_CBK* callback procedure. You derive the number and types of the arguments that Windows sends to it from the Windows SDK documentation or from MSDN:

```
Sub Timer_CBK(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal idEvent As Long, ByVal SysTime As Long)
    ' Just display the system time in a label control.
    Form1.lblTimer = SysTime
End Sub
```

In this implementation, you can safely ignore the first three parameters and concentrate on the last one, which receives the number of milliseconds elapsed since the system started. This particular callback routine doesn't return a value and is therefore implemented as a procedure; you'll see later that in most cases callback routines return values to the operating system and therefore are implemented as functions. As usual, you'll find on the companion CD a complete demonstration program that contains all the routines described in this section.

Windows enumeration

Interesting and useful examples of using callback techniques are provided by the *EnumWindows* and *EnumChildWindows* API functions, which enumerate the top-level windows and the child windows of a given window, respectively. The approach used by these functions is typical of most API functions that enumerate Windows objects. Instead of loading the list of windows in an array or another structure, these functions use a callback procedure in the main application for each window found. Inside the callback function, you can do what you want with such data, including loading it into an array or into a ListBox or TreeView control. The syntax for these functions is the following:

```
Declare Function EnumWindows Lib "user32" (ByVal lpEnumFunc As Long, _
    ByVal lParam As Long) As Long

Declare Function EnumChildWindows Lib "user32" (ByVal hWndParent As Long, _
    ByVal lpEnumFunc As Long, ByVal lParam As Long) As Long
```

hWndParent is the handle of the parent window. *lpEnumFunc* is the address of the callback function. And *lParam* is a parameter passed to the callback function; this value can be used when the same callback routine is used for different purposes in the application. The syntax of the callback function is the same for both *EnumWindows* and *EnumChildWindows*:

```
Function EnumWindows_CBK(ByVal hWnd As Long, ByVal lParam As Long) As Long
    ' Process the window's data here.
End Function
```

where *hWnd* is the handle of the window found, and *lParam* is the value passed as the last argument to the *EnumWindows* or *EnumChildWindows* function. This function returns 1 to ask the operating system to continue the enumeration or 0 to stop the enumeration.

It's easy to create a reusable procedure that builds on these API functions and returns an array with the handles of all the child windows of a given window:

```
' An array of Longs holding the handles of all child windows
Dim windows() As Long
' The number of elements in the array.
Dim windowsCount As Long

' Return an array of Longs holding the handles of all the child windows
' of a given window. If hWnd = 0, return the top-level windows.
Function ChildWindows(ByVal hWnd As Long) As Long()
    windowsCount = 0                    ' Reset the result array.
    If hWnd Then
        EnumChildWindows hWnd, AddressOf EnumWindows_CBK, 1
    Else
        EnumWindows AddressOf EnumWindows_CBK, 1
    End If
```

(continued)

```
    ' Trim uninitialized elements, and return to caller.
    ReDim Preserve windows(windowsCount) As Long
    ChildWindows = windows()
End Function

' The callback routine, common to both EnumWindows and EnumChildWindows
Function EnumWindows_CBK(ByVal hWnd As Long, ByVal lParam As Long) As Long
    If windowsCount = 0 Then
        ' Create the array at the first iteration.
        ReDim windows(100) As Long
    ElseIf windowsCount >= UBound(windows) Then
        ' Make room in the array if necessary.
        ReDim Preserve windows(windowsCount + 100) As Long
    End If
    ' Store the new item.
    windowsCount = windowsCount + 1
    windows(windowsCount) = hWnd
    ' Return 1 to continue the enumeration process.
    EnumWindows_CBK = 1
End Function
```

On the companion CD, you'll find the source code of an application—also shown in Figure A-8—that displays the hierarchy of all the windows that are currently open in the system. This is the code that loads the TreeView control with the window hierarchy. Thanks to the recursion technique, the code is surprisingly compact:

```
Private Sub Form_Load()
    ShowWindows TreeView1, 0, Nothing
End Sub

Sub ShowWindows(tvw As TreeView, ByVal hWnd As Long, ParentNode As Node)
    Dim winHandles() As Long
    Dim i As Long, Node As MSComctlLib.Node

    If ParentNode Is Nothing Then
        ' If no Parent node, let's add a "desktop" root node.
        Set ParentNode = tvw.Nodes.Add(, , , "Desktop")
    End If
    ' Retrieve all the child windows.
    winHandles() = ChildWindows(hWnd)
    For i = 1 To UBound(winHandles)
        ' Add a node for this child window--WindowDescription is a routine
        ' (not shown here) that returns a descriptive string for the window.
        Set Node = tvw.Nodes.Add(ParentNode.Index, tvwChild, , _
            WindowDescription(winHandles(i)))
        ' Recursively call this routine to show this window's children.
        ShowWindows tvw, winHandles(i), Node
    Next
End Sub
```

