WINDOWS PROGRAMMING

# Get on the Track to Faster Apps

Click & Retrieve Source CODE!

*Use the 32-bit API to improve the performance of your applications.*

by Sam Patterson

One of the most often heard complaints about Visual Basic 4.0 is that it can be somewhat slow, depending on what your application does. However, I've found that Visual Basic actually does many things pretty quickly, especially when you consider the amount of time it takes to write the application. In my last Windows Programming column ["API Calls to Help You Optimize," *VBPJ* January 1996], I used the performance API to measure some of the performance-related issues for your Visual Basic 4.0 programs. This month I'll examine how to use the Windows 32-bit API to improve the performance of your applications by calling some API functions that decrease the memory and resource usage of your applications. Programmers often overlook the idea of decreasing memory and resource usage as a way of increasing application performance. Performance-related issues seem to be a hot topic, so I want to present some neat techniques that can help boost your application's performance.

Around six months ago I was approached by Ward Hitt to write a chapter for his *Optimizing Visual Basic 4* book from Que. I want you to know up front that I am not receiving royalties from this book, and I only wrote the one chapter on using the API to optimize your application. This small plug is genuine: it is a great book. If performance issues plague your VB4 application, this book contains some neat tips and techniques that help optimize the resource usage of your application and improve performance. I thought it would be interesting to include some of these API tips in this month's column.

As you have probably learned from my past columns, using the Windows API can be a good way to speed up the internals of your application. You can also use it to reduce memory requirements and limit the number of resources that your application uses. Calling the Windows API can, at times, be very easy. At other times, because of its vastness, you might have difficulty finding the information and functions to use. The Windows API covers many diverse areas of functionality. Most API calls are directly accessible from Visual Basic, but some are not. If you need an API function that provides callback functionality, or a function that has a parameter that requires a pointer to a function to be passed, then you need to use an intermediate method to call it. Many third-party companies provide callback OLE controls that provide the functionality needed to use these callback functions: the MhSubClass control in MicroHelp's OLETools or Desaware's SpyWorks/OCX, for example. You should not overlook callback functions because they often can provide performance boosts or provide functionality that otherwise would be unavailable to your application.

## USE THE API WISELY

One word of warning: using the API can sometimes be a risky business while you are learning it. You may inadvertently cause Access Violations (formerly known as GPFs) if you accidentally call the wrong function with the wrong information. Keep this in mind while you are programming, and *always* save your project before you try to test it. I give this advice because it still happens to me. In fact it even happened to me while writing this article!

As you may also have learned from previous Windows Programming columns, Visual Basic itself is a Windows application that was written in C++ using the Windows API. Many of the built-in functions, properties, and methods of Visual Basic objects are really just calls that are passed on from your Visual Basic program to the underlying operating system. By calling these functions directly, you bypass much of the overhead involved

*Sam Patterson is general manager of the Component Products Business Unit of MicroHelp Inc. and a contributing editor of* Visual Basic Programmer's Journal. *He is also owner of Gold Leaf Systems, a Los Angeles, California-based consultancy specializing in VBX/OLE Control component software development. He is coauthor of MicroHelp's OLETools, VBTools, SpellPro, Thesaurus, and VBComm 3.0 Communications Library. He is also the author of the MCI, MAPI, Masked Edit, Rich Text, Toolbar, and TabStrip OLE controls included with Microsoft's Visual Basic 4.0, Visual FoxPro 3.0, and Visual C++ 4.0. Contact Sam at MicroHelp Inc., by e-mail at SamP@microhelp.com or by fax at 404-645-2122; or at Gold Leaf Systems, by mail at 5301 Beethoven Street #190, Los Angeles, CA 90066-7061 or by fax at 310-574-6301. Reach Sam on CompuServe at 72000,1751.*

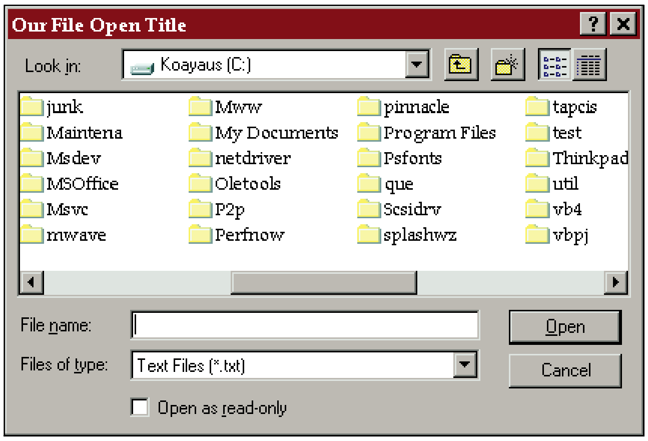

FIGURE 1 ***What Do These Dialogs Have in Common?*** *Whether you use the API or the Common Dialog OLE control, these two dialogs look the same.*

with setting a property in Visual Basic or calling a method on an object. For example, say you have a list box on a form and you implement this code:

```
List1.Clear
```

