David McCarter's VB Tips and Techniques

DAVID McCARTER

Apress™

David McCarter's VB Tips and Techniques
Copyright ©2000 by David McCarter

Developmental Editor: Nancy DelFavero
Page Composition: Impressions Book and Journal Services, Inc.
Indexer: Mike Gossman
Cover and Interior Design: Derek Yee Design

Distributed to the book trade worldwide by Springer-Verlag New York, Inc.
175 Fifth Avenue, New York, NY 10010
In the United States, phone 1-800-SPRINGER; orders@springer-ny.com
http://www.springer-ny.com

For information on translations, please contact APress directly:
APress, 6400 Hollis Street, Suite 9, Emeryville, CA 94608
Phone: 510/595-3110; Fax: 510/595-3122; info@apress.com; www.apress.com

CHAPTER 9

Tips and Tricks on Some Advanced Stuff

THIS CHAPTER IS GEARED TO THOSE programmers who need to push Visual Basic that extra mile. To benefit from most of the tips listed here, you will need to be an experienced user of Windows API calls. You'll also find some tips on which files to install to get your Visual Basic program to run correctly, plus other helpful tidbits.

An Introduction to the AddressOf Operator

Compatible with: Visual Basic 5 and 6
Applies to: Message Callbacks

Unlike Microsoft Windows, the Visual Basic development environment is not based on a message-driven programming model. In Microsoft Windows, messages control most of everything that happens.

Visual Basic, on the other hand, supports a predefined set of events for each object (a Form or Control) that you create. This means that an application written in Visual Basic cannot respond to messages from Microsoft Windows that are not handled directly by a Visual Basic event. This might change some day, if we keep our fingers crossed, but for now you have to work around it.

For example, if you want to provide some help to a user whenever the user moves his mouse over menu items in your project, it can't be done with just Visual Basic functions. But with the use of Windows API calls and the Visual Basic `AddressOf` function, you can listen to all the messages that are being sent to your Form. When you find the appropriate message, you can then invoke some sort of action.

In the sample project for this tip, you will be looking for the `WM_MENUSELECT` message and displaying some help in the status bar. (See Figure 9-1 for an example of such a display.) You can find the example code in the addressof.vbp project in the Chapter 9 directory in the download for this title on the Apress Web site (see the Introduction for more information).

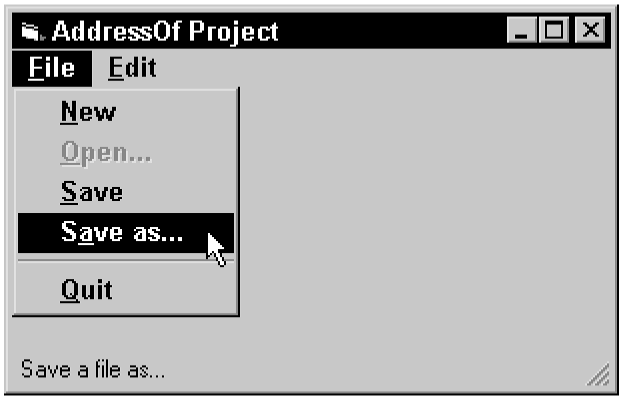


Figure 9-1. Using the `AddressOf` *function to display context-sensitive help*

The `AddressOf` function works like this: All Visual Basic methods live in memory when your project is running. Therefore, all methods have an address in memory that they are occupying. The `AddressOf` function returns that address to you for use in the `SetWindowsHookEx` API function. This function "hooks" into the Windows message queue so that you can intercept the messages coming to the Form.