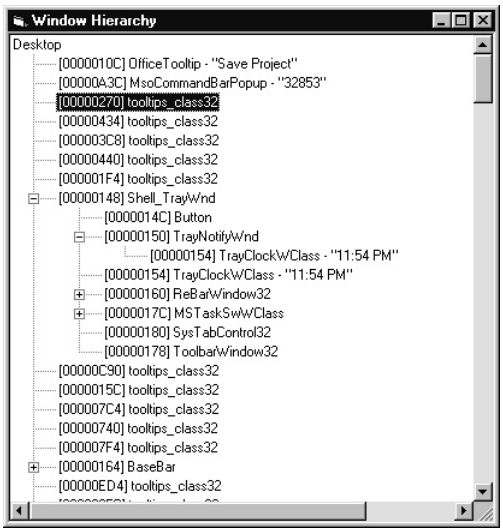


Figure A-8. *A utility to explore all the open windows in the system.*

Subclassing Techniques

Now that you know what a callback procedure is, comprehending how subclassing works will be a relatively easy job.

Basic subclassing

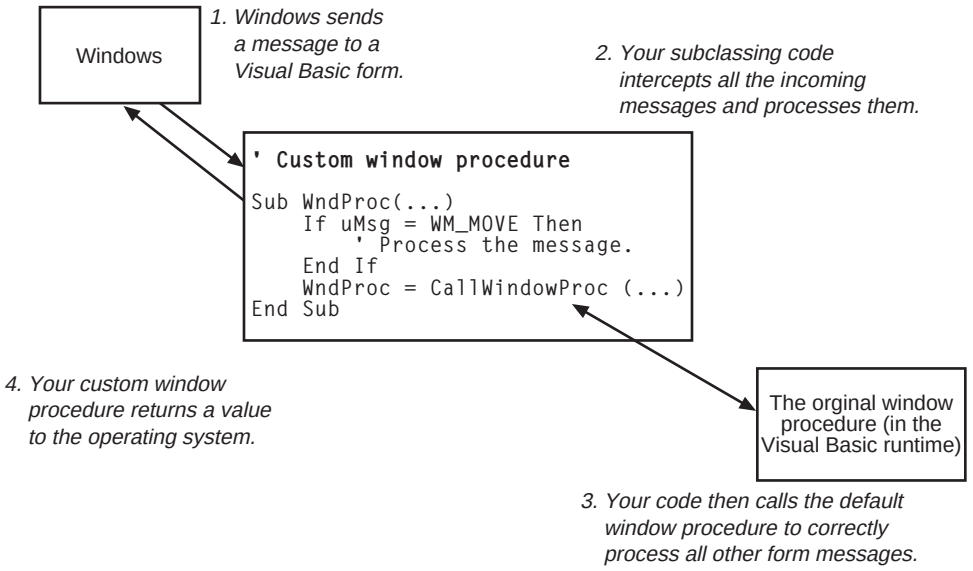
You already know that Windows communicates with applications via messages, but you don't know yet how the mechanism actually works at a lower level. Each window is associated with a *window default procedure*, which is called any time a message is sent to the window. If this procedure were written in Visual Basic, it would look like this:

```
Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    ...
End Function
```

The four parameters that a window procedure receives are exactly the arguments that you (or the operating system) pass to *SendMessage* when you send a message to a given window. The purpose of the window procedure is to process all the incoming messages and react appropriately. Each class of windows—top-level windows, MDI windows, TextBox controls, ListBox controls, and so on—behave differently because their window procedures are different.

