Welcome to the 11th Edition of the *VBPJ* Technical Tips Supplement!

These tips and tricks were submitted by professional developers using Visual Basic 3.0 through 6.0, Visual Basic for Applications (VBA), and Visual Basic Script (VBS). The editors at *Visual Basic Programmer's Journal* compiled the tips. Instead of typing the code published here, download the tips for free from the *VBPJ* Web site at www.vbpj.com.

If you'd like to submit a tip to *VBPJ*, please send it electronically to vbpjtips@fawcette.com. You can also send it to User Tips, Fawcette Technical Publications, 209 Hamilton Ave., Palo Alto, California, USA, 94301-2500; or fax it to 650-853-0230. Please include a clear explanation of what the technique does and why it's useful, and indicate if it's for VBA, VBS, VB3, VB4 16- or 32-bit, VB5, or VB6. Please limit code length to 20 lines. Don't forget to include your e-mail and mailing addresses, and let us know your payment preference: $25 per published tip or extending your *VBPJ* subscription by one year.

VB4 32, VB5, VB6
Level: Intermediate

Retrieve File Version Information

Win32 file images can contain a file version resource that stores product and version information about the file. The version number is actually four 16-bit values typically displayed using dot notation (such as 4.0.9.4566). You can use this information when determining whether one file is newer or older than another.

This code implements the GetVersionInfo procedure in a standard BAS module. Pass the name of a file to GetVersionInfo, and a dot-formatted string of the version number returns, if available, or "N/A" returns if the file does not contain a version resource:

```
Private Type VS_FIXEDFILEINFO
   dwSignature As Long
   dwStrucVersion As Long
   dwFileVersionMSl As Integer
   dwFileVersionMSh As Integer
   dwFileVersionLSl As Integer
   dwFileVersionLSh As Integer
   dwProductVersionMSl As Integer
   dwProductVersionMSh As Integer
   dwProductVersionLSl As Integer
   dwProductVersionLSh As Integer
   dwFileFlagsMask As Long
   dwFileFlags As Long
   dwFileOS As Long
   dwFileType As Long
   dwFileSubtype As Long
   dwFileDateMS As Long
   dwFileDateLS As Long
End Type
Private Declare Function GetFileVersionInfo _
   Lib "Version.dll" Alias _
   "GetFileVersionInfoA" (ByVal lptstrFilename _
   As String, ByVal dwHandle As Long, ByVal _
   dwLen As Long, lpData As Any) As Long
Private Declare Function _
   GetFileVersionInfoSize Lib "Version.dll" _
   Alias "GetFileVersionInfoSizeA" (ByVal _
   lptstrFilename As String, lpdwHandle As _
   Long) As Long
Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" (dest As Any, src As _
   Long, ByVal length As Long)
Private Declare Function VerQueryValue Lib _
   "Version.dll" Alias "VerQueryValueA" _
   (pBlock As Any, ByVal lpSubBlock As String, _
   lplpBuffer As Any, puLen As Long) As Long
Public Function GetVersionInfo(ByVal sFile As _
   String) As String
   Dim lDummy As Long
   Dim sBuffer() As Byte
   Dim lBufferLen As Long, lVerPointer As Long
   Dim lVerBufferLen As Long
   Dim udtVerBuffer As VS_FIXEDFILEINFO
   ' Default return value
   GetVersionInfo = "N/A"
   ' Attempt to retrieve version resource
   lBufferLen = GetFileVersionInfoSize(sFile, _
      lDummy)
   If lBufferLen > 0 Then
      ReDim sBuffer(lBufferLen)
      If GetFileVersionInfo(sFile, 0&, _
         lBufferLen, sBuffer(0)) <> 0 Then
         If VerQueryValue(sBuffer(0), _
            "\", lVerPointer, lVerBufferLen) _
            <> 0 Then
            CopyMemory udtVerBuffer, ByVal _
               lVerPointer, Len(udtVerBuffer)
            With udtVerBuffer
               GetVersionInfo = _
                  .dwFileVersionMSh & "." & _
                  .dwFileVersionMSl & "." & _
                  .dwFileVersionLSh & "." & _
                  .dwFileVersionLSl
            End With
         End If
      End If
   End If
End Function
```

—James D. Murray, Huntington Beach, California

VB5, VB6
Level: Beginning

Enforce Case With Enums

Enumerated constants are great, but they have a quirk that's a bit obnoxious: They don't retain their capitalization in the Integrated Development Environment (IDE), which a lot of folks use to provide visual feedback that they haven't misspelled the constant name. You can fool the IDE into retaining the capitalization by also declaring the Enums as public variables and surrounding the declarations with "#If False...#End If" compiler directives so they won't be compiled:

```
Public Enum MyEnum
   EnumOne=1
   EnumTwo
   EnumThree
End Enum
#If False Then
   Public EnumOne
   Public EnumTwo
   Public EnumThree
#End If
```

—Barry Garvin, Georgetown, Massachusetts

VB3 and up
Level: Intermediate

Tokenize Your Strings
I find the strtoken function in C quite powerful. This function strips tokens off a string-based delimiter in the string. The function returns the token and stripped string so that the next call to strtoken returns the next token. This sequence continues until there are no more tokens, when the string would be empty. Here's a function in VB that provides the same functionality:

```
Public Function StrGetToken(ByRef psString As _
   String, ByVal psDelim As String) As String
   Dim nPos As Long
   Dim sToken As String
   On Error GoTo ERROR_Handler
   sToken = psString
   ' Check for empty string
   If (Len(psString)) Then
      ' Check for position of delimeter
      nPos = InStr(psString, psDelim)
      ' If delimeter is found, strip off token
      If (nPos > 0) Then
         sToken = Left$(psString, nPos - 1)
         ' Strip token and delimiter from
         ' string passed in
         psString = Mid$(psString, nPos + Len(psDelim))
      Else
         ' No delimiter found, set string
         ' passed into an empty string
         psString = ""
      End If
   End If
   StrGetToken = sToken
   Exit Function
ERROR_Handler:
   StrGetToken = ""
End Function
```

Here's an example of how you might use this function:

```
   Dim s As String
   s = "Dim sToken As String"
   Do While Len(s)
      Debug.Print StrGetToken(s, " ")
   Loop
```

—Joe Halfman, received by e-mail

VB4, VB5, VB6
Level: Beginning

Drag Files Into Project Window
Creating a new project in VB6 (and earlier versions) has always been a pain when you have a large library of modules to add in. You can do it a module at a time—the Add File dialog doesn't support multiple select—or you can edit the VBP file manually. Either way is difficult.

But try this: Open VB, then open an Explorer window. Find and highlight several BAS, FRM, or CLS modules, and simply drag them into the Project window in VB. Voilà! No more single-select nightmares. All the files are added to the project instantly.

—Darin Higgins, Fort Worth, Texas

VB6
Level: Beginning

Use StrReverse to Find Last Character Occurrence
When you need to determine the last occurrence of a character within a larger string, you typically use a Mid$ or Instr in a loop. VB6's introduction of the new string reversal function, StrReverse, offers an easier way. A fully qualified filespec might include several backslashes, and you must find the last one to extract only the filename. This routine uses StrReverse to quickly extract only the information you need:

```
Public Function GetFileName(ByVal FileSpec As _
   String) As String
   Dim sRevName As String
   If Len(FileSpec) Then
      sRevName = StrReverse(FileSpec)
      GetFileName = StrReverse(Left$( _
         sRevName, InStr(1, sRevName, "\") - 1))
   End If
End Function
```

Using the same logic, you can strip the filename from the path to return the full path alone:

```
Public Function GetPath(ByVal FileSpec As _
   String) As String
   Dim sRevName As String
   Dim sPathName As String
   If Len(FileSpec) Then
      sRevName = StrReverse(FileSpec)
      sPathName = StrReverse(Right$(sRevName, _
         Len(sRevName) - InStr(sRevName, "\")))
      If Right$(sPathName, 1) = ":" Then
         ' Root directory, add backslash
         sPathName = sPathName & "\"
      End If
      GetPath = sPathName
   End If
End Function
```

—Balaraman M. Sriram, Richardson, Texas

VB3 and up
Level: Beginning

Clear Structure Data With One Assignment
User-defined types are useful when you need to store structured data that has no specific behavior. (If you have associated behavior, you should encapsulate the data in its own class.) Here's a quick way to clear user-defined type variables without setting each subvariable:

```
' A user-defined type:
Private Type udtSomeType
   SubVariableOne As Integer
   SubVariableTwo As String
   SubVariableThree As Long
End Type
' A couple of class-level user-defined
' type variables:
Private TypeVariableOne As udtSomeType
Private TypeVariableTwo As udtSomeType
' A method in the class:
Private Sub ResetData()
   Dim CleanTypeVariable As udtSomeType
   TypeVariableOne = CleanTypeVariable
   TypeVariableTwo = CleanTypeVariable
End Sub
```

This method is especially convenient when the user-defined type has numerous subvariables.

—Dave Doknjas, Surrey, British Columbia, Canada

VB4 32, VB5, VB6
Level: Intermediate

Create Captionless MDI Forms

Sometimes you might want an MDI form without any captions or buttons, such as in a game or acting as a background parent object. You can create a captionless MDI form easily with a few API calls. The system menu keys, such as Alt-F4 and Alt-Space, still work, even though the system menu icon is invisible. If you don't want the system menu to pop up, then uncomment the Xor WS_SYSMENU line below. Note that Alt-F4 still works, however:

```
Private Declare Function GetWindowLong Lib _
   "user32" Alias "GetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long) As Long
Private Declare Function SetWindowLong Lib _
   "user32" Alias "SetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long, ByVal _
   dwNewLong As Long) As Long
Private Const GWL_STYLE = (-16)
Private Const WS_CAPTION = &HC00000
Private Const WS_SYSMENU = &H80000
Private Sub MDIForm_Load()
   Dim lStyle As Long
   lStyle = GetWindowLong(Me.hWnd, GWL_STYLE)
   lStyle = lStyle Xor WS_CAPTION
   ' Uncomment to remove system menu access:
   ' lStyle = lStyle Xor WS_SYSMENU
   Call SetWindowLong (Me.hWnd, GWL_STYLE, _
      lStyle)
   ' *** Required by VB4 32, not VB5 or later:
   ' Form1.Show
   ' Unload Form1
End Sub
```

If you're still using VB4, you must load a child form before the caption bar will disappear. If you don't want to show a child loaded initially, then load and immediately unload a small MDI child form in the MDI's Form_Load( ) event.

—Matt Hart, Tulsa, Oklahoma

VB3 and up
Level: Beginning

Quick and Easy Queue

Listboxes provide suitable functionality to act as a quick queue. Create a listbox named ListMyQueue. Use this code to add to your Queue:

```
Public Sub Enqueue(StringToAdd As String)
   If Len(String_to_Add) > 0 Then
      ParentForm.ListMyQueue.AddItem _
         StringToAdd
   End If
End Sub
```

Use this code to retrieve from your Queue:

```
Public Function Dequeue() As Variant
   If ParentForm.ListQueue.ListCount > 0 Then
      Dequeue = ParentForm.ListQueue.List(0)
      Parent_Form.ListQueue.RemoveItem (0)
   Else
      MsgBox "Queue is Empty"
   End If
End Function
```

—Eric Robuck, Lyon Station, Pennsylvania

VB4 32, VB5, VB6
Level: Intermediate

Quick Recordset Copy to Excel Workbook

One of the most common things VB programmers do with Excel is load data into an Excel worksheet from a Recordset object. The method I see used most often to do this is looping through each column and row of the recordset, placing the values individually into the corresponding cells on the Excel worksheet. However, a far faster and more efficient way takes advantage of the little-known CopyFromRecordset method of the Excel Range object.

All you need to use this method is an object reference to the top-left cell of the destination range. Then invoke the CopyFromRecordset method for this Range object, passing it the Recordset object you want to load into the worksheet.