Visual Basic, when it executes the code, is using the API function SendMessage to send the message LB_RESETCONTENT to the list box. You could replace this code with your own appropriate SendMessage code and therefore bypass the overhead of the Visual Basic method. The code would look something like this:

```
Private Declare Function SendMessage _
   Lib "user32" Alias "SendMessageA" _
   (ByVal hwnd As Integer, ByVal msg _
   As Integer, ByVal wp As Integer, _
   ByVal lp As Long) As Long

b = SendMessage(List1.hWnd, &H184, 0, 0)
```

The time saved by this technique would be negligible because you have to write code that accesses the hWnd property on the list box object in order to send the message. If you need to clear this list box thousands of times in an application, however, then the SendMessage API would make a small but noticeable difference in overall execution speed. If you find an API call that does something like this, you should investigate whether or not the speed or efficiency of your application could improve by using it, and whether the improvement justifies the extra code you have to write.

## COMMON DIALOGS WITHOUT THE OCX

Let's look at a more complicated example. Most applications that allow loading, saving, or using disk files usually use the Windows common dialogs to get the file name and path the user wants. Using the common dialogs is a good reuse of resources because Windows always has the common dialog library loaded (COMDLG32.DLL). Many programmers think that to use the common dialogs from Visual Basic you have to use the Common Dialog OLE custom control (COMDLG32.OCX) that comes in the box with Visual Basic. This is not necessarily true. The Common Dialog control makes it easier for some programmers to get the functionality of the dialogs easily, quickly, and modularly into an application. To tune performance of your application, however, you should try to use as few custom controls as possible. Most of the common dialog routines in the Win32 API are directly callable from Visual Basic.

For example, I've included code to bring up the FileOpen common dialog box (see Listing 1). By calling the COMDLG32.DLL, it gives you the FileOpen functionality without all the extra resource and memory overhead of loading the Common Dialog OLE control. This method takes a little more code than you would have to use to get the file name from the OLE control, but the dialogs that are displayed are the same (see Figure 1). Although the final EXE size differs by only a few hundred bytes, the amount of resources used during run time and the distributable code size goes down quite a bit. You also no longer have to distribute the 80K+ COMDLG32.OCX file on your diskettes.

One thing to watch out for: you might lose some features by using the API to bring up the FileOpen dialog. If you rely on the Help Button on the dialogs, the Common Dialog control uses the HelpFile property to bring up your help file when a user presses that button. If you want to have something similar happen when using the API, you will either have to use a callback OLE control as I mentioned earlier, or write a C DLL wrapper around your call to the Common Dialog code that will handle the notification message that results from the user pressing the help button. If you choose to write a wrapper, it still beats the extra overhead of the full Common Dialog control.

Bringing up the color-selection common dialog box is a similar process, but it takes a small amount of additional programming because the ChooseColor user-defined type contains a member called lpCustColors (see Listing 2). lpCustColors is a

VB4

```
Private Type OPENFILENAME
     lStructSize As Long
     hwndOwner As Long
     hInstance As Long
     lpstrFilter As String
     lpstrCustomFilter As String
     nMaxCustFilter As Long
     nFilterIndex As Long
     lpstrFile As String
     nMaxFile As Long
     lpstrFileTitle As String
     nMaxFileTitle As Long
     lpstrInitialDir As String
     lpstrTitle As String
     flags As Long
     nFileOffset As Integer
     nFileExtension As Integer
     lpstrDefExt As String
     lCustData As Long
     lpfnHook As Long
     lpTemplateName As String
End Type

Private Declare Function GetOpenFileName Lib _
  "comdlg32.dll" Alias "GetOpenFileNameA" _
  (pOpenfilename As OPENFILENAME) As Long

Private Sub Command1_Click()
  Dim ofn As OPENFILENAME

  ofn.lStructSize = Len(ofn)
  ofn.hwndOwner = Form1.hwnd
  ofn.hInstance = App.hInstance
  ofn.lpstrFilter = "Text Files (*.txt)" + Chr$(0) + _
     "*.txt" + Chr$(0) + "Rich Text Files (*.rtf)" + _
     Chr$(0) + "*.rtf" + Chr$(0)
  ofn.lpstrFile = Space$(254)
  ofn.nMaxFile = 255
  ofn.lpstrFileTitle = Space$(254)
  ofn.nMaxFileTitle = 255
  ofn.lpstrInitialDir = "c:\"
  ofn.lpstrTitle = "Our File Open Title"
  ofn.flags = 0

  a = GetOpenFileName(ofn)

  If (a) Then
     MsgBox "File to Open: " + Trim$(ofn.lpstrFile)
  Else
     MsgBox "Cancel was pressed"
  End If
End Sub

End Sub
```