The principle of the subclassing technique is simple: You write a custom window procedure, and you ask Windows to call your window procedure instead of the standard window procedure associated with a given window. The code in your

Visual Basic application traps all the messages sent to the window before the window itself (more precisely, its default window procedure) has a chance to process them, as I explain in the following illustration:



To substitute the standard window procedure with your customized procedure, you must use the *SetWindowLong* API function, which stores the address of the custom routine in the internal data table that is associated with each window:

```
Const GWL_WNDPROC = -4
Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" _
    (ByVal hWnd As Long, ByVal ndx As Long, ByVal newValue As Long) As Long
```

hWnd is the handle of the window. *ndx* is the index of the slot in the internal data table where you want to store the value. And *newValue* is the 32-bit value to be stored in the internal data table at the position pointed to by *nxd*. This function returns the value that was previously stored in that slot of the table; you must store such a value in a variable because you must definitely restore it before the application terminates or the subclassed window is closed. If you don't restore the address of the original window procedure, you're likely to get a GPF. In summary, this is the minimal code that subclasses a window:

```
Dim saveHWnd As Long        ' The handle of the subclassed window
Dim oldProcAddr As Long     ' The address of the original window procedure

Sub StartSubclassing(ByVal hWnd As Long)
    saveHWnd = hWnd
    oldProcAddr = SetWindowLong(hWnd, GWL_WNDPROC, AddressOf WndProc)
End Sub
```

```
Sub StopSubclassing()
    SetWindowLong saveHWnd, GWL_WNDPROC, oldProcAddr
End Sub

Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    ' Process the incoming messages here.
End Function
```

Let's focus on what the custom window procedure actually does. This procedure can't just process a few messages and forget about the others. On the contrary, it's responsible for correctly forwarding all the messages to the original window procedure; otherwise, the window wouldn't receive all the vital messages that inform it when it has to resize, close, or repaint itself. In other words, if the window procedure stops all messages from reaching the original window procedure, the application won't work as expected any longer. The API function that does the message forwarding is *CallWindowProc*:

```
Declare Function CallWindowProc Lib "user32" Alias "CallWindowProcA" _
    (ByVal lpPrevWndFunc As Long, ByVal hwnd As Long, ByVal Msg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
```

lpPrevWndFunc is the address of the original window procedure—the value that we saved in the *oldProcAddr* variable—and the other arguments are those received by the custom window procedure.

Let's see a practical example of the subclassing technique. When a top-level window—a form, in Visual Basic parlance—moves, the operating system sends it a WM_MOVE message. The Visual Basic runtime eats this message without exposing it as an event to the application's code, but you can write a custom window procedure that intercepts it before Visual Basic sees it:

```
Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    ' Send the message to the original window procedure, and then
    ' return to Windows the return value from the original procedure.
    WndProc = CallWindowProc(oldProcAddr, hWnd, uMsg, wParam, lParam)
    ' See if this is the message we're waiting for.
    If uMsg = WM_MOVE Then
        ' The window has moved.
    End If
End Function
```

I've prepared a demonstration program that uses the code described in this section to trap a few messages related to forms, such as WM_MOVE, WM_RESIZING, and WM_APPACTIVATE. (See Figure A-9.) The last message is important because it lets you determine when an application loses and regains the input focus, something that you can't easily do in pure Visual Basic code. For example, the Windows hierarchy utility shown in Figure A-8 might subclass this message to automatically refresh its contents when the user switches to another application and then goes back to the utility.

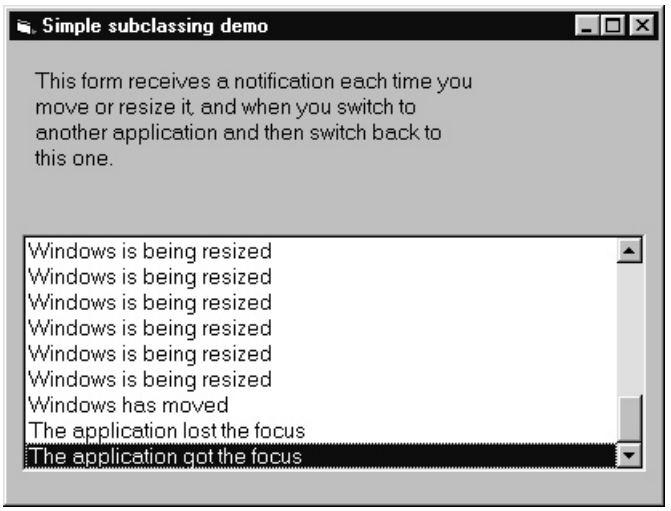


Figure A-9. *A program that demonstrates the basic concepts of window subclassing.*

You can generally process the incoming messages before or after calling the *CallWindowProc* API function. If you're interested only in knowing when a message is sent to the window, it's often preferable to trap it after the Visual Basic runtime has processed it because you can query updated form's properties. Remember, Windows expects that you return a value to it, and the best way to comply with this requirement is by using the value returned by the original window procedure. If you process a message before forwarding it to the original procedure, you can change the values in *wParam* or *lParam*, but this technique requires an in-depth knowledge of the inner workings of Windows. Any error in this phase is fatal because it prevents the Visual Basic application from working correctly.