Here's a simple example, which requires that your project reference the Data Access Objects (DAO) and Excel object libraries:

```
Sub Main()
   Dim db As DAO.Database
   Dim rs As DAO.Recordset
   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   ' Open the recordset.
   Set db = DBEngine.Workspaces(0). _
      OpenDatabase("D:\db1.mdb")
   Set rs = db.OpenRecordset("SELECT _
      * FROM MyTable")
   ' Open the destination Excel workbook.
   Set xlApp = New Excel.Application
   Set xlBook = xlApp.Workbooks. _
      Open("D:\Book1.xls")
   ' This is all it takes to copy the contents
   ' of the recordset into the first worksheet
   ' of Book1.xls.
xlBook.Worksheets(1).Range("A1"). _
   CopyFromRecordset rs
   ' Clean up everything.
   xlBook.Save
   xlBook.Close False
   xlApp.Quit
   rs.Close
   db.Close
   Set xlBook = Nothing
   Set xlApp = Nothing
   Set rs = Nothing
   Set db = Nothing
End Sub
```

Excel 97 supports only plain vanilla DAO recordsets for this operation (not including ODBCDirect recordsets). However, Excel 2000 has added support for all flavors of DAO and ActiveX Data Objects (ADO) recordsets, making this a powerful tool for Office 2000 programming.

—Rob Bovey, Edmonds, Washington

VB3 and up
Level: Beginning

Return Fractional Part of a Number

No native VB function returns a fractional part of a decimal number. However, by subtracting the whole portion, obtained with Fix, from the original value, you can derive the fractional portion easily. Return the absolute value of this calculation to remove unwanted negatives:

```
Public Function Frac(ByVal Value As Double) _
   As Double
   Frac = Abs(Value - Fix(Value))
End Function
```

—William Powell Jr., Lanham, Maryland

VB5, VB6
Level: Advanced

Shorten Long Path to Fit Narrow Area

When you have to display a long file path in a limited amount of space, you can use a Shell Light Weight API to do the job for you. Create a form with two textboxes (txtShortPath and txtLongPath) and a command button. This code demonstrates the call to PathCompactPath, which is Unicode only:

```
Private Declare Function PathCompactPathW _
   Lib "shlwapi.dll" (ByVal hDC As Long, _
   ByVal lpszPath As Long, ByVal dx As Long) _
   As Boolean
Private Declare Function GetDC Lib "user32" _
   (ByVal hWnd As Long) As Long
Private Declare Function ReleaseDC Lib _
   "user32" ByVal hWnd As Long, ByVal hDC As _
   Long) As Long
Private Sub Command1_Click()
   Dim hDC     As Long
   Dim sPath   As String
   Dim nWidth  As Long
   Const MAX_PATH As Long = 260
   ' txtLongPath should contain a long path
   ' to a file, txtShortPath should be narrow
   ' enough that it does not normally display
   ' the long path.
   hDC = GetDC(txtShortPath.hWnd)
   sPath = Left$(txtLongPath.Text & _
      vbNullChar & Space$(MAX_PATH), MAX_PATH)
   nWidth = Me.ScaleX(txtShortPath.Width, _
      Me.ScaleMode, vbPixels)
   If PathCompactPathW(hDC, StrPtr(sPath), _
      nWidth) Then
      txtShortPath.Text = Left(sPath, _
         InStr(sPath, vbNullChar) - 1)
   Else
      ' False means it could not be made
      ' that short or the call failed
      txtShortPath.Text = "Error"
   End If
   ReleaseDC txtShortPath.hWnd, hDC
End Sub
```

You can also achieve similar functionality with a DrawText Windows API call using the DT_PATH_ELLIPSIS and DT_MODIFYSTRING constants. PathCompactPath requires shlwapi.dll version 4.71 or higher, which ships with Internet Explorer 4. See http://msdn.microsoft.com/library/psdk/shellcc/shell/versions.htm for versioning details.

—Phil Fresle, Corfe Mullen, England

VB4, VB5, VB6
Level: Beginning

Iterate Control Arrays Without Error

Control arrays are odd beasts in that they can have missing elements. The simplest way to iterate a control array uses this method:

```
Dim i As Integer
For i = Text1.LBound To Text1.UBound ...
```

However, if you have holes in your array, that method tosses an error. To avoid that eventuality, treat the array like a collection:

```
Dim txt As TextBox
For Each txt In Text1
   txt.Text = "Hello, World!, My Index is " & txt.Index
Next txt
```

—Guy Dafny, Tel Aviv, Israel

VB5, VB6
Level: Advanced

Generate Relative Path Between Folders

You can use a Shell Light Weight API to generate a relative path by using this code:

```
Private Declare Function PathRelativePathToW _
   Lib "shlwapi.dll" (ByVal pszPath As Long, _
   ByVal pszFrom As Long, ByVal dwAttrFrom As _
   Long, ByVal pszTo As Long, ByVal dwAttrTo _
   As Long) As Boolean
Private Function GetRelativePath( _
   sRelativePath As String, ByVal sPathFrom _
   As String, ByVal sPathTo As String) As _
   Boolean
   Dim bResult As Boolean
   Const MAX_PATH As Long = 260
   sRelativePath = Space(MAX_PATH)
   ' Set "dwAttr..." to vbDirectory for
   ' directories, 0 for files
   bResult = PathRelativePathToW(StrPtr _
      (sRelativePath), StrPtr(sPathFrom), _
      vbDirectory, StrPtr(sPathTo), 0)
   If bResult Then
      sRelativePath = Left(sRelativePath, _
         InStr(sRelativePath, vbNullChar) - 1)
   Else
      sRelativePath = ""
   End If
   GetRelativePath = bResult
End Function
Private Sub Command1_Click()
   Dim sRelative As String
   ' txtFromPath should contain the directory
   ' path to go from, txtToPath should contain
   ' the file path to go to.
   ' txtRelativePath will contain the result
   If GetRelativePath(sRelative, _
      txtFromPath.Text, txtToPath.Text) Then
      txtRelativePath.Text = sRelative
   Else
      txtRelativePath.Text = "Error"
   End If
End Sub
```

PathRelativePathTo requires shlwapi.dll version 4.71 or higher, which ships with Internet Explorer 4. See http://msdn.microsoft.com/library/psdk/shellcc/shell/versions.htm for versioning details.

—Phil Fresle, Corfe Mullen, England

VB3 and up
Level: Beginning

Obtain Regional Decimal Character Without API

Use this function to read a number decimal symbol from regional settings:

```
Sub Form_Load()
   Dim DecS   As String
   DecS = ReadDecimalSymbol()
End Sub
Function ReadDecimalSymbol() As String
   ReadDecimalSymbol = Mid$(CStr(1.1), 2, 1)
End Function
```

—Gianfranco Callino, Verona, Italy

VB4 32, VB5, VB6
Level: Intermediate

Retrieve File Description

This routine takes a passed filename as an argument and generates a description for it. It returns the same string as Windows Explorer does when it has been set to Details view.

For example, if you pass the file c:\windows\win.com to the routine, it returns the string "MS-DOS Application." For files it can't describe, the routine returns a generic message of "{filetype} File." If the file passed doesn't exist, it returns "Unknown File," but you can change this easily. This code is especially useful for telling beginning users what type of file they're dealing with:

```
Private Declare Function SHGetFileInfo Lib _
   "shell32.dll" Alias "SHGetFileInfoA" _
   (ByVal pszPath As String, ByVal _
   dwFileAttributes As Long, psfi As _
   SHFILEINFO, ByVal cbFileInfo As Long, _
   ByVal uFlags As Long) As Long
Private Const SHGFI_TYPENAME = &H400
Private Const MAX_PATH = 260
Private Type SHFILEINFO
   hIcon As Long
   iIcon As Long
   dwAttributes As Long
   szDisplayName As String * MAX_PATH
   szTypeName As String * 80
End Type
Public Function GetFileType(lpStrFile As _
   String) As String
   Dim sfi As SHFILEINFO
   ' Make API Call to fill structure with
   ' information
   If SHGetFileInfo(lpStrFile, 0, sfi, _
      Len(sfi), SHGFI_TYPENAME) Then
      ' Return filetype string
      GetFileType = Left$(sfi.szTypeName, _
         InStr(sfi.szTypeName, vbNullChar) - 1)
   Else
      ' If failed then return "Unknown File"
      GetFileType = "Unknown File"
   End If
End Function
```

—Adam Lanzafame, Adelaide, South Australia, Australia

VB6
Level: Beginning

Employ Radio Buttons in a ListView

A simple piece of code can force the checkboxes in a ListView control to behave like radio buttons. Set the ListView's Checkboxes property to True and place this code in its ItemCheck event procedure:

```
Private Sub ListView1_ItemCheck(ByVal Item _
   As MSComctlLib.ListItem)
   Dim li As MSComctlLib.ListItem
   For Each li In ListView1.ListItems
      If li.Checked = True Then
         If li <> Item Then li.Checked = False
      End If
   Next li
End Sub
```

Each time the user checks one list item, any that were checked previously become unchecked.

—James D. Murray, Huntington Beach, California

VB4 32, VB5, VB6
Level: Intermediate

Authenticate Component Usage

Bundling functionality and program logic into an ActiveX DLL is an excellent form of encapsulation. But even when you expose functionality to your client application, you don't need to allow unrestricted access to all of your public functions. Use this simple mechanism to secure your proprietary functions from unauthorized access.

Create a private global variable, g_Authorized, of type Boolean to hold the authorization state for your DLL. When the DLL loads, g_Authorized is initialized to False. Each function (or sub) that you wish to protect should first check the value of g_Authorized before proceeding. If g_Authorized = False, then raise a runtime error advising the user that the function call is not authorized. If g_Authorized = True, then execute the function. For example, here is a snippet from one of the encryption routines. You use encryption to keep the data private, so you want to protect the encyption function itself from unauthorized access:

```
Public Function Encrypt(PlainText As String, _
   CipherType As axCipherType, Optional _
   ByVal Key As Long = 0) As String
   Dim iX As Long
   Dim iAscii As Integer
   Dim CipherText As String
   Dim StringLen As Long
   If Not g_Authenticated Then
      Err.Raise vbObjectError, "Encrypt", _
         "Application is not authorized " & _
         "to use this function"
   End If
```

The client application must call this public function to set the state of the DLL to Authorized (g_Authorized = True):

```
Public Sub Authenticate(Code As Variant)
   If Code = "asd93d,ssd" Then
      g_Authenticated = True
   End If
End Sub
```

Passing the code parameter as a Variant makes it more secure because a potential hacker would have no idea what sort of data the expected authentication code is.

This is the basic methodology for DLL protection. Actually, you could employ much more secure algorithms. You could derive the code from any number of potential values, such as the current system date/time, hard disk free space, or any other checkable value. Only your own applications would know the correct algorithm, so they would be the only applications on the client PC capable of authenticating the DLL for their use. Such a code would be more secure from a hack attack because it would actually change from minute to minute or machine to machine.

—Joseph Geretz, Monsey, New York

VB4 32, VB5, VB6
Level: Beginning

Dump Resource Strings to Text File

The Resource Editor add-in is useful to add resource file text entries, but it doesn't provide any facility to print the current contents. You can add this routine to a project module to generate the desired documentation:

```
Public Sub DumpResStrings(Start As Long, _
   Finish As Long, FileSpec As String)
   Dim hFile As Long
   Dim sText As String
   Dim i As Long
   On Error Resume Next
   hFile = FreeFile
   Open FileSpec For Output As #hFile
   For i = Start To Finish
      sText = LoadResString(i)
      If Err.Number = 0 Then
         Print #hFile, i & vbTab & sText
      Else
         Err.Clear
      End If
   Next i
   Close #hFile
End Sub
```

To extract the desired range of resource file numbers into a text file, open the Immediate window and call DumpResStrings, passing appropriate parameters:

```
call dumpresstrings(1,2999,"resdat.txt")
```

When resource file documentation is complete, set the function to Private for standard application development.