LISTING 1 ***Calling the FileOpen Common Dialog.*** *To use this code, start a new project. Add a command button (Command1), and add this code to create the example program that calls the FileOpen common dialog routines.*

VB4

```
Private Type ChooseColor
    lStructSize As Long
    hwndOwner As Long
    hInstance As Long
    rgbResult As Long
    lpCustColors As String
    flags As Long
    lCustData As Long
    lpfnHook As Long
    lpTemplateName As String
End Type

Private Declare Function ChooseColor Lib
"comdlg32.dll" _
  Alias "ChooseColorA" (pChoosecolor As ChooseColor) _
    As Long

Private Sub Command1_Click()
  Dim cc As ChooseColor
  Dim CustColor(16) As Long

  cc.lStructSize = Len(cc)
  cc.hwndOwner = Form1.hWnd
  cc.hInstance = App.hInstance
  cc.flags = 0

  'NOTE:  We can't pass an array of pointers so
  ' we fake this passing a string of chars:  In this
  'example we set all custom colors to 0, or black.
  cc.lpCustColors = String$(16 * 4, 0)

  a = ChooseColor(cc)

  Cls
  If (a) Then
     MsgBox "Color chosen:" & Str$(cc.rgbResult)

     'Create the custom color array based on
     'the colors passed back from the String
     For x = 1 To Len(cc.lpCustColors) Step 4
        c1 = Asc(Mid$(cc.lpCustColors, x, 1))
        c2 = Asc(Mid$(cc.lpCustColors, x + 1, 1))
        c3 = Asc(Mid$(cc.lpCustColors, x + 2, 1))
        c4 = Asc(Mid$(cc.lpCustColors, x + 3, 1))

        CustColor(x / 4) = (c1) + (c2 * 256) + _
           (c3 * 65536) + (c4 * 16777216&)
        Print "Custom Color"; Int(x / 4); "="; _
           CustColor(x / 4)
     Next x
  Else
     MsgBox "Cancel was pressed"
  End If
End Sub
```

LISTING 2 ***Show Your True Colors.*** *After you start a new project and add a command button (Command1), add this code to create the example program that brings up the Color common dialog box.*

pointer to an array of long integers. Because there is no way to directly pass lpCustColors in Visual Basic, send a string (allocating four bytes for each entry) for each of the 16 custom colors that the dialog supports. On return from the ChooseColor API function call, convert these four characters back into a long integer that you store in an array of custom colors.

From the sample code you should be able to work with the other common dialogs and get them implemented in code instead of using their OLE control equivalents. In the future, Microsoft will continue adding new dialogs to the Common Dialog controls. The more you use the underlying code that they have written as opposed to the control itself, the smaller your applications will be.

## MULTIMEDIA SUPPORT

Many applications provide or use some type of multimedia support, whether or not they are really considered a "multimedia" application. If you play wave sound files (WAV), or start a Video for Windows (AVI) file, you may benefit from using the MCI API directly. Currently you probably are using the Multimedia Control Interface OLE custom control (MCI32.OCX) in your application. Again, if you remove this custom control, your resource usage will go down—in this case by quite a bit. The MCI control not only uses resources for bitmaps, but also incurs other Windows overhead as it creates a button hWnd for each of the buttons that are visible on the control. By eliminating the need for this control, you also improve your load-time performance.

To play a wave file using the MCI interface, add this code to a form:

```
Private Declare Function mciSendString _
   Lib "winmm.dll" Alias _
   "mciSendStringA" (ByVal _
   lpstrCommand As String, ByVal _
   lpstrReturnString As String, _
   ByVal uReturnLength As Long, ByVal _
   hwndCallback As Long) As Long

Private Sub Form_Click()
   Dim ReturnString As String
   ReturnString = Space$(255)
   a = mciSendString("play d:\win95\media\tada.wav", )
   ReturnString, Len(ReturnString), 0)
End Sub
```

The mciSendString API call can provide most of the functionality of the MCI OLE control, minus the graphical interface of course. With a few lightweight image controls and some mouse-handling code, you will have a result that works like the MCI without all the associated overhead. You may still need to do some additional work to get all the functionality of the OLE control, such as subclassing (using a subclassing or callback control) to let your application receive specific notification messages. For further information on what commands are available through the mciSendString API call, see the Windows Multimedia help file that is included with the Win32 SDK, or look in the appropriate Win32 users' manual. Supported devices include, but are not necessarily limited to, wave sound files (WAV), MIDI, Mixer, Video Disc/VCR, and Digital Video (AVI).

As you can see, the API can be a great way to lower the resource and memory usage of your application. When using the API, you can also include fewer OLE controls to decrease the amount of disk space needed to distribute your application. If you use it wisely, this diverse tool can improve your application's actual speed—the API can even improve your app's perceived speed because it can quicken load times. In my next column I'll examine routines that execute faster because they call API routines instead of built-in Visual Basic functions, methods, or objects. Until then, examine the API and happy programming.