> **CAUTION** Of all the advanced programming techniques you can employ in Visual Basic, subclassing is undoubtedly the most dangerous one. If you make a mistake in the custom window procedure, Windows won't forgive you and won't give you a chance to fix the error. For this reason, you should *always* save your code before running the program in the environment. Moreover, you should *never* stop a running program using the End button, an action which immediately stops the running program and prevents the *Unload* and *Terminate* events from executing, therefore depriving you of the opportunity to restore the original window procedure.

A class for subclassing

Although the code presented in the previous version works flawlessly, it doesn't meet the requirements of real-world applications. The reason is simple: In a complex program, you usually subclass multiple forms and controls. This practice raises a couple of interesting problems:

- You can't use simple variables to store the window's handle and the address of the original window procedure—as the previous simplified example does—but you need an array or a collection to account for multiple windows.
- The custom window procedure must reside in a BAS form, so the same procedure must serve multiple subclassed windows and you need a way to understand which window each message is bound to.

The best solution to both problems is to build a class module that manages all the subclassing chores in the program. I've prepared such a class, named MsgHook, and as usual you'll find it on the companion CD. Here's an abridged version of its source code:

```
' The MsgHook.cls class module
Event AfterMessage(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, retValue As Long)

Private m_hWnd As Long          ' Handle of the window being subclassed

' Start the subclassing.
Sub StartSubclass(ByVal hWnd As Long)
    ' Terminate current subclassing if necessary.
    If m_hWnd Then StopSubclass
    ' Store argument in member variable.
    m_hWnd = hWnd
    ' Add a new item to the list of subclassed windows.
    If m_hWnd Then HookWindow Me, m_hWnd
End Sub

' Stop the subclassing.
Sub StopSubclass()
    ' Delete this item from the list of subclassed windows.
    If m_hWnd Then UnhookWindow Me
End Sub

' This procedure is called when a message is sent to this window.
' (It's Friend because it's meant to be called by the BAS module only.)
Friend Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, _
    ByVal oldWindowProc As Long) As Long

    Dim retValue As Long, Cancel As Boolean
    ' Call original window procedure.
    retValue = CallWindowProc(oldWindowProc, hWnd, uMsg, wParam, lParam)
    ' Call the application.
    ' The application can modify the retValue argument.
    RaiseEvent AfterMessage(hWnd, uMsg, wParam, lParam, retValue)
```

(continued)

```
    ' Return the value to Windows.
    WndProc = retValue
End Function

' Stop the subclassing when the object goes out of scope.
Private Sub Class_Terminate()
    If m_hWnd Then StopSubclass
End Sub
```

As you see, the class communicates with its clients through the *AfterMessage* event, which is called immediately after the original window procedure has processed the message. From the client application's standpoint, subclassing a window has become just a matter of responding to an event, an action familiar to all Visual Basic programmers.

Now analyze the code in the BAS module in which the subclassing actually occurs. First of all, you need an array of UDTs, where you can store information about each window being subclassed:

```
' The WndProc.Bas module
Type WindowInfoUDT
    hWnd As Long            ' Handle of the window being subclassed
    oldWndProc As Long      ' Address of the original window procedure
    obj As MsgHook          ' The MsgHook object serving this window
End Type

' This array stores data on subclassed windows.
Dim WindowInfo() As WindowInfoUDT
' This is the number of elements in the array.
Dim WindowInfoCount As Long
```