—Trevor Marr, Chessington, Surrey, England

VB3 and up
Level: Intermediate

Short-Circuit Your Code

Be aware that VB doesn't short-circuit Boolean expressions, unlike programming languages such as C. Short-circuit evaluation means to evaluate only as much of an expression as is absolutely necessary to determine the Boolean value. For example, (A And B And C) is certain to be False if A is False, so you can ignore B and C. Similarly, (A Or B Or C) where B is True is certain to be True, so you can ignore C. VB doesn't behave this way—it evaluates the entire expression. So this If statement results in a runtime error if oItem has not been assigned a valid object value:

```
Dim oItem As Object
If Not (oItem Is Nothing) And (oItem.Text _
   <> "") Then
   ' Do something
End If

Instead, write:

If Not (oItem Is Nothing) Then
   If (oItem.Text <> "") Then
      ' Do something
   End If
End If
```

This difference is especially important if the Boolean expression includes side effects, such as a function modifying a local static variable or module scope variable.

—John Calvert, Ottawa, Ontario, Canada

VB4 32, VB5, VB6
Level: Beginning

Convert Short Filename Into Long Filename

You can use the Dir() function to return a long filename, but the return does not include path information. By parsing a given short path/filename into its constituent directories, you can use the Dir() function to build a long path/filename with 32-bit versions of VB, without the assistance of APIs:

```
Public Function GetLongFilename(ByVal _
   sShortName As String) As String
   Dim sLongName As String
   Dim sTemp As String
   Dim iSlashPos As Integer
   ' Add \ to short name to prevent Instr from failing
   sShortName = sShortName & "\"
   ' Start from 4 to ignore the "[Drive
   ' Letter]:\" characters
   iSlashPos = InStr(4, sShortName, "\")
   ' Pull out each string between \ character for conversion
   Do While iSlashPos
      sTemp = Dir(Left$(sShortName, _
         iSlashPos - 1), vbNormal Or vbHidden _
         Or vbSystem Or vbDirectory)
      If sTemp = "" Then
         ' Error 52 - Bad File Name or Number
         GetLongFilename = ""
         Exit Function
      End If
      sLongName = sLongName & "\" & sTemp
      iSlashPos = InStr(iSlashPos + 1, sShortName, "\")
   Loop
   ' Prefix with the drive letter
   GetLongFilename = Left$(sShortName, 2) & sLongName
End Function
'From any place, add this line
GetLongFilename("C:\PROGRA~1\COMMON~1")
```

This function, as written, expects a standard fully qualified, drive-based filespec.

—Alex Leyfman, Brooklyn, New York

VB5, VB6
Level: Intermediate

Use Shell Functions in Browser Control

When you're playing with the Web browser control (Microsoft Internet Controls), you can browse some of the objects that come with it by pressing F2 and selecting the Shell object of SHDocVwCtl. You'll see many cool routines such as FindComputer, FileRun, Explore, and ShutdownWindows, but the two most fascinating are MinimizeAll and UndoMinimizeALL. To get the code to work, open a VB project, open Components (Ctrl-T), check Microsoft Internet Controls, put two command buttons on your form, and include this code:

```
' Minimize All
Private Sub Command1_Click()
   Dim IShell As New Shell
   IShell.MinimizeAll
End Sub
' Undo Minimize All
Private Sub Command2_Click()
   Dim IShell As New Shell
   IShell.UndoMinimizeALL
End Sub
```

—Doug Weems, Locust Grove, Georgia

VB5, VB6
Level: Advanced

Unhook Subclassing When Windows is Ready
Don't unhook your Windows procedures from Form_Unload when subclassing forms. When you subclass forms, the hook is often set during Form_Load with code like this:

```
OriginalProc = SetWindowLong Me.hWnd, _
   GWL_WNDPROC, AddressOf MyWindowProc
```

A common mistake is forgetting to put the corresponding unhook call in your Form_Unload event:

```
SetWindowLong Me.hWnd, GWL_WNDPROC, _
   AddressOf OriginalProc
```

If you forget to reinstate the old procedure in your Form_Unload event, it's bye-bye VB. Instead, add this code within your subclassing procedure:

```
Select Case Msg
   Case WM_NCDESTROY
      If OriginalProc <> 0 Then
         Call SetWindowLong(hWnd, _
            GWL_WNDPROC, OriginalProc)
         OriginalProc=0
      End If
   Case ...
```

This code restores the original procedure automatically when the window is destroyed. To make it even safer, place all your subclassing code in a separate DLL and debug your subclassed forms without worrying about the Integrated Development Environment (IDE) crashing. You can always move the code back to your EXE when it's fully debugged.

—Simon Bryan, Newbury, Berkshire, England

VB4 32, VB5, VB6
Level: Beginning

Sort and Reverse-Sort a ListView
This routine performs the standard column sorting on a ListView control found in many commercial applications, such as Windows Explorer and Outlook. Using this routine, the ListView sorts itself automatically whenever the user clicks on a column. Clicking on the same column toggles the sort order between ascending and descending order. Call this routine from the ListView control's ColumnClick event procedure by passing both a reference to the ListView and the ColumnHeader reference passed to the original event:

```
Public Sub ListView_ColumnClick(ByRef _
   MyListView As ListView, ByVal ColumnHeader _
   As ColumnHeader)
   With MyListView
      .Sorted = False
      If .SortKey <> ColumnHeader.Index - _
         1 Then
         .SortKey = ColumnHeader.Index - 1
         .SortOrder = lvwAscending
      Else
         If .SortOrder = lvwAscending Then
            .SortOrder = lvwDescending
         Else
            .SortOrder = lvwAscending
         End If
      End If
      .Sorted = True
   End With
End Sub
```

—Jim Pragit, Glen Ellyn, Illinois

VB4, VB5, VB6
Level: Beginning

Test for Alpha Characters Only
Although VB has an IsNumeric function, it has no IsAlpha function. Use this routine whenever you want to determine whether a character or string of characters is alphabetic (A-Z or a-z). Add Case conditions for other characters you're willing to allow, such as hyphens, apostrophes, or whatever you consider legal:

```
Public Function IsAlpha(ByVal MyString As _
   String) As Boolean
   Dim i As Long
   ' Assume success
   IsAlpha = True
   ' Check to be sure
   For i = 1 To Len(MyString)
      Select Case Asc(Mid$(MyString, i, 1))
         Case vbKeyA To vbKeyZ
         Case (vbKeyA + 32) To (vbKeyZ + 32)
         Case vbKeySpace
         ' Add more tests to suit
         Case Else
            IsAlpha = False
            Exit For
      End Select
   Next i
End Function
```

—Jim Pragit, Glen Ellyn, Illinois

VB3 and up
Level: Beginning

Test for Alpha Characters Only, Part II
I have a much simpler form for the IsAlphaNum function:

```
Public Function IsAlphaNum(ByVal sString _
   As String) As Boolean
   If Not sString Like "*[!0-9A-Za-z]*" _
      Then IsAlphaNum = True
End Function
```

You can modify this function for other conditions. Simply put the acceptable characters—such as a space, hyphen, or dot—into the square brackets.

—Rick Rothstein, Trenton, New Jersey

VB3 and up
Level: Beginning

Test for Alpha Characters Only, Part III
Traditional testing for alphabetic characters—for example, to restrict characters that can be entered in a textbox—uses the ASCII value of the keypress:

```
If KeyAscii >=65 And KeyAscii < 113 Then
```

However, this test doesn't allow for international code pages, which might include characters with an ASCII code higher than 113. A more logical definition of an alphabetical character is one that has a distinct upper and lowercase. To test whether something is alphabetic, use this code:

```
' International IsAlpha character test.
' Returns true if the input letter is
' alphabetical in any code page or language.
Public Function IsAlphaIntl(sChar As String) _
   As Boolean
   IsAlphaIntl = Not (UCase$(sChar) = LCase$(sChar))
End Function
```

—Duncan Jones, Caistor, Lincolnshire, England

VB3 and up
Level: Beginning

Float Buttons Over Tab Panels
When using the SSTab control to build a tabbed dialog, I often find the design more pleasing when the OK/Cancel buttons or other controls appear on every tab. It's easy to place a similar control on every tab page, but with more than a few tabs, this process can be resource-consuming and potentially more difficult to debug.

A better way to do this is to use the natural Z-order of controls. If you draw and size the controls you want to be "floating" outside the SSTab control, then use the Bring to Front command on the Format bar, or right-click on them and select Bring to Front. The controls will appear above the tab control. Simply move them to a position that's empty in every tab, and you have a floating control.

Lightweight controls, such as labels and image boxes, do not draw above the tabs, so test the design carefully.

—Michael Lewis, Chiang Mai, Thailand

VB4, VB5, VB6, VBA
Level: Beginning

Perform String "Math" With Leading Zeroes
In an introduction to VB's String datatype [CS 101, "The String's the Thing," *VBPJ* July 1999] by Ron Schwarz, the author uses this example code:

```
Dim Foo As String
Dim Bar As String
Foo = "10"
Bar = "20"
Print Foo + Bar
Print Foo + Bar + 1
Print Foo + Bar & 1
```

The code produces this output:

```
1020
1021
10201
```

However, you could use this code:

```
Foo = "001"
Print Foo & 1
Print Foo + 1
```

that would produce this output:

```
0011
2
```

Would you expect the last result from the string datatype? Developers often deal with strings that start with single or multiple zeroes, such as account or check numbers. The way VB treats strings with leading zeroes appears more like a bug than an advantage. Here's a simple line of code that preserves the leading zeroes:

```
Print Format$(Val(Foo) + 1, String$(Len(Foo), "0"))
```

Now you'll see this:

```
002
```

—Edward Vakhler, Brooklyn, New York

VB3 and up
Level: Beginning

Restore Properties as They Were
If you don't want the user to do something while other processes are in progress, you can disable certain controls, especially if the intermediate processing requires user interactions. I have often seen this:

```
cmdButton.Enabled = False 'Don't let them do it
   'Do stuff
cmdButton.Enabled = True 'Now it's OK
```

The problem is that perhaps you shouldn't really enable cmdButton because it might have been disabled before your routine was called.

If you're privy to all the requirements of when cmdButton is enabled or disabled, you can centralize those and call the centralized routine when you're ready. But if your section of the project isn't in charge of the button, the easiest and most polite thing to do is set it back the way it was:

```
Dim blnWasEnabled As Boolean
blnWasEnabled = cmdButton.Enabled
frmParent.cmdButton.Enabled = False
   'Don't let them do it
   'Do stuff
frmParent.cmdButton.Enabled = blnWasEnabled
   'Set it back
```

You can also use this logic to control visibility, AutoRedraw properties, or anything else that might be set from multiple places in the code. It works best within a single procedure and where you aren't calling other routines that also might affect the property you're setting in the meantime.

—Paul Backstrom, Kirkland, Washington

VB4 32, VB5, VB6
Level: Intermediate

Fill Combo With Available Drive Letters
To create a drop-down control with a list of used or unused drive letters, place two ComboBox controls on a form, named Combo1 and Combo2, and include this code to initialize the lists:

```
Private Declare Function GetLogicalDrives Lib _
   "kernel32" () As Long
Private Sub Form_Load()
   FillCombo Combo1, True
   FillCombo Combo2, False
End Sub
Private Sub FillCombo(cbo As ComboBox, _
   ByVal bUsed As Boolean)
   Dim DriveNum As Long
   cbo.Clear
   For DriveNum = 0 To 25
      If CBool(GetLogicalDrives And (2 ^ _
         DriveNum)) = bUsed Then
         cbo.AddItem Chr$(Asc("A") + _
            DriveNum) & ":"
      End If
   Next DriveNum
End Sub
```

—Brian Dial, Decatur, Alabama

VB5, VB6
Level: Beginning

Open Your VB Projects With a Clean Slate

If you're like me, you hate all the clutter of open form, class, and module windows when you open your VB projects in the Integrated Development Environment (IDE). Here's a simple workaround to let you start your project with a clean workspace.