Now take a look at how the `AddressOf` function is used in the sample project. First, you need to declare the following code in a module.

```
Private Type POINTAPI
  x As Long
  y As Long
End Type
Private Type MSG
  hWnd As Long
  Message As Long
  wParam As Long
  lParam As Long
  time As Long
  pt As POINTAPI
End Type
Private Const WH_MSGFILTER = (-1)
Private Const WM_MENUSELECT = &H11F
Private Const MSGF_MENU = 2
Private Const MF_BYCOMMAND = &H0&
Private Const MF_BYPOSITION = &H400&
Private Const MF_DISABLED = &H2&
Private Const MF_POPUP = &H10&
Private Declare Function SetWindowsHookEx Lib "user32" Alias _
                    "SetWindowsHookExA" (ByVal idHook As Long, _
                    ByVal lpfn As Long, ByVal hmod As Long, _
                    ByVal dwThreadId As Long) As Long
Private Declare Function UnhookWindowsHookEx Lib "user32" _
                    (ByVal mlhHook As Long) As Long
Private Declare Function CallNextHookEx Lib "user32" _
                    (ByVal mlhHook As Long, ByVal nCode As Long, _
                    ByVal wParam As Long, lParam As Any) As Long
Private Declare Function GetWindowThreadProcessId Lib "user32" _
                    (ByVal hWnd As Long, lpdwProcessId As Long) _
                    As Long
Private Declare Function GetMenuString Lib "user32" Alias _
                    "GetMenuStringA" (ByVal hMenu As Long, _
                    ByVal wIDItem As Long, _
                    ByVal lpString As String, _
                    ByVal nMaxCount As Long, ByVal wFlag As Long) _
                    As Long
Private mlhHook As Long
Private mlMenuhWnd As Long
Private mfrmCallBack As Form
```

You then need to set up the callback in the `Form_Load` event.

```
Call InitHook(Me)
```

Then, put the `InitHook` subroutine in the same modules as those shown in the code I just listed.

```
Public Sub InitHook(CallBackForm As Form)
  Set mfrmCallBack = CallBackForm
  mlMenuhWnd = CallBackForm.hWnd
  mlhHook = SetWindowsHookEx(WH_MSGFILTER, _
                      AddressOf HookMenuProc, _
                            ByVal 0&, _
                      GetWindowThreadProcessId(mlMenuhWnd, _
                        ByVal 0&))
  If mlhHook = 0 Then
    End
  End If
End Sub
```

The `InitHook` subroutine uses `AddressOf` to set up a method into which the messages from the Windows message queue are pumped. In this sample code, you are sending the messages to the HookMenuProc subroutine. With the help of the UpdateStatus subroutine in the test Form, the code filters out the message you want and displays help based on the menu item the user's mouse pointer is resting over.

```
Private Function HookMenuProc(ByVal lCode As Long, _
                                ByVal lParam As Long, _
                                mMessage As MSG) As Long
Dim lReturn As Long
Dim lItemID As Long
Dim lItemHandle As Long
Dim lItemFlags As Long
Dim lItemMaxCount As Long
Dim sItemString As String
Dim bMenuDisabled As Boolean
  If lCode = MSGF_MENU Then
    If mMessage.Message = WM_MENUSELECT Then
      lItemHandle = mMessage.lParam
      lItemID = mMessage.wParam And 65535
      lItemFlags = mMessage.wParam / 65535
      If lItemFlags = &HFFFF And lItemHandle = vbNull Then
        sItemString = vbNullString
        Else
          sItemString = Space$(255)
          lItemMaxCount = 255
          If (lItemFlags And MF_POPUP) = MF_POPUP Then
            lReturn = GetMenuString(lItemHandle, lItemID, _
                      sItemString, _
                      lItemMaxCount, MF_BYPOSITION)
            Else
              lReturn = GetMenuString(lItemHandle, lItemID, _
                                      sItemString, lItemMaxCount, _
                                      MF_BYCOMMAND)
          End If
          sItemString = Left$(sItemString, lReturn)
      End If
      If (lItemFlags And MF_DISABLED) = MF_DISABLED Then
        bMenuDisabled = True
        Else
          bMenuDisabled = False
      End If
      mfrmCallBack.UpdateStatus sItemString, bMenuDisabled
    End If
  End If
  lReturn = CallNextHookEx(mlhHook, lCode, lParam, mMessage)
 End Function
```

The code in `HookMenuProc` is fairly easy to follow. I do need to point out one thing, however. At the end of the subroutine, you are calling the `CallNextHookEx` API function because you may not be the only one listening to the messages from the Windows message queue; therefore, you need to pass the message along.