The *HookWindow* and *UnhookWindow* procedures are called by the MsgHook class's *StartSubclass* and *StopSubclass* methods, respectively:

```
' Start the subclassing of a window.
Sub HookWindow(obj As MsgHook, ByVal hWnd As Long)
    ' Make room in the array if necessary.
    If WindowInfoCount = 0 Then
        ReDim WindowInfo(10) As WindowInfoUDT
    ElseIf WindowInfoCount > UBound(WindowInfo) Then
        ReDim Preserve WindowInfo(WindowInfoCount + 9) As WindowInfoUDT
    End If
    WindowInfoCount = WindowInfoCount + 1

    ' Store data in the array, and start the subclassing of this window.
    With WindowInfo(WindowInfoCount)
        .hWnd = hWnd
        Set .obj = obj
        .oldWndProc = SetWindowLong(hWnd, GWL_WNDPROC, AddressOf WndProc)
```

```
    End With
End Sub

' Stop the subclassing of the window associated with an object.
Sub UnhookWindow(obj As MsgHook)
    Dim i As Long, objPointer As Long
    For i = 1 To WindowInfoCount
        If WindowInfo(i).obj Is obj Then
            ' We've found the object that's associated with this window.
            SetWindowLong WindowInfo(i).hWnd, GWL_WNDPROC, _
                WindowInfo(i).oldWndProc
            ' Remove this element from the array.
            WindowInfo(i) = WindowInfo(WindowInfoCount)
            WindowInfoCount = WindowInfoCount - 1
            Exit For
        End If
    Next
End Sub
```

The last procedure left to be seen in the BAS module is the custom window procedure. This procedure has to search for the handle of the target window of the incoming message among those stored in the *WindowInfo* array and notify the corresponding instance of the MsgHook class that a message has arrived:

```
' The custom window procedure
Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    Dim i As Long, obj As MsgHook
    Const WM_DESTROY = &H2

    ' Find this handle in the array.
    For i = 1 To WindowInfoCount
        If WindowInfo(i).hWnd = hWnd Then
            ' Notify the object that a message has arrived.
            WndProc = WindowInfo(i)obj.WndProc(hWnd, uMsg, wParam, lParam,_
                WindowInfo(i).oldWndProc)
            ' If it's a WM_DESTROY message, the window is about to close,
            ' so there is no point in keeping this item in the array.
            If uMsg = WM_DESTROY Then WindowInfo(i).obj.StopSubclass
            Exit For
        End If
    Next
End Function
```

NOTE The preceding code looks for the window handle in the array using a simple linear search; when the array contains only a few items, this approach is sufficiently fast and doesn't add significant overhead to the class. If you plan to subclass more than a dozen forms and controls, you should implement a more sophisticated search algorithm, such as a binary search or a hash table.

In general, a window is subclassed until the client application calls the *StopSubclass* method of the related MsgHook object or until the object itself goes out of scope. (See the code in the class's *Terminate* event procedure.) The code in the *WndProc* procedure uses an additional trick to ensure that the original window procedure is restored before the window is closed. Because it's already subclassing the window, it can trap the WM_DESTROY message, which is the last message (or at least one of the last messages) sent to a window before it closes. When this message is detected, the code immediately stops subclassing the window.

Using the MsgHook class

Using the MsgHook class is pretty simple: You assign an instance of it to a WithEvents variable, and then you invoke its *StartSubclass* method to actually start the subclassing. For example, you can trap WM_MOVE messages using this code:

```
Dim WithEvents FormHook As MsgHook

Private Sub Form_Load()
    Set FormHook = New MsgHook
    FormHook.StartSubclass Me.hWnd
End Sub

Private Sub FormHook_AfterMessage(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, retValue As Long)
    Const WM_MOVE = &H3
    If uMsg = WM_MOVE Then
        lblStatus.Caption = "The window has moved."
    End If
End Sub
```

If you want to subclass other forms or controls, you have to create multiple instances of the MsgHook class—one for each window to be subclassed—and assign them to distinct WithEvents variables. And of course, you have to write the proper code in each *AfterMessage* event procedure. The complete class provided on the companion CD supports some additional features, including a *BeforeMessage* event that fires before the original window procedure processes the message and an *Enabled* property that lets you temporarily disable the subclassing for a given window. Keep in mind that the MsgHook class can subclass only windows belonging to the current application; interprocess window subclassing is beyond the current capabilities of Visual Basic and requires some C/C++ wizardry.

The MsgHook class module encapsulates most of the dangerous details of the subclassing technique. When you turn it into an ActiveX DLL component—or use the version provided on the companion CD—you can safely subclass any window created by the current application. You can even stop an interpreted program without any adverse effects because the End button doesn't prevent the *Terminate* event from

firing if the class has been compiled in a separate component. The compiled version also solves most—but not all—of the problems that occur when an interpreted code enters break mode, during which the subclassing code can't respond to messages. In such situations, you usually get an application crash, but the MsgHook class will prevent it from happening. I plan to release a more complete version of this class, which I'll make available for download from my Web site at *http://www.vb2themax.com*.