Edit the accompanying VB Workspace file for your project. It has the same name as your project file, but with a VBW file extension. Delete all the lines in this file and save it. Now make this file read-only by right-clicking on the file, choosing Properties, then selecting the read-only checkbox.

Whenever you save your project from then on, VB won't update this file because it is read-only, and it won't complain. Each time you open your project, your workspace will start fresh with no clutter. If for any reason you want to revert to the old way, simply change the read-only flag back.

—Richard Edwards, Belleville, Ontario, Canada

VB5, VB6
Level: Advanced

☆☆☆☆☆ Five Star Tip

Load a Bitmap Resource From a DLL

You can employ any DLL's bitmap resources in VB using this LoadPicture function. You need to set a reference to OLE Automation:

```
Private Type GUID
   Data1 As Long
   Data2 As Integer
   Data3 As Integer
   Data4(7) As Byte
End Type
Private Type PicBmp
   Size As Long
   Type As Long
   hBmp As Long
   hPal As Long
   Reserved As Long
End Type
Private Declare Function _
   OleCreatePictureIndirect Lib _
   "olepro32.dll" (PicDesc As PicBmp, RefIID _
   As GUID, ByVal fPictureOwnsHandle As Long, _
   IPic As IPicture) As Long
Private Declare Function LoadBitmap Lib _
   "user32" Alias "LoadBitmapA" (ByVal _
   hInstance As Long, ByVal lpBitmapID As _
   Long) As Long
Private Declare Function DeleteObject Lib _
   "gdi32" (ByVal hObject As Long) As Long
Private Declare Function LoadLibrary Lib _
   "kernel32" Alias "LoadLibraryA" (ByVal _
   lpLibFileName As String) As Long
Private Declare Function FreeLibrary Lib _
   "kernel32" (ByVal hLibModule As Long) _
   As Long
Public Function LoadPicture(sResourceFileName _
   As String, lResourceId As Long) As Picture
   Dim hInst As Long
   Dim hBmp As Long
   Dim Pic As PicBmp

   Dim IPic As IPicture
   Dim IID_IDispatch As GUID
   Dim lRC As Long
   hInst = LoadLibrary(sResourceFileName)
   If hInst <> 0 Then
      hBmp = LoadBitmap(hInst, lResourceId)
      If hBmp <> 0 Then
         IID_IDispatch.Data1 = &H20400
         IID_IDispatch.Data4(0) = &HC0
         IID_IDispatch.Data4(7) = &H46
         Pic.Size = Len(Pic)
         Pic.Type = vbPicTypeBitmap
         Pic.hBmp = hBmp
         Pic.hPal = 0
         lRC = OleCreatePictureIndirect(Pic, _
            IID_IDispatch, 1, IPic)
         If lRC = 0 Then
            Set LoadPicture = IPic
            Set IPic = Nothing
         Else
            Call DeleteObject(hBmp)
         End If
      End If
      Call FreeLibrary(hInst)
      hInst = 0
   End If
End Function
Private Sub Form_Load()
   ' Try ID 130 in Win98, or 131 in NT
   ' to see the Windows logo...
   Set Me.Picture = _
      LoadPicture("shell32.dll", 130)
End Sub
```

—Michael Hill, Northridge, California

VB4 32, VB5, VB6
Level: Intermediate

Add Incremental Search to a Combo Box

As the user types into a drop-down combo box, he or she passes keystrokes to the ComboIncrementalSearch routine, which then searches the combo box's data for the best match:

```
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
Private Const CB_FINDSTRING = &H14C
Private Sub Combo1_KeyPress(KeyAscii _
   As Integer)
   Call ComboIncrementalSearch(Combo1, _
      KeyAscii)
End Sub
Public Sub ComboIncrementalSearch(cbo As _
   ComboBox, KeyAscii As Integer)
   Static dTimerLast As Double
   Static sSearch As String
   Static hWndLast As Long
   Dim nRet As Long
   Const MAX_KEYPRESS_TIME = 0.5
   ' Weed out characters that are not scanned
   If (KeyAscii < 32 Or KeyAscii > 127) _
      Then Exit Sub
   If (Timer - dTimerLast) < _
      MAX_KEYPRESS_TIME And hWndLast = _
      cbo.hWnd Then
      sSearch = sSearch & Chr$(KeyAscii)
   Else
      sSearch = Chr$(KeyAscii)
      hWndLast = cbo.hWnd
   End If
   ' Search the combo box
   nRet = SendMessage(cbo.hWnd, _
      CB_FINDSTRING, -1, ByVal sSearch)
   If nRet >= 0 Then
      cbo.ListIndex = nRet
   End If
   KeyAscii = 0
   dTimerLast = Timer
End Sub
```

—Michael Hill, Northridge, California

VB3 and up
Level: Intermediate

Generate Normal Distributions of Random Numbers
Visual Basic's Rnd function returns a pseudo-random number between zero and one with uniform distribution. In other words, all numbers between zero and one are equally likely. In many situations, however, the normal distribution, with its familiar bell-shaped curve, is a more suitable distribution.

Values from an approximately normal distribution are easy to generate with this function, which returns a pseudo-random number based on the mean and standard deviation you specify:

```
Public Function RndNormal(sngMean As Single, _
   sngStdDev As Single) As Single
   RndNormal = sngMean + (sngStdDev * (Sqr(-2 _
      * Log(Rnd)) * Cos(6.28 * Rnd)))
End Function
```

Because this function relies on the Rnd function, you can initialize this function much as you would Rnd itself. That is, executing the Randomize statement with a particular number before calling this function generates the same set of normally distributed random numbers each time—assuming Rnd is not called elsewhere. Executing Randomize with no argument generates a different set of normal values each time.

Beware, however, that calling Rnd with an argument of zero before calling this function does not cause this function to return the previous normally distributed value because this function calls Rnd twice. If you need the previous value, store it in a variable for reuse.

—John Ricketts, Willowbrook, Illinois

VB3 and up
Level: Beginning

Force the Short-Circuit
To optimize the If construct, separating testing conditions ("And" operator or "Or" operator) in different expressions can improve the performance. For example, don't use this sort of test:

```
If intNum1 > 10 And intNum2 > 20 Then
...
End If
```

Instead, break the individual parts into separate tests:

```
If intNum1 > 10 Then
   If intNum2 > 20 Then
...
   End If
End If
```

VB evaluates the logical result of all test conditions only after performing each individual test. In the example, the And operator requires both test conditions to be True. If intNum is not greater than 10, why should VB waste time in testing the second expression? Similarly, in an Or case, if the first expression is True, you don't need to evaluate the second expression.

—Jasvinder Kakar, Des Moines, Iowa

VB3 and up
Level: Beginning

Calculate Finishing Time
If you ever wondered how some DOS applications, such as Norton Utilities, can calculate the estimated finishing time for a long process, here's the formula:

```
Estimated = DateAdd("s", (DateDiff("s", _
   StartTime, Time()) / PercentDone) * 100, StartTime)
```

To see how it works, add a form with three label controls and one command button. Add this code in the Command button Click event:

```
Private Sub Command1_Click()
   Dim Start As Double
   Dim i As Long
   Dim lMax As Long
   lMax = 300000
   Const fmt As String = "hh:mm:ss ampm"
   Start = InitEstTime()
   Label1.Caption = Format$(Start,fmt)
   For i = 1 To lMax
      Label2.Caption = Format$(UpdateEstTime( _
         Start, (i / lMax) * 100), fmt)
      DoEvents
   Next i
   Label3.Caption = Format$(Time, fmt)
End Sub
```

Include these two functions:

```
Public Function UpdateEstTime(ByVal StartTime _
   As Double, ByVal PercentDone As Single)
   ' Avoid dividing by zero error
   If PercentDone <= 0 Then PercentDone = _
      0.000000001
   UpdateEstTime = DateAdd("s", (DateDiff( _
      "s", StartTime, Time()) / PercentDone) _
      * 100, StartTime)
End Function
Public Function InitEstTime() As Double
   InitEstTime = Time()
End Function
```

—Jacques Levy, Clearwater, Florida

VB5, VB6
Level: Beginning

Customize the IDE Toolbar
The VB Integrated Development Environment (IDE) contains many toolbars and menu functions, and the functions you normally use are usually on different toolbars and menus, so most programmers have several different toolbars showing. You can make your life easier by creating your own toolbar and customizing it with all the toolbar and menu functions you use normally. Then you have only one toolbar showing with all the functions you use regularly.

Right-click on the toolbar area and choose Customize. Choose New and name it My Toolbar. In the Toolbars box on the left, the name of the new toolbar is checked. Uncheck it and check it again. A small Toolbar pops up with nothing on it (look carefully). Now click on the Command tab. You'll see a Categories box and Commands box. Click on the categories you want and the Commands box will contain the functions you can choose from. Drag items from the Commands box and drop them on the new toolbar you created. When you're finished, drag the toolbar to the toolbar area. Again, right-click on the toolbar area and uncheck the other toolbars. Now you have one toolbar with all the functions you want, and not the ones you never use.

—David Bailey, Columbus, Ohio

VB4 32, VB5, VB6
Level: Beginning

Operate on Array of Selected ListItems
The fast way to get multiple selected items from a ListBox control is to send it a LB_GETSELITEMS window message. Here's a simple example that moves items from one ListBox to another ListBox. To test this example, place two ListBox controls (lstFrom and lstTo) and a Command button control (cmdMove) on a form, then copy this code into the form's code editing page:

```
Option Explicit
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
Private Const LB_GETSELCOUNT = &H190
Private Const LB_GETSELITEMS = &H191
Private Sub Form_Load()
   ' Add some items into source list
   lstFrom.AddItem "Matthew So"
   lstFrom.AddItem "Join"
   lstFrom.AddItem "Hello"
   lstFrom.AddItem "Morning"
   lstFrom.AddItem "Apple"
End Sub
Private Sub cmdMove_Click()
   Dim nRet As Long
   Dim nSel() As Long
   Dim i As Long
   nRet = SendMessage(lstFrom.hWnd, _
      LB_GETSELCOUNT, 0, ByVal 0&)
   Me.Caption = CStr(nRet)
   If nRet > 0 Then
      ' Allocate enough memory for the array
      ReDim nSel(0 To nRet - 1)
      ' Get an array of ListIndexes for the
      ' selected items
      nRet = SendMessage(lstFrom.hWnd, _
         LB_GETSELITEMS, lRet, nSel(0))
      ' Start from the end of list to avoid
      ' index change of the source list
      For i = UBound(nSel) To LBound(nSel) _
         Step -1
         ' Copy item from source list to
         ' destination list
         lstTo.AddItem lstFrom.List(nSel(i))
         ' Remove selected item from source
         ' list
         lstFrom.RemoveItem nSel(i)
      Next i
   End If
End Sub
```

The tricks here are to redimension the nSel array using the count of selected items returned by LB_GETSELCOUNT, and to move the selected items starting from the end of the origin ListBox. If you try to move items from the beginning of the ListBox, the ListItems are shifted downward and the saved array of item ListIndexes is no longer valid.