The `UpdateStatus` subroutine should be put in the Form, as follows.

```
Public Sub UpdateStatus(Message As String, MenuDisabled As Boolean)
Dim sDescription As String
  Select Case Message
    Case "&File"
      sDescription = "File menu"
    Case "&New"
      sDescription = "Create a new file"
    Case "&Open..."
      sDescription = "Open a file"
    Case "&Save"
      sDescription = "Save a file"
    Case "S&ave as..."
      sDescription = "Save a file as..."
    Case "&Quit"
      sDescription = "End the application"
    Case "&Edit"
      sDescription = "Edit menu"
    Case "&Cut"
      sDescription = "Cut the text"
    Case "C&opy"
      sDescription = "Copy the text"
    Case "&Paste"
      sDescription = "Paste a text"
    Case Else
      sDescription = vbNullString
  End Select
  If Len(sDescription) > 0 And MenuDisabled Then
    sDescription = "Disabled: " & sDescription
  End If
  StatusBar1.Panels("status").Text = sDescription
End Sub
```

When the Form unloads, you must tell the operating system to stop sending messages. Here is how you do that.

```
Private Sub Form_QueryUnload(Cancel As Integer, _
                              UnloadMode As Integer)
  Call EndHook
End Sub
```

Finally, put the following code in the module file.

```
Public Sub EndHook()
Dim lReturn As Long
  lReturn = UnhookWindowsHookEx(mlhHook)
  Set mfrmCallBack = Nothing
End Sub
```

That's all there is to it. For more information, take a look at the Visual Studio help file for information on `AddressOf` and other things you can do with the API calls used in this tip.

WARNING *Do not put a break in your code while running in the Visual Basic IDE. If you stop your code at any time whenever you use a message hook, very bad things will happen, such as your entire system locking up. This is primarily due to the fact that you are intercepting messages from the operating system. When your code stops at a break, the messages keep coming to your program and undesirable things can happen. If you want safer message hooking, you might want to get a copy of Spyworks from Desaware* (`www.desaware.com`).

Detecting whether Your Program Is Running in the Integrated Development Environment

Compatible with: Visual Basic 32-bit
Applies to: Applications

It is often useful to know whether or not a program is running in the Integrated Development Environment (IDE) instead of as a compiled executable. For instance, an error-handling routine can stop a program if an error occurs rather than handle the situation, which would be the case in the compiled version.

The routine for this tip takes advantage of an idiosyncrasy in Visual Basic. All Windows programs must declare a window class for every window. In Visual

Basic, the window class of all Forms is always ThunderForm ("Thunder" was the code name for Visual Basic 1.0). But, the window class for the hidden parent window is different depending on whether the Form is running from the IDE or if it is compiled.

In the IDE, the parent window class is ThunderForm, just as it is with all other Visual Basic windows. In the EXE, however, the class of the parent window is ThunderRT6Form (for Visual Basic 6). Therefore, the `IDEMode` function will get the window class of the hidden parent window and look for the string `RT` (which stands for runtime). If the string is found, the program is running as an EXE.

Declare

You'll need to declare the following code in a module file.

```
Public Declare Function GetClassName Lib "user32" Alias _
                          "GetClassNameA" (ByVal hwnd As Long, _
                          ByVal lpClassName As String, _
                          ByVal nMaxCount As Long) As Long
Public Declare Function GetWindowLong Lib "user32" Alias _
                          "GetWindowLongA" (ByVal hwnd As Long, _
                          ByVal nIndex As Long) As Long
Public Const GWL_HWNDPARENT = (-8)
```

Code

Here is the `IDEMode` function that returns `True` if the project is running in the IDE.

```
Function IDEMode(FormhWnd As Long) As Boolean
Dim lParent As Long
Dim sClass As String
Dim lReturn As Long
  lParent = GetWindowLong(FormhWnd, GWL_HWNDPARENT)
  sClass = Space$(32)
  lReturn = GetClassName(CLng(lParent), sClass, Len(sClass))
  sClass = Left$(sClass, lReturn)
  Debug.Print sClass
  If InStr(sClass, "RT") Then
    IDEMode = False
    Else
      IDEMode = True
  End If
End Function
```

Example

The following is an example of how to use the `IDEMode` function.

```
Private Sub Form_Load()
  If IDEMode(Me.hwnd) Then
    MsgBox "IDE detected"
  End If
End Sub
```

You can find the sample code for this tip in the IDEMode.vbp project in the Chapter 9 directory in the downloadable code on the Apress Web site.