More subclassing examples

Now that you have a tool that implements all the nitty-gritty details of subclassing, you might finally see how subclassing can actually help you deliver better applications. The examples I show in this section are meant to be just hints of what you can really do with this powerful technique. As usual, you'll find all the code explained in this section in a sample application provided on the companion CD. The demonstration application is also shown in Figure A-10.

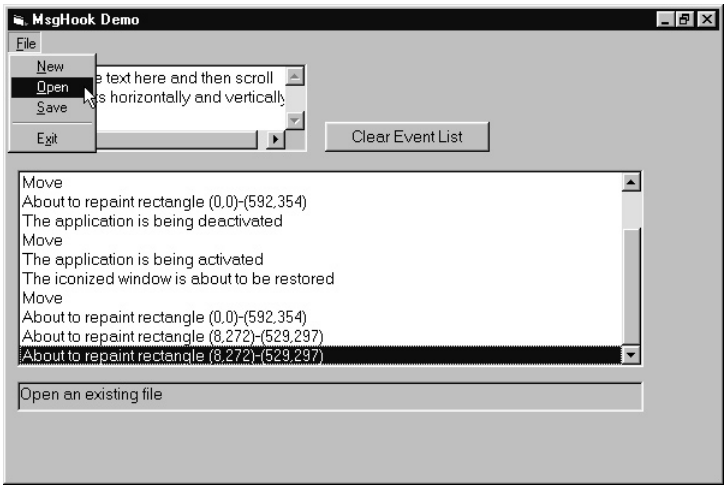


Figure A-10. *The demonstration application that shows what you can achieve with the MsgHook ActiveX DLL.*

Windows sends Visual Basic forms a lot of messages that the Visual Basic runtime doesn't expose as events. Sometimes you don't have to manipulate incoming parameters because you're subclassing the form only to find out when the message arrives. There are many examples of such messages, including WM_MOUSEACTIVATE (the form or control is being activated with the mouse), WM_TIMECHANGE (the system date and time has changed), WM_DISPLAYCHANGE (the screen resolution has changed), WM_COMPACTING (Windows is low in memory and is asking applications to release as much memory as possible), and WM_QUERYOPEN (a form is about to be restored to normal size from an icon).

Many other messages can't be dealt with so simply, though. For example, the WM_GETMINMAXINFO message is sent to a window when the user begins to move or resize it. When this message arrives, *lParam* contains the address of a MINMAXINFO structure, which in turn holds information about the region to which the form can be moved and the minimum and maximum size that the window can take. You can retrieve and modify this data, thus effectively controlling a form's size and position when the user resizes or maximizes it. (If you carefully look at Figure A-10, you'll see from the buttons in the window's caption that this form is maximized, even if it doesn't take the entire screen estate.) To move this information into a local structure, you need the *CopyMemory* API function:

```
Type POINTAPI
    X As Long
    Y As Long
End Type
Type MINMAXINFO
    ptReserved As POINTAPI
    ptMaxSize As POINTAPI
    ptMaxPosition As POINTAPI
    ptMinTrackSize As POINTAPI
    ptMaxTrackSize As POINTAPI
End Type

Private Sub FormHook_AfterMessage(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, retValue As Long)
    Select Case uMsg
        Case WM_GETMINMAXINFO
            ' Windows is querying the form for its
            ' minimum and maximum size and position.
            Dim mmInfo As MINMAXINFO
            ' Read contents of structure pointed to by lParam.
            CopyMemory mmInfo, ByVal lParam, Len(mmInfo)
            With mmInfo
                ' ptMaxSize is the size of the maximized form.
                .ptMaxSize.X = 600
                .ptMaxSize.Y = 400
                ' ptMaxPosition is the position of the maximized form.
                .ptMaxPosition.X = 100
                .ptMaxPosition.Y = 100
                ' ptMinTrackSize is the minimum size of a form when
                ' resized with the mouse.
                .ptMinTrackSize.X = 300
                .ptMinTrackSize.Y = 200
                ' ptMinTrackSize is the maximum size of a form when
                ' resized with the mouse (usually equal to ptMaxSize).
                .ptMaxTrackSize.X = 600
                .ptMaxTrackSize.Y = 400
            End With
```

```
            ' Copy the data back into the original structure in memory.
            CopyMemory ByVal lParam, mmInfo, Len(mmInfo)
            ' Return 0 to say that the structure has been modified.
            retValue = 0
    End Select
End Sub
```