—Matthew So, Hong Kong

VB6
Level: Beginning

Return Empty Arrays Too
With VB6 came the ability to return arrays from functions. Returning an uninitialized array is a problem because there is no easy way—other than error-trapping—to find whether an array has been dimensioned. Also, ReDim myArr(-1) does not work. You can use the Split function to return an empty array—one with no elements and no data—and an LBound of 0 and an UBound of -1. This practice simplifies code for looping through the returned array:

```
Private Function Foo(args...) As String()
   Dim myArr() As String
   ' Initialize array dimensions as 0 to -1
   myArr = Split("")
   If Condition Then
      ReDim myArr(n)
      ' further processing...
   End If
   Foo = myArr
End Function
```

Here, no additional checking is required to use Foo. But omitting the call to Split can lead to a "Subscript out of range" error in a routine that attempts to use Foo's return; when you use Split to establish an empty array, this loop is simply skipped over:

```
   Dim i As Integer
   Dim retArr() As String
   retArr = Foo
   For i = LBound(retArr) To _
      UBound(retArr)
      ' Does not execute as LBound > UBound.
   Next i
```

—Anand Likhite, Orlando, Florida

VB3 and up
Level: Beginning

Handle Errors Within Forms
When you load or show a form, errors don't bubble up. That is, even if the calling procedure has an error handler, and an error occurs in Form_Load, Form_Initialize, or any other form event, processing doesn't transfer to the calling error handler. In this code, Sub Main has an error handler. But when an error occurs in the Form_Load, the error handler isn't called:

```
' Code in a bas module
Sub Main()
   On Error Resume Next
   Load Form1
   ' further processing code ...
End Sub
' Code in Form1
Private Sub Form_Load()
   Dim a As Integer
   a = 1 / 0  ' error is fatal!
End Sub
```

If you check the call stack when the error occurs, you see an entry '<Non-Basic Code>' before the Form_Load. Even though Sub Main is loading Form1, Sub Main is not the direct caller of Form_Load, and that results in this behavior.

—Ravindra Okade, Phoenix, Arizona

VB4 32, VB5
Level: Intermediate

Select a Whole Row in Any ListView
Only in VB6 do the common controls OCX files provide an option to select a full line in a listview. In earlier versions, you can select a ListItem only by clicking on the left-most column. This code allows the user to click anywhere on the line to select and highlight the ListItem:

```
Private Sub ListView1_MouseDown(Button As _
   Integer, Shift As Integer, x As Single, y _
   As Single)
   Dim itm As ListItem
   Dim i As Long
   With ListView1
      Set .DropHighlight = _
         .HitTest(Screen.TwipsPerPixelX * 2, y)
      Set itm = .HitTest(Screen. _
         TwipsPerPixelX * 2, y)
      Set .SelectedItem = itm
      ' Use the following if you have code
      ' to execute on a user click.
      For i = 1 To .ListItems.Count Step 1
         If .ListItems(i).Selected Then
            Call ListView1_ItemClick _
               (.ListItems(i))
            Exit For
         End If
      Next i
   End With
End Sub
```

—Alex Whyte, Sydney, Australia

VB5, VB6
Level: Beginning

Restrict Control Sizing
A few ActiveX controls, such as the Common Dialog Control, don't allow themselves to be resized at design time because they're typically invisible at run time. In these situations, you don't want users to resize the ActiveX controls you create. You can also save some work for users by correctly sizing an ActiveX control whenever they instantiate it on a form. Add this code to the UserControl_Paint event to size your control correctly:

```
Private Sub UserControl_Paint()
   UserControl.Width = <width>
   UserControl.Height = <height>
End Sub
```

The UserControl_Resize event is another option for controls that are visible—and potentially painting frequently—at run time. But code around the inevitable recursion if you choose this other route:

```
Private Sub UserControl_Resize()
   Static Busy As Boolean
   ' Restrict size to desired dimensions
   If Not Busy Then
      Busy = True
      UserControl.Width = <width>
      UserControl.Height = <height>
      Busy = False
   End If
End Sub
```

—Narasimhan Padmanabhan, Bellevue, Washington

VB3 and up
Level: Beginning

Provide Default Value for Null Fields
Be careful when reading field values of a recordset into variables. If the field value is Null and you try to assign it to a variable, the error message "Invalid use of Null" pops up. To prevent this from happening, use this simple function:

```
Public Function CatchNull(vOldValue As _
   Variant, vNewValue As Variant) as Variant
   ' Check for Null
   If IsNull(vOldValue) Then
      ' If Null, use the new value
      CatchNull = vNewValue
   Else
      ' Otherwise, use the existing value
      CatchNull = vOldValue
   End If
End Function
```

Use the function in your recordset traversing loop:

```
   ' Check !EmpNum for Null
   iEmployeeNumber = CatchNull(!EmpNum, 0)
```

If the field value is Null, put 0 for the iEmployeeNumber. You can use this function for any kind of variable. The function is useful to assign any default value for fields containing Null.

—Srinivasa R. Kella, received by e-mail

VB3 and up
Level: Beginning

Find the Error-Generating Line
After trapping an error in debug mode with the usual "On Error Goto ErrorHandlingCode", you might be frustrated that you can't tell exactly what line of code generated the error. If the code you're debugging is not read-only, you can slip in a Resume statement at the beginning of the error handler, use Ctrl-F9 to make it the next line to execute, and press F8 to single step. Then you're positioned at the instruction that caused the error. Don't forget to remove the Resume statement before saving the file.

However, if your code is read-only—which is often the case when you have the code under source control—you cannot temporarily add a Resume statement. In this case, it's handy to modify any error-handling code blocks that terminate with the End Function or End Sub statements by adding an explicit Exit Function or Exit Sub followed by a Resume. This looks a little odd because there is no way to execute the Resume statement, but that's the idea. During normal processing, the Resume statement is never executed, but when you are debugging, it's right there for you to select with Ctrl-F9:

```
Private Sub SampleSub()
   On Error Goto ErrorHandler1
   SubroutineCall1
   SubroutineCall2
   SubroutineCall3
   Exit Sub
ErrorHandler1:
   ...
   <normal error-handling code>
   ...
   Exit Sub
   Resume
End Sub
```

—Stephen E. Coleman, Chantilly, Virginia

VB3 and up
Level: Beginning

Assign Null Fields to Controls Without Error

Here's a great way to assign Null values from a database to your VB controls. Concatenating an empty string to Null produces an empty string. Using the & operator, you can convert all nonstring values to strings using little code:

```
txtName.Text = rs("Name") & ""
```

—Deborah Hammel, Sparks, Maryland

VB4, VB5, VB6
Level: Beginning

Use Objects Directly Within Collections

If you use collections in your apps, you're probably familiar with the For... Each loop to iterate through the collection. But when you want to access only a single element of the collection to perform some temporary calculations or modifications, you might be tempted to do something like this:

```
Set Obj = coll.Item(KeyName)
Obj.Property = something
Call Obj.Method(parameter)
   ...<etc.>
```

Instead, you can simplify your code by using the Collection object in place of Obj. Then you never have to dimension these temporary holders for collection items:

```
coll(KeyName).Property = something
Call coll(KeyName).Method(parameter)
...<etc.>
```

Or you can use this:

```
With coll(KeyName)
   .Property = something
   Call .Method(parameter)
   ...<etc.>
End With
```

—Ian Fenton, Burlington, Ontario, Canada

VB3 and up
Level: Beginning

Use Safer International Conversions

I have problems developing applications for non-English–speaking users because of the noninternational behavior of the Val function. The functions CSng and CDbl provide internationally aware conversions, but they don't work correctly if the argument is empty or contains alpha characters, as can often be the case when converting from nonvalidated TextBox controls. I avoid the errors with this simple function:

```
Private Function CTxtToSng(sInput As String) As Single
   ' CTxtToSng at beginning is worth 0 (zero)
   ' If CSng produces a conversion error, the
   ' value stays 0 (zero)
   On Error Resume Next
   CTxtToSng = CSng(sInput)
End Function
```

You can edit this sample to produce a Double return, if that's more desirable.

—Giovanni Buommino, Dreieich, Germany

VB6
Level: Intermediate

Store Multiple Values in Tag

It would often be convenient to store multiple values in the Tag property. Here are two simple functions that help you do that. The first function stores the value in the control's Tag, tagging the new value with a key value of your choice:

```
Public Function SetTag(ctl As Control, ByVal _
   Key As String, NewValue As String) As String
   Dim myArry() As String
   Dim i As Integer, k As Integer
   Dim yTag As String, yValue As String
   k = -1
   Key = UCase$(Key)
   If ctl.Tag = "" Then
      ctl.Tag = ctl.Tag & "|" & Key & "=" & _
         NewValue
   Else
      myArry = Split(ctl.Tag, "|")
      For i = LBound(myArry) To UBound(myArry)
         If UCase$(Left$(myArry(i), _
            Len(Key))) = Key Then k = i
      Next
      If k > -1 Then
         myArry(k) = Key & "=" & NewValue
         ctl.Tag = Join(myArry, "|")
      Else
         ctl.Tag = ctl.Tag & "|" & Key & "=" _
            & NewValue
      End If
   End If
   SetTag = ctl.Tag
End Function
```

You can store the Tag value easily using a statement like this:

```
Call SetTag(myCtrl, "ID", LoginID)
```

A complementary function allows you to retrieve the stored value:

```
Public Function GetTag(ctl As Control, Key As _
   String) As String
   Dim myArry() As String,
   Dim i As Integer, k As Integer
   Dim yPoze As String, yValue As String
   k = -1
   If ctl.Tag = "" Then
      yValue = ""
   Else
      myArry = Split(ctl.Tag, "|")
      For i = LBound(myArry) To UBound(myArry)
         If UCase$(Left$(myArry(i), _
            Len(Key))) = UCase$(Key) Then k = i
      Next i
      If k > -1 Then
         yPoze = InStr(myArry(k), "=")
         yValue = Mid$(myArry(k), yPoze + 1)
      Else
         yValue = ""
      End If
   End If
   GetTag = yValue
End Function
```

You can retrieve the code using a statement like this:

```
LoginID = GetTag(myCtrl, "ID")
```

As written, these functions are case-insensitive with the key names. If you want case-sensitive key values, remove all UCase calls.

—Enrico Di Cesare, Arese, Italy

VB3 and up
Level: Beginning

Perform Strict Date Validity Check
If you use dates in your program frequently and you want them to be in a particular format, you don't get the desired result with VB's IsDate function. For example, if you want the date to be in MM/DD/YYYY format, this statement returns True because it assumes the string to be in DD/MM/YYYY format:

```
IsDate("23/03/1999")
```

This function checks whether the string passed to it is a valid date in your chosen format:

```
' Input Parameters:
' strdate: String containing the Date
' to be validated.
' strDateFormat: The string containing the
' format for the above date.
' OutPut: Boolean Value indicating whether the
' Date was valid date and in the format
' specified in format string.
Public Function ValidDate(ByVal strDate As _
   String, ByVal strDateFormat As String) _
   As Boolean
   Dim dtCurrDate As Date
   Dim strModDate As String
   On Error GoTo DateError
   ' Convert the Date string to the Given
   ' Format
   strModDate = Format(strDate, strDateFormat)
   ' strModDate will contain the formated Date.
   ' There are three conditions here:
   ' 1. strDate is invalid Date: strModDate
   ' contains string same as strDate.
   ' 2. strDate is valid Date and in Format
   ' specified: strModDate will contain string
   ' same as strDate.
   ' 3. strDate is valid Date and in not Format
   ' specified: strModDate will contain strdate
   ' converted to the date format specified.
   ' Below if statement will Eliminate the 3rd
   ' condition.
   If UCase$(strModDate) = _
      UCase$(Trim$(strDate)) Then
      ' Next statement eliminates 1st
      ' condition.
      ' If strdate string is invalid date,
      ' execution pointer goes to first
      ' statement following DateError Label
      dtCurrDate = Format$(strDate, _
         strDateFormat)
      ValidDate = IsDate(dtCurrDate)
   Else
      ValidDate = False
   End If
   Exit Function
DateError:
   ValidDate = False
'ERROR HANDLING IF DESIRED
End Function
```

Executing this function in the Immediate window yields these results:

```
?ValidDate("03/03/1999","MM/DD/YYYY")
True
?ValidDate("03/23/1999","MM/DD/YYYY")
True
?ValidDate("23/03/1999","MM/DD/YYYY")
False
```

—Madan S. Yadav, Niskayuna, New York

VB3 and up
Level: Beginning

Create a VB Error Message Reference List
Have you ever wanted to have a hard copy listing all the VB error messages? Create a new standard EXE VB application and drop this code into the Load event of the main form:

```
Private Sub Form_Load()
   Dim i As Integer
   For i = 0 To 32000
      'Use this test in VB3:
      'If Error$(i) <> _
         "User-defined error" Then
      'Use this test in VB4 and later:
      If Error$(i) <> "Application-defined " _
         & "or object-defined error" Then
         Printer.Print i, Error$(i)
      End If
   Next iEnd Sub
```

This program prints all defined error descriptions. All undefined error descriptions return "Application-defined or object-defined error" (or "User-defined error" in VB3), so you check for these and ignore them.