Using the SHFormatDrive Function to Format Drives

Compatible with: Visual Basic 32-bit
Applies to: Drives, Windows 95/98, and Windows NT

If you want to provide an easy way for users of your program to format a drive, why not bring up the same Format dialog box that Microsoft Explorer does? (See Figure 9-2.)

To call the Format dialog box, use the `SHFormatDrive` API function. This function is undocumented by Microsoft (I guess they don't trust us with it). So don't look for any help with this on the Microsoft Development Network CD (although you can find some information on `SHFormatDrive` for C++ developers in article Q173688 in the Microsoft Knowledge Base). Here, I give you all the information you need to get `SHFormatDrive` to work properly.

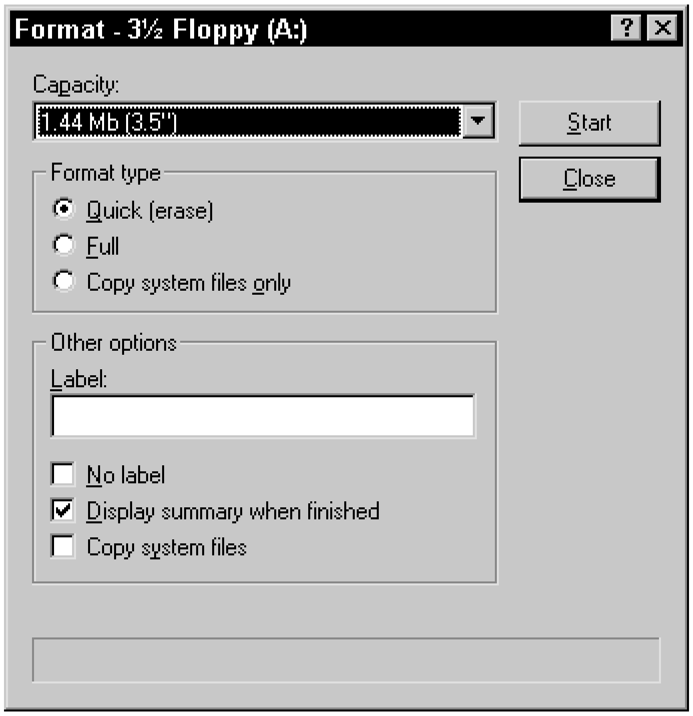
First, take a look at the parameters for `SHFormatDrive`.

```
Public Declare Function SHFormatDrive Lib "shell32" _
                  (ByVal hWnd As Long, ByVal Drive As Long, _
                   ByVal fmtID As Long, ByVal Options As Long) _
                   As Long
```

- **`hWnd`** The Windows handle to the Form from which you are calling this function.

- **`Drive`** The drive number that you want to format. Every drive in a computer is assigned a number starting with the A: drive, which is assigned the number 0. Here is an easy way to determine the drive number.

  ```
  lDriveNum = (Asc("A") - 65)
  ```



Capacity: 1.44 Mb (3.5")

Format type
[✓ Quick (erase)]
Full
Copy system files only

Other options
Label:
No label
[✓ Display summary when finished]
Copy system files

Figure 9-2. The Format dialog box can be called with the `SHFormatDrive` *API function*

- **fmtID** This parameter is used to configure the type of media in the drive. You can set this parameter to work with any type of format, but it is easier to simply let it default to the type of media that is in the drive. Here is constant used to accomplish that.

```
Public Const SHFMT_ID_DEFAULT = &HFFFF
```

- **`Options`** The parameter used to configure the type of formatting required for the particular media in the drive. You have the following choices.

```
Public Const SHFMT_OPT_QUICK = &H0 'Quick Format
Public Const SHFMT_OPT_FULL = &H1 'Full Format
Public Const SHFMT_OPT_SYSONLY = &H2 'System Files Only
```

- **Return Values** If an error occurs, the function returns one of the following values.

```
Public Const SHFMT_NOFORMAT = &HFFFFFFFD
Public Const SHFMT_CANCEL = &HFFFFFFFE
Public Const SHFMT_ERROR = &HFFFFFFFF
```

If an error occurs, a message is displayed to the user.