By subclassing the WM_MENUSELECT message, you can add a professional touch to your application. This message fires whenever the user highlights a menu item using the mouse or arrow keys, and you can employ it for displaying a short explanation of the menu item, as most commercial programs do (as shown in Figure A-10). The problem with this message is that you have to process the values stored in *wParam* and *lParam* to extract the caption of the highlighted menu item:

```
' Put this code inside a FormHook_AfterMessage event procedure.
Case WM_MENUSELECT
    ' The menu item identifier is in the low-order word of wParam.
    ' The menu handle is in lParam.
    Dim mnuId As Long, mnuCaption As String, length As Long
    mnuId = (wParam And &HFFFF&)
    ' Get the menu caption.
    mnuCaption = Space$(256)
    length = GetMenuString(lParam, mnuId, mnuCaption, Len(mnuCaption), 0)
    mnuCaption = Left$(mnuCaption, length)
    Select Case mnuCaption
        Case "&New"
            lblStatus.Caption = "Create a new file"
        Case "&Open"
            lblStatus.Caption = "Open an existing file"
        Case "&Save"
            lblStatus.Caption = "Save a file to disk"
        Case "E&xit"
            lblStatus.Caption = "Exit the program"
    End Select
```

WM_COMMAND is a multipurpose message that a form receives on many occasions—for example, when a menu command has been selected or when a control sends the form a notification message. You can trap EN_HSCROLL and EN_VSCROLL notification messages that TextBox controls send their parent forms when their edit area has been scrolled:

```
' Put this code inside a FormHook_AfterMessage event procedure.
Case WM_COMMAND
    ' If this is a notification from a control, lParam holds its handle.
    If lParam = txtEditor.hwnd Then
        ' In this case, the notification message is in the
        ' high-order word of wParam.
        Select Case (wParam \ &H10000)
            Case EN_HSCROLL
```

(continued)

```
            ' The TextBox control has been scrolled horizontally.
        Case EN_VSCROLL
            ' The TextBox control has been scrolled vertically.
    End Select
End If
```

Of course, you can subclass any control that exposes the *hWnd* property, not just forms. For example, TextBox controls receive a WM_CONTEXTMENU message when the user right-clicks on them. The default action for this message is to display the default edit pop-up menu, but you can subclass the TextBox control to suppress this action so that you might display your own pop-up menu. (Compare this technique with the trick shown in the "Pop-Up Menus" tip in Chapter 3.) To achieve this result, you need to write code in the *BeforeMessage* event procedure and you must set the procedure's *Cancel* parameter to False to ask the MsgHook class not to execute the original window procedure. (This is one of the few cases when it's safe to do so.)

```
Dim WithEvents TextBoxHook As MsgHook

Private Sub Form_Load()
    Set TextBoxHook = New MsgHook
    TextBoxHook.StartSubclass txtEditor.hWnd
End Sub

Private Sub TextBoxHook_BeforeMessage(hWnd As Long, uMsg As Long, _
    wParam As Long, lParam As Long, retValue As Long, Cancel As Boolean)
    If uMsg = WM_CONTEXTMENU Then
        ' Show a custom popup menu.
        PopupMenu mnuMyCustomPopupMenu
        ' Cancel the default processing (i.e., the default context menu).
        Cancel = True
    End If
End Sub
```

This appendix has taken you on quite a long journey through API territory. But as I told you at the beginning, these pages only scratch the surface of the immense power that Windows API functions give you, especially if you couple them with subclassing techniques. The MsgHook class on the companion CD is a great tool for exploring these features because you don't have to worry about the implementation details, and you can concentrate on the code that produces the effects you're interested in.

If you want to learn more about this subject, I suggest that you get a book, such as *Visual Basic Programmer's Guide to the Win32 API* by Dan Appleman, specifically on this topic. You should also always have the Microsoft Developer Network at hand for the official documentation of the thousands of functions that Windows exposes. Become an expert in API programming, and you'll see that there will be very little that you can't do in Visual Basic.