—Vince A. Sempronio, Rockville, Maryland

VB6
Level: Intermediate

Copy Listview Contents to Clipboard
This easy routine copies the contents of a listview, including column headers, to the clipboard for pasting into Excel or other applications:

```
Public Sub SendToClipboard(ByVal ListViewObj _
   As MSComctlLib.ListView)
   Dim ListItemObj As MSComctlLib.ListItem
   Dim ListSubItemObj As MSComctlLib.ListSubItem
   Dim ColumnHeaderObj As _
      MSComctlLib.ColumnHeader
   Dim ClipboardText As String
   Dim ClipboardLine As String
   Clipboard.Clear
   For Each ColumnHeaderObj In _
      ListViewObj.ColumnHeaders
      Select Case ColumnHeaderObj.Index
      Case 1
         ClipboardText = ColumnHeaderObj.Text
      Case Else
         ClipboardText = ClipboardText & _
            vbTab & ColumnHeaderObj.Text
      End Select
   Next ColumnHeaderObj
   For Each ListItemObj In _
      ListViewObj.ListItems
      ClipboardLine = ListItemObj.Text
      For Each ListSubItemObj In ListItemObj.ListSubItems
         ClipboardLine = ClipboardLine & _
            vbTab & ListSubItemObj.Text
      Next ListSubItemObj
      ClipboardText = ClipboardText & vbCrLf _
         & ClipboardLine
   Next ListItemObj
   Clipboard.SetText ClipboardText
End Sub
```

—Chris Schneider, Newark, Delaware

VB4, VB5, VB6
Level: Beginning

Modernize Your MDI Apps
The MDI look seems to have gone out of favor recently, judging from the popularity of the Explorer and Outlook styles in modern apps. Maybe that's because of the inconvenience of all those windows cluttering up the place, when only one is required at a time.

However, you can convert an MDI app to an Explorer-style app quickly. First, add a header bar, which in effect replaces the title bar of the child form. This can have your document name and properties on it. To implement this, add a PictureBox to the MDI form, and set its Align property to 1 - AlignTop.

Next, add a navigation bar. You typically place a TreeView or similar control here to provide a more intuitive way to locate and open views or items. This is also a PictureBox, but with its Align property equal to 3 - Align Left (resize the PictureBox to an acceptable width after doing this). Then, for each of your MDI child forms, set these properties:

- Caption: "" (Empty string).
- ControlBox: False
- WindowState: 2 - Maximized

Now change your code so the forms are loaded from the navigation bar instead of through the menus. Note that when MDI child forms are loaded, they appear to fill the entire remaining client space of the MDI form. However, without the control box, minimize, restore, and close icons that would normally be just below the title bar.

—Mark Bertenshaw, Kingston upon Thames, Surrey, England

VB3 and up
Level: Beginning

Standardize Error Reporting Messages
Who doesn't get frustrated with VB's error trapping? Short of buying a commercial add-in, there's not much you can do. But you can keep from typing this over and over:

```
MsgBox "Error " & Err.Number & " - " & Err.Description
```

Instead, put the statement in a module as a Public sub:

```
Public Sub ErrMsg()
   MsgBox "Error " & Err.Number & " - " & Err.Description
End Sub
```

Then you do your error trap like this:

```
Private Some Sub()
   On Error GoTo err_here:
   Exit Sub
err_here:
   ' You might want to trap for and handle a
   ' specific error here
   ErrMsg
End Sub
```

—Mitch Mattek, Norman, Oklahoma

VB4 32, VB5, VB6
Level: Intermediate

☆☆☆☆☆ Five Star Tip

Find the Associated Executable
Sometimes you might need to determine the full path name to a Windows executable file associated with a given file extension. The usual recourse is to use the FindExecutable API. However, there's one hitch: FindExecutable requires an actual file. So to make use of this API, you first need to create a temporary file with the proper extension. Passing the desired extension to this GetAssociatedExecutable routine—for example, TXT or MDB—is an easy way to do this:

```
Private Declare Function FindExecutable Lib _
   "shell32.dll" Alias "FindExecutableA" _
   (ByVal lpFile As String, ByVal lpDirectory _
   As String, ByVal lpResult As String) As Long
Private Declare Function GetTempFileName Lib _
   "kernel32" Alias "GetTempFileNameA" (ByVal _
   lpszPath As String, ByVal lpPrefixString _
   As String, ByVal wUnique As Long, ByVal _
   lpTempFileName As String) As Long
Private Declare Function GetTempPath Lib _
   "kernel32" Alias "GetTempPathA" (ByVal _
   nBufferLength As Long, ByVal lpBuffer As _
   String) As Long
' Usage: GetAssociatedExecutable("MDB")
Public Function GetAssociatedExecutable(ByVal _
   Extension As String) As String
   Dim Path As String
   Dim FileName As String
   Dim nRet As Long
   Const MAX_PATH As Long = 260
   ' Create a tempfile
   Path = String$(MAX_PATH, 0)
   If GetTempPath(MAX_PATH, Path) Then
      FileName = String$(MAX_PATH, 0)
      If GetTempFileName(Path, "~", 0, _
         FileName) Then
         FileName = Left$(FileName, _
            InStr(FileName, vbNullChar) - 1)
         ' Rename it to use supplied extension
         Name FileName As Left$(FileName, _
            InStr(FileName, ".")) & Extension
         FileName = Left$(FileName, _
            InStr(FileName, ".")) & Extension
         ' Get name of associated EXE
         Path = String$(MAX_PATH, 0)
         Call FindExecutable(FileName, _
            vbNullString, Path)
         GetAssociatedExecutable = Left$( _
            Path, InStr(Path, vbNullChar) - 1)
         ' Clean up
         Kill FileName
      End If
   End If
End Function
```

—Pier Minneci, Torino, Italy

VB6
Level: Intermediate

No More Blank Data Reports
When generating a data report based on an ADO recordset linked to a remote database (such as Oracle or SQL Server), you can end up with a blank report. Try setting the recordset's CursorLocation property to adUseClient. If the report is linked to a Command object in a DataEnvironment, this property is set for you.

—Keith Walton, Sacramento, California

VB5, VB6
Level: Intermediate

Detect Change of Windows Locale

While designing multilingual applications, I had to make them respond when a user changes the Windows locale setting—for example, by loading a different language's captions or text. Although it's possible to intercept a Windows message generated and broadcast when the Windows locale changes, this requires subclassing, which is not always desirable.

Another solution is to create an ActiveX control responsible for detecting the change and raising an event. The UserControl object raises a private AmbientChanged event when any of the AmbientProperties change, and it passes the changed property's name to the event. The Ambient object provides a LocaleID property, which you can pass to the client with a public custom LocaleChanged event. Your ActiveX control might have no visual interface, pretty much like a Timer control, and you can place it on any form required to react to a change of Windows locale. You can load the new language's elements subsequently from a database or a resource file:

```
Public Event LocaleChanged(ByVal LocaleID As Long)
Private Sub _
   UserControl_AmbientChanged(PropertyName As String)
   If PropertyName = "LocaleID" Then
      RaiseEvent LocaleChanged(Ambient.LocaleID)
End Sub
```

—Brian Hunter, Brooklyn, New York

VB6
Level: Advanced

Clean Up the MonthView

If you've tried using the Microsoft MonthView control, part of the Windows Common Controls 2 collection, you probably discarded it after discovering the quirky spinner that pops up when you click on the year. This is supposed to make it easy to change years. Unfortunately, when you click on one of the spinner buttons, an ugly border artifact appears to the right of the spinner. If you don't mind eliminating the spinners, you can still use the control. You must subclass the MonthView control temporarily and destroy the spinner button when it is created:

```
' Form code
Option Explicit
Private Declare Function SetWindowLong Lib _
   "user32" Alias "SetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long, ByVal _
   dwNewLong As Long) As Long
Private Const GWL_WNDPROC = (-4)
Private Sub MonthView1_MouseDown(Button As _
   Integer, Shift As Integer, X As Single, Y _
   As Single)
   Dim d As Date
   If MonthView1.HitTest(X, Y, d) = _
      mvwTitleYear Then
      lmvWndProc = _
         SetWindowLong(MonthView1.hWnd, _
         GWL_WNDPROC, AddressOf MVWndProc)
   End If
End Sub
' Module code
Option Explicit
Private 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
Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" (hpvDest As Any, _
   hpvSource As Any, ByVal cbCopy As Long)
Private Declare Function DestroyWindow Lib _
   "user32" (ByVal hWnd As Long) As Long
Private Declare Function SetWindowLong Lib _
   "user32" Alias "SetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long, ByVal _
   dwNewLong As Long) As Long
Private Const GWL_WNDPROC = (-4)
Private Const WM_CREATE = &H1
Private Const WM_PARENTNOTIFY = &H210
Public lmvWndProc As Long
Public Function MVWndProc(ByVal hWnd As Long, _
   ByVal msg As Long, ByVal wParam As Long, _
   ByVal lParam As Long) As Long
   Select Case msg
      Case WM_PARENTNOTIFY
         Select Case LoWord(wParam)
            Case WM_CREATE
               DestroyWindow lParam
               SetWindowLong hWnd, _
                  GWL_WNDPROC, lmvWndProc
         End Select
   End Select
   MVWndProc = CallWindowProc(lmvWndProc, _
      hWnd, msg, wParam, lParam)
End Function
Public Function LoWord(lnum As Long) As Integer
   CopyMemory LoWord, lnum, 2
End Function
Public Function HiWord(lnum As Long) As Integer
   CopyMemory HiWord, ByVal VarPtr(lnum) + 2, 2
End Function
```

—Matt Hart, Tulsa, Oklahoma

VB5, VB6
Level: Advanced

☆☆☆☆☆ Five Star Tip

Capture Reference to UserControl

Many programmers are familiar with declaring an object variable in class modules and other places to capture events from a form and handle them in a generic way:

```
Private WithEvents m_Form As Form
```

It might be useful to do this for user controls as well, but you need a reference to the UserControl object. Getting this reference proves harder than it should be. This code sets up the m_UserControl variable:

```
' Declarations
Private WithEvents m_UserControl As UserControl
Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" (pDest As Any, _
   pSource As Any, ByVal ByteLen As Long)
Private Sub UserControl_Initialize()
   ' Code to set up the m_UserControl variable
   Dim UC As UserControl
   CopyMemory UC, UserControl, 4
   Set m_UserControl = UC
   CopyMemory UC, 0&, 4
End Sub
```

Once this code has been executed, the m_UserControl events fire as expected. Using this technique and sharing the created reference, you can sink the UserControl events in a class module, allowing development of generic event handlers for your controls.