Example

Here is an example of how to use `SHFormatDrive`.

```
lReturn = SHFormatDrive(Me.hWnd, lDriveNum, SHFMT_ID_DEFAULT, _
                              SHFMT_OPT_QUICK)
```

Under most circumstances, you should make sure that the drive is removable (most likely it's a floppy drive). You can use the following API call and its constants to do that.

```
Public Const DRIVE_REMOVABLE = 2
Public Const DRIVE_FIXED = 3
Public Const DRIVE_REMOTE = 4
Public Const DRIVE_CDROM = 5
Public Const DRIVE_RAMDISK = 6
Public Declare Function GetDriveType Lib "kernel32" Alias _
                  "GetDriveTypeA" (ByVal nDrive As String) _
                   As Long
```

You should always use the `GetDriveType` function and make sure the number returned is 2 before you use the `SHFormatDrive` function.

You can find the example code for this tip in the FormatDrive.vbp project in the Chapter 9 directory for this book's downloadable code samples (see the Introduction).

WARNING *Be careful when using this code. You could accidentally wipe out your drive or a user's drive if you're not careful!*

Visual Basic Runtime Files

Compatible with: Visual Basic 5 and 6
Applies to: Installations

One question I'm frequently asked at my San Diego Visual Basic Users Group meetings is why a particular program won't work on another computer. Programmers will tell me about e-mailing an executable to someone who can't get it to run. What they may not realize is that for any Visual Basic executable to run, it needs at a minimum the 2MB-plus of the runtime files that get installed with Visual Basic.

Surprise, fear, and horror are just a few of the expressions I see on their faces after I tell them this.

Of course, your programs may need many more files than just the basic Visual Basic runtime files. I could fill an entire chapter on writing good installations (I'll keep that in mind for my next book). If you stick to using the Package and Deployment Wizard that comes with Visual Basic, there isn't much to worry about. It includes all the runtime files Visual Basic requires.

Most seasoned programmers do not use the Package and Deployment Wizard because it's buggy, not very configurable, and slow. The newsgroup traffic in Visual Basic discussion groups would probably drop by 25 percent if Microsoft would stop including it. Most of us turn to a "real" installation package, such as the Wise Installation System (`www.wisesolutions.com`) or InstallShield (`www.installshield.com`). I prefer the Wise Installation System because it's easier to use (and cheaper, too).

If you are creating the install using one of the software programs I just mentioned, you will need a list of the Visual Basic runtime files. The list is also good to have when diagnosing problems on a user's computer. Tables 9-1 and 9-2 list the runtime files and their install settings required for Visual Basic 5 and 6.

*Table 9-1. Visual Basic 5 Runtime Files**

FILE NAME	INSTALL SETTINGS
MSVBVM50.DLL	Version Check, Win Shared DLL, Self Register
STDOLE2.TLB	Version Check, TLB Register
OLEAUT32.DLL	Version Check, Win Shared DLL, Self Register
OLEPRO32.DLL	Version Check, Win Shared DLL, Self Register
ASYCFILT.DLL	Version Check, Win Shared DLL, Self Register
CTL3D32.DLL	Version Check
COMCAT.DLL	Version Check, Win Shared DLL, Self Register

*Total Disk Space Required: 2,308,672 bytes

*Table 9-2. Visual Basic 6 Runtime Files**

FILE NAME	INSTALL SETTINGS
MSVBVM60.DLL	Version Check, Win Shared DLL, Self Register
STDOLE2.TLB	Version Check, TLB Register
OLEAUT32.DLL	Version Check, Win Shared DLL, Self Register
OLEPRO32.DLL	Version Check, Win Shared DLL, Self Register
ASYCFILT.DLL	Version Check, Win Shared DLL, Self Register
COMCAT.DLL	Version Check, Win Shared DLL, Self Register

*Total Disk Space Required: 2,318,640 bytes

The runtime file sizes may differ on your computer depending on which service pack you have installed on your system. Always try to use the latest Visual Basic service pack available from Microsoft.

The following describes the different Install Settings listed in Tables 9-1 and 9-2.

- **Version Check** The file must be versioned checked. Never install an older version over a newer version and make sure you install the file if the version is the same or older. Why should you install over the same version? Wouldn't it save time if you didn't? Yes, but if the file was somehow corrupted and left the version information intact, then you need to install over it.

- **Win Shared DLL** The file is a Windows Shared DLL. You should mark it as a shared DLL with your install program so that Windows keeps track of how many programs installed that file. Then when an uninstall program tries to remove the file, Windows will warn the user that the file may be used by other programs.