—Jeremy Adams, Tiverton, Devon, England

VB3 and up
Level: Intermediate

Convert Color to Grayscale
Whether you're printing on a grayscale printer, changing a color picture to grayscale, or just trying to decide whether white or black is a better contrast over an arbitrary background color, you need to know how "bright" some color is. This function returns a number between 0 (black) and 255 (white) for any Long color value you send it. It uses the standard algorithm found in everything from color TVs to black-and-white laser printers:

```
Public Function GrayScale (ByVal Colr As _
   Long) As Long
   ' Takes a long integer color value,
   ' returns an equivalent grayscale value
   ' between 0 and 255
   Dim R As Long, G As Long, B As Long
   ' Break up long color into r, g, b
   R = Colr Mod 256
   Colr = Colr \ 256
   G = Colr Mod 256
   Colr = Colr \ 256
   B = Colr Mod 256
   ' Find equivalent grayscale value
   GrayScale = (77 * R + 150 * G + 28 * B) _
      / 255
End Function
```

—Jim Deutch, Syracuse, New York

VB5, VB6
Level: Intermediate

Design Extended Multiselect Listboxes
When you design a database in Access 2000, the most intuitive way for your users to look at the data they want might be to let them use extended multiselect listboxes to pick their information. Here's a function that converts users' selections into a clause that appends to a SQL WHERE statement, creating or modifying a query. You can also use this function as a filter to open a form or report. Because the values passed are always strings, and the first column of the listboxes always contains unique values, you can build the column number and delimiters into the function, where you can easily replace them with optional variables. If the user selects nothing, this function returns an empty string. Although I wrote this code specifically for Access 2000, it should also work in Access 97, Excel 5.0 and higher, and Visual Basic:

```
Public Function WHEREFromListbox(lst As _
   ListBox,strField As String) As String
' Returns a SQL WHERE Clause from a multiselect
' listbox. WHERE is not included so that this
' function can be used as a filter in Access.

   For intLp = 0 To lst.ListCount
      If lst.Selected(intLp) = True Then
         If Len(strResult) > 1 Then
            WHEREFromListbox = _
               WHEREFromListbox & " OR "
         End If
            WHEREFromListbox = _
               WHEREFromListbox & strField & _
               " = '" & lst.Column(0, intLp) _
               & "'"
      End If
   Next intLp
End Function
```

—Carolyn J. Howorth, Murrysville, Pennsylvania

VB3 and up
Level: Intermediate

Draw Arrows
If you've ever wanted to draw a flowchart, diagram, or clock face, you've probably wanted to put arrowheads on some of your lines. This is easy enough if the lines are all vertical or horizontal, but it's tough to do in the general case without the insight that line directions are easily expressed in polar coordinates. This code transforms x,y-space into r,theta-space and back to determine the endpoints of the arrowhead lines:

```
Public Type PointAPI
   X As Long
   Y As Long
End Type
Public Sub DrawArrow(startpt As PointAPI, _
   endpt As PointAPI, canvas As Object, colr _
   As Long, Size As Long)
   Dim x3 As Long, x4 As Long
   Dim y3 As Long, y4 As Long
   Dim xs As Long, ys As Long
   Dim theta As Double 'arrow direction
   Const pi = 3.14159
   ' Polar coords centered on EndPt:
   xs = endpt.X - startpt.X
   ys = endpt.Y - startpt.Y
   ' This is an embedded atan2() function:
   If xs <> 0 Then
      theta = Atn(ys / xs)
      If xs < 0 Then
         theta = theta + pi
      End If
   Else
      If ys < 0 Then
         theta = 3 * pi / 2 '90
      Else
         theta = pi / 2  '270
      End If
   End If
   ' Rotate direction
   theta = theta - 0.8 * pi
   'Find end of one side of arrow:
   x3 = Size * Cos(theta) + endpt.X
   y3 = Size * Sin(theta) + endpt.Y
   ' Rotate other way for other arrow line
   theta = theta + 1.6 * pi
   x4 = Size * Cos(theta) + endpt.X
   y4 = Size * Sin(theta) + endpt.Y
   ' Draw the lines
   canvas.Line (startpt.X, startpt.Y)- _
      (endpt.X, endpt.Y), colr
   canvas.Line (endpt.X, endpt.Y)-(x3, y3), _
      colr
   canvas.Line (endpt.X, endpt.Y)-(x4, y4), _
      colr
End Sub
```

All units are those used by the drawing canvas—a form, picture box, or user control.

—Jim Deutch, Syracuse, New York

VB4 32, VB5, VB6
Level: Intermediate

Limit User Typing in Combo Box

The standard textbox has a MaxChars property that lets you limit the number of characters a user can type into it. The drop-down combo does not, but you can emulate this property setting with a simple API call:

```
Private Declare Function SendMessage Lib _
   "user32" Alias "SendMessageA" (ByVal hWnd _
   As Long, ByVal msg As Long, ByVal wParam _
   As Long, ByVal lParam As Long) As Long
Private Const CB_LIMITTEXT = &H141
Private Sub Form_Load()
   Const Max_Char = 24
   Call SendMessage(Combo1.hWnd, _
      CB_LIMITTEXT, Max_Char, 0&)
End Sub
```

Editor's Note: This tip works in 16-bit versions of VB as well, but you'll need to substitute the correct 16-bit declarations for SendMessage and CB_LIMITTEXT.

—Jim Deutch, Syracuse, New York

VB3 and up
Level: Intermediate

Ask for Directions

In graphical applications, you often need to know the angle between two lines. You can move their intersection point to the origin easily, so all you need to do is choose a point on each line and find the angle between a line from the origin to that point and the x-axis. The angle between the lines is the difference. The arctangent of y/x is the mathematical function you need to find these angles, but VB's Atn( ) function only returns angles between -PI/2 and PI/2 (-90 to +90 degrees). You lose half the circle! And it fails completely if x = 0 (divide by zero error!).

Many languages provide an Atn2( ) function that extends Atn( ) to the entire circle, taking the x and y arguments separately to avoid the divide error. Here's a straightforward VB function that treats all possible cases separately:

```
Function Atn2(ByVal x As Double, ByVal y _
   As Double) As Double
   Dim theta As Double
   Const pi As Double = 3.14159265359
   If x <> 0 Then
      theta = Atn(y / x)
      If x < 0 Then
         theta = theta + pi
      End If
   Else
      If y < 0 Then
         theta = 3 * pi / 2  ' 90 deg
      Else
      theta = pi / 2  ' 270 deg
      End If
   End If
   Atn2 = theta
End Function
```

—Jim Deutch, Syracuse, New York

VB3 and up
Level: Beginning

Center Your Logo on MDI Forms

You can display a logo in the middle of your MDI form. The logo stays in the middle even when the MDI form is resized. After creating your own MDI form, add a standard form to your project and put an Image control named imgLogo on it. Instead of the Image control, you can use a Label control or whatever you want. The standard form (frmLogo) should have these properties set:

```
PROPERTIES of frmLogo:
   MDIChild = True
   BorderStyle = 0 - None
```

Then put this code in your MDI form Resize event:

```
Private Sub MDIForm_Resize()
   ' Now center the frmLogo form in your MDI form
   frmLogo.Left = (Me.ScaleWidth - frmLogo.Width) / 2
   frmLogo.Top = (Me.ScaleHeight - frmLogo.Height) / 2
End Sub
```

Put this code in the logo form's Activate and Resize events:

```
Private Sub Form_Activate()
   ' Force logo to background
   Me.ZOrder vbSendToBack
End Sub
Private Sub Form_Resize()
   ' Move logo to upper-left
   imgLogo.Move 0, 0
   ' The next fragment makes frmLogo's
   ' Width and Height equal to imgLogo's
   ' Width and Height
   Me.Width = imgLogo.Width
   Me.Height = imgLogo.Height
End Sub
```

Load and show the logo form during the MDI form's Load event:

```
Private Sub MDIForm_Load()
   frmLogo.Show
End Sub
```

—Pavel Tsekov, Varna, Bulgaria

VB5, VB6
Level: Beginning

View Right Side of Truncated String

You see trailing ellipses (...) when VB truncates either the expression or data portion of a data tip (the mouse-hover watch value you get while debugging). This is great if you want to see the left side of a long string value, but not quite as compelling if you care about the right side. Hold the control key down and rehover over the expression to force VB to truncate on the left instead of the right. VB truncates all instant watch strings at 251 characters, so you won't see the end of very long strings.

—Matt Curland, Redmond, Washington

VB4 32, VB5, VB6
Level: Beginning

Contain Tab Groupings Within a Frame
The best way to work with the tab control is to set up a different frame for each tab. If you set the frames to be indexed, you can quickly move and make the correct frame visible with this code:

```
' Move and resize the frames to the tabstrip
' control and make the first one visible.
' This should be called in the Form_Load event.
For i = 0 To fraTab.Count - 1
   fraTab(i).Move TabStrip1.ClientLeft, _
      TabStrip1.ClientTop, TabStrip1. _
      ClientWidth, TabStrip1.ClientHeight
   fraTab(i).Visible = (i = 0)
Next i
' To make the correct frame visible use the value
' SelectedItem.Index -1 as the index for the Frames.
' Put this code in the TabStrip1_Click event.
For i = 0 To fraTab.Count - 1
   fraTab(i).Visible = (i = _
      (TabStrip1.SelectedItem.Index - 1))
Next i
```

This approach works with any number of tabs on the TabStrip control.

—Wayne Matheson, Elizabethtown, Kentucky

VB3 and up
Level: Beginning

Avoid Errors on Assigning Null Values
Whenever you read a database field, concatenate an empty string to the field value before assigning it to a variable or control property. This prevents the program from giving an error if a Null value is read from the field. Here's an example:

```
Dim Temp as String
Dim db as Database
Dim rst as Recordset
' Open the Database using DAO
Set db = OpenDatabase(App.Path & "\MyDB.MDB")
' Open the Recordset
Set rst = db.OpenRecordset("SELECT * FROM MyTbl")
Do While not rst.EOF
   ' Read the desired field from the recordset
   Temp = rst.Fields("MyFld") & ""
   List1.Additem Temp
Loop
```

If you don't use the "" piece, the program gives an error if MyFld contains a Null value.

—Kedar Sathe, Houston, Texas

VB3 and up
Level: Beginning

Use ScrollBar Instead of UpDown or Spin Controls
If you want the user to be able to select from a fixed range of numeric values, you have a number of choices. In VB6, you can use the UpDown control, part of Common Controls 2 (630K), or the Spin control, part of the Forms 2.0 Object Library (more than 1 MB and not redistributable). Instead of either of these controls, you can use a standard vertical ScrollBar to provide the same functionality in any VB version from VB3 through VB6, without any additional baggage in your installation.

Place a TextBox (Text1) and a vertical ScrollBar (VScroll1) on the form, with the ScrollBar touching the right edge of the TextBox. Make sure that the ScrollBar is the same height as the TextBox, and that the thumb is not visible. You do not want the user to be able to change the TextBox value by entering a new value directly, so you should set the TextBox Locked property to True.

You need to make the ScrollBar work the same way the other controls do—that is, the value in the TextBox should increment when the user clicks on the up arrow and decrement when the user clicks on the down arrow. This is the reverse of normal ScrollBar behavior, so you must reverse the assignment of the Minimum and Maximum properties. Set the Minimum property to the highest value you want the control to have, and set the Maximum property to the lowest value. You also need to define an increment; the TextBox value changes by this amount when the user clicks on the up or down arrow:

```
Option Explicit
Const MIN_VALUE = 10
Const MAX_VALUE = 100
Const INCREMENT = 10
' Set ScrollBar to act as "up-down" control
Private Sub Form_Load()
   With VScroll1
      ' max < min, so down arrow = decrement,
      ' up arrow = increment
      .Max = MIN_VALUE
      .Min = MAX_VALUE
      .SmallChange = INCREMENT
      ' Start at LOWEST value
      .Value = .Max
   End With
End Sub
```

When the user changes the ScrollBar by clicking on the up or down arrow, you need to update the value in the TextBox:

```
' Updates TextBox value when ScrollBar is changed
Private Sub VScroll1_Change()
   Text1.Text = VScroll1.Value
   If Me.Visible Then Text1.SetFocus
End Sub
```

If the input focus is on the ScrollBar, pressing the up and down arrow keys works just like clicking on the corresponding arrow button on the ScrollBar. If you want the arrow keys to work the same way when the input focus is on the TextBox, use this code:

```
' Change ScrollBar value using up and down
' arrows when TextBox has the input focus
Private Sub Text1_KeyDown(KeyCode As Integer, _
   Shift As Integer)
   VScroll1.SetFocus
   If KeyCode = vbKeyUp Then
      SendKeys "{UP}"
   ElseIf KeyCode = vbKeyDown Then
      SendKeys "{DOWN}"
   End If
End Sub
```

If you hold the arrow key down or keep the up/down button pressed, the TextBox value continues to update until the upper or lower limit is reached.

—Eric Schuyler, Snyder, New York

VB3 and up
Level: Beginning

Pass ByVal to ByRef Parameters

By default, VB passes all arguments to a procedure by reference, which means the procedure can change the values of the variables you pass. However, there's a simple way to override a ByRef argument without changing the procedure's ByRef behavior. Here's a typical procedure with ByRef arguments:

```
Private Sub ModifyByRef(sVar1 As String, _
   sVar2 As String)
   sVar1 = sVar1 & " has been modified."
   sVar2 = sVar2 & " has been modified."
End Sub
```

Use this syntax before calling the procedure:

```
   sVar1 = "Var1"
   sVar2 = "Var2"
   Call ModifyByRef(Var1,Var2)
```

Then you'll get this result:

```
   sVar1 contains "Var1 has been modified."
   sVar2 contains "Var2 has been modified."
```

To override the ByRef for Var1, use VB's expression evaluator and place parentheses around the variable before calling the Modify-ByRef( ) procedure:

```
   ' Note the parentheses around Var1.
   Call ModifyByRef((Var1),Var2)
```

Then you'll get this result:

```
   sVar1 contains the original value "Var1"
   sVar2 contains the changed value _
      "Var2 has been modified."
```

Using the expression evaluator's parentheses gives you control over exactly how parameters are passed into a called procedure, without having to resort to assigning temporary variables to override the behavior of ByRef. By using parentheses, you have a choice.

—David Tate Helene, Rockville, Maryland

VB5, VB6
Level: Beginning

Use Control/Space for VB IntelliSense

You can press Ctrl-Spacebar to make IntelliSense prompt you for variables, methods, properties, or events at any point in a code window.

For example, if you have a variable named myvariable, typing myv, then pressing Ctrl-Spacebar autocompletes the variable name. If more than one item matches what you type, IntelliSense offers a list of matches.

—Doug Waterman, Appleton, Wisconsin

VB4 32, VB5, VB6
Level: Intermediate

Copy an Array Faster

Here's an optimized method of copying one array to another. Usually when copying an array to another array, the developer iterates through each item of the source array, assigning the item to the associated item of the destination array:

```
Private Declare Function timeGetTime Lib _
   "winmm.dll" () As Long
Private Sub Copy()
   Dim i
   Dim startTime As Long
   Dim endTime As Long
   Dim intSrc(1 To 6000000) As Integer
   Dim intDest(1 To 6000000) As Integer
   startTime = timeGetTime
      For i = LBound(intSrc) To UBound(intSrc)
         intDest(i) = intSrc(i)
      Next i
   endTime = timeGetTime
   Debug.Print "Copy took: " & endTime - _
      startTime & " ms."
End Sub
```

Instead, use the Win32 API function CopyMemory to copy the array from source to destination:

```
Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" (Destination As Any, _
   Source As Any, ByVal Length As Long)
Private Sub FastCopy()
   Dim startTime As Long
   Dim endTime As Long
   Dim bytes As Long
   Dim intSrc(1 To 6000000) As Integer
   Dim intDest(1 To 6000000) As Integer
   bytes = (UBound(intSrc) - LBound(intSrc) _
      + 1) * Len(intSrc(LBound(intSrc)))
   startTime = timeGetTime
      CopyMemory intDest(LBound(intDest)), _
         intSrc(LBound(intSrc)), bytes
   endTime = timeGetTime
   Debug.Print "FastCopy took: " _
      & endTime - startTime & " ms."
End Sub
```

When compiled to native code with all optimizations, this second method averages up to 15 times faster, and a stunning 30 to 35 times faster when running as compiled p-code or in the Integrated Development Environment (IDE). But be warned: You can also GPF at blinding speeds if you miscalculate the number of bytes to copy, use bad source or destination addresses, or if your destination array isn't sized sufficiently.

—Andrew Holliday, Phoenix, Arizona

VB3 and up
Level: Beginning

Allow Only Programmatic Input to Combo

Database applications often need to forbid user input to combo boxes, which limits the user to a given set of choices. But just as often, these apps need to assign values to the Text property. Setting a combo's Style property to "2 - Dropdown list" accomplishes the first goal, but prevents the second, throwing an error if an assignment is made to the Text property.

The solution: Set the Style property to "0 - Dropdown combo" at design time, and eat all keystrokes in the combo's KeyPress event (to prevent further processing of the keystrokes), allowing programmatic assignment to the Text property but disallowing user input:

```
Private Sub Combo1_KeyPress(KeyAscii As Integer)
   KeyAscii = 0
End Sub
```

—Kishore Gnanananda, Dubai, United Arab Emirates

VB5, VB6
Level: Intermediate

Clear File Attributes From Entire Directory Tree of Files

This function removes attributes from files in the folder you specify in strFolder. It also can walk down all subfolders of this folder recursively. This function was written mainly for removing the read-only flag that all files have after you copy a bunch from a CD (oldattribute = vbReadOnly), but you can use it with other attributes as well. Look for the other constants for file attributes in the object browser (subset scripting). To use this function, you must first set a Reference to the "Microsoft Scripting Runtime":

```
Option Explicit
Private fso As New Scripting.FileSystemObject
Public Function ClearAttribute(strFolder As _
   String, fIncludeSubfolders As Boolean, _
   oldAttribute As VbFileAttribute) As Long
   Dim fld As Scripting.Folder
   Dim subfld As Scripting.Folder
   Dim file As Scripting.file
   Dim lngCount As Long
   Set fld = fso.GetFolder(strFolder)
   For Each file In fld.Files
      If file.Attributes And oldAttribute Then
         file.Attributes = file.Attributes _
            And Not oldAttribute
         ' Count the files
         lngCount = lngCount + 1
      End If
   Next file
   If fIncludeSubfolders Then
      For Each subfld In fld.SubFolders
         ' Add total from this subfolder
         ' And its subfolders
         lngCount = lngCount + _
            ClearAttribute(subfld.Path, _
               True, oldAttribute)
      Next subfld
   End If
   ' Number of total processed files
   ClearAttribute = lngCount
End Function
```

—Hans Weichselbaumer, Passau, Germany

VB3 and up
Level: Intermediate

Validate Text Against a List of Values

I often find it necessary to check that a string is valid by ensuring it exists in a list of strings. For instance, you might need to check that a user-entered province/state code exists within a list of valid province/state codes. You can do this quickly without looping through the list each time you need to compare. Suppose you have an array containing all valid province/state codes:

```
Private ValidCodes() As String
```

First, translate this array into a string of separated valid codes:

```
Private ValidList As String
Private Sub CreateValidList()
   Dim i As Long, sT As String
   For i = LBound(ValidCodes) To _
      UBound(ValidCodes)
      sT = "|" & ValidCodes(i)
   Next i
   sT = sT & "|"
   ValidList = sT
End Sub
```

I separate the items in the string with vertical bars ("|") because in this situation, bars don't appear in the list of valid codes. You might need to replace the bars with a more suitable character or set of characters. Now, to validate a code, check to see if it's in the string of valid codes:

```
Public Function ValidateCode(sCode As String) _
   As Boolean
   ValidateCode = (InStr(ValidList, "|" & _
      sCode & "|") <> 0)
End Function
```

—Andrew Sadavoy, Toronto, Ontario, Canada

VB5, VB6
Level: Advanced

Keep a MAP File

If the LINK environment variable is set to /MAP when VB launches, then you get a MAP file whenever you Make EXE, even when you are not generating debug symbols. Keeping a map file for all shipped binary files is useful for decoding stack trace information. MAP files are also indispensable when choosing DLL base addresses.

—Matt Curland, Redmond, Washington

VB4 32, VB5, VB6
Level: Intermediate

Avoid Regenerating Overlay Images

If you're creating overlays with the ListImages.Overlay method, always set the ImageList control's BackColor property to the MaskColor property before calling the OverLay method. Return the BackColor to its previous value (probably vbWindowBackground) after calling OverLay. If you generate overlay images with MaskColor = BackColor, then the images don't have to be regenerated when the user changes system colors.

—Matt Curland, Redmond, Washington

VB6
Level: Advanced

Don't Use Default Properties When Working With Hierarchical Recordsets
It's common VB knowledge that you can omit the default property of a control or an object. For example, statements such as Text1.Text = "blah" and Text1 = "blah" are equivalent, with Text being a default property of text control. Objects behave similarly.

When you have two tables, Orders and Items, in a parent-child relationship by OrderNumber, you can build a hierarchical recordset that stores Items' data for a particular order within its parent's record in a column called chapter1 as specified in the rsOrders.Open... statement.

To retrieve the Items for particular orders, you need to loop through the parent recordset rsOrders, and assign the data stored in column chapter1 to a child recordset rsItems. Use a statement such as this:

```
Set rsItems = rsOrders("chapter1").Value
```

If you rely on Value being the default property of Column object and you omit this property in code, you get a Type Mismatch error if your rsItems variable is declared as ADODB.Recordset. Some examples I found in the Microsoft help files would declare this variable as Variant. In this case, you can pass the assignment Set rsItems = rsOrders("chapter1") with no error generated, but later on, if you try to loop through the child set of data, you get an Object Required error referring to an .EOF property that does not exist on a Variant.

None of these problems happen if you declare rsItems properly as ADODB.Recordset and use the Value property of the Column object explicitly. Strangely enough, when you omit .Columns (which is a default property of Recordset object), nothing bad happens. Here's a code example that works:

```
Private Sub cmdGetRecords_Click()
Dim connstring As String
Dim cnn As ADODB.Connection
Dim rsOrders As ADODB.Recordset
Dim rsItems As ADODB.Recordset
Set rsOrders = New ADODB.Recordset
Set cnn = New ADODB.Connection
connstring = "Provider=MSDataShape.1;Data " & _
   "Source=TestDatabase;Initial " & _
   "Catalog=SalesOrderProcess; Connection " & _
   "Timeout=15;DataProvider=SQLOLEDB; User " & _
   "ID=sa; Password=pass"
cnn.Open connstring
rsOrders.Open "SHAPE {Select * From " & _
   "Orders} APPEND ({Select * From Items} " & _
   "as chapter1 RELATE OrderNumber TO " & _
   "OrderNumber)", cnn
Do Until rsOrders.EOF
   Set rsItems = rsOrders("chapter1").Value
   Do Until rsItems.EOF
      Debug.Print rsItems(0), rsItems(1), _
         rsItems(2), rsItems(3)
      rsItems.MoveNext
   Loop
   rsOrders.MoveNext
Loop
End Sub
```

—Brian Hunter, Brooklyn, New York

VB4, VB5
Level: Beginning

Duplicate the Join Function for VB4 and VB5
The native VB6 Split and Join functions have highlighted a number of useful techniques, and now VB5 and VB4 programmers can use this extended facility as well. This code emulates the Join function of VB6 for use in earlier versions. This function takes in an array of information and gives a String as output with delimiters per the user request:

```
Public Function Join(arr As Variant, Optional _
   ByVal delimiter) As String
   Dim sRet As String
   Dim i As Integer
   If IsArray(arr) Then
      