- **Self Register** The file is self-registering.

- **TLB Register** The TLB file needs to be registered. Your install program should provide a way to do this. This step is just as important as registering an executable file.

ActiveX Components and Long File Names

Compatible with: Visual Basic 5 and 6
Applies to: ActiveX (COM) Components

Be especially careful when naming your ActiveX DLL, EXE and OCX files. For example, at my workplace we designed an application (in Visual Basic 5 and C++) and used long file names. For some unexplained reason, we had trouble running the COM objects after registering them in the Registration Database. We switched to the old 8.3 file names and the problem went away.

I could not find any official information about this in the Microsoft Knowledge Base. But, if you look, all COM components released by Microsoft use 8.3 file names. Go figure.

Registering and Unregistering COM Objects Easily

Compatible with: All COM (ActiveX) Objects
Applies to: DLLs, OCXs, EXEs

After you create COM (ActiveX) objects in Visual Basic or have used other COM objects in your projects, you will need to register and unregister those objects, for a number of reasons. For instance, you may need to manually install COM objects or you may need to register or unregister objects for testing purposes. I'll bet most of you do this by choosing Start ➔ Run and typing in `regsvr32` and the file name and path of the file you want to register, or typing in `regsvr32 /u` and the file name and path of the file you want to unregister.

Wouldn't you rather register an object by right-clicking the file name in Explorer? (See Figure 9-3.) You can by simply adding some entries into the Registration Database.

Instead of going into a lengthy explanation of how to do this, I have provided some .reg files that enable you to easily add the appropriate entries in the

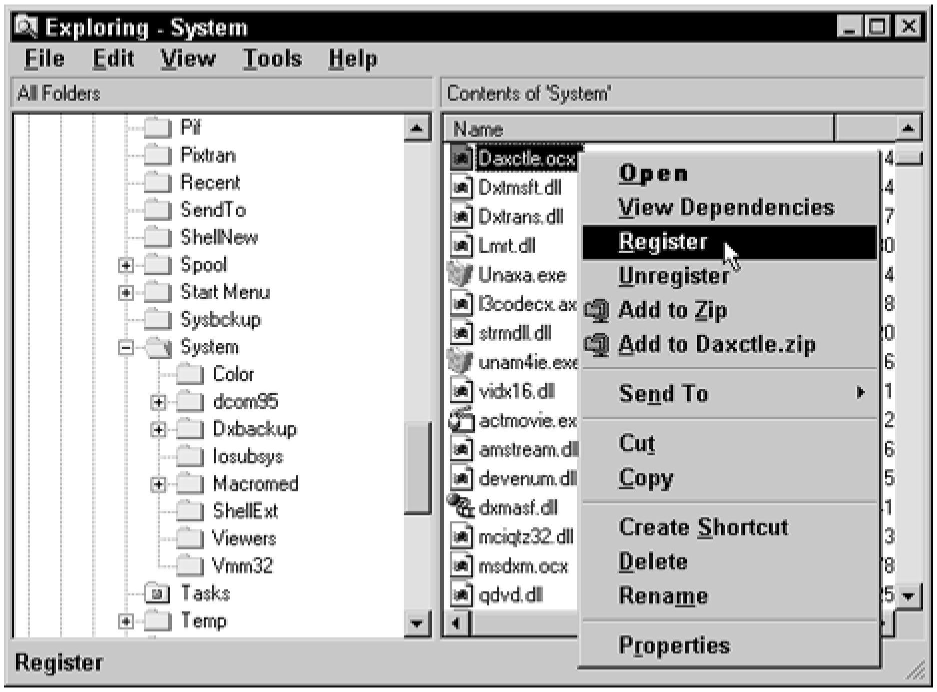


Figure 9-3. Registering COM objects via the Windows Explorer

Registration Database. You can find the following .reg files in the Chapter 9 directory in the downloadable code for this book on the Apress Web site (see the Introduction).

- **regunregexe.reg** Adds the appropriate entries to register and unregister EXE files.

- **regunregdll.reg** Adds the appropriate entries to register and unregister DLL files.

- **regunregocx.reg** Adds the appropriate entries to register and unregister OCX files.

To add the entries, simply double-click the reg file of your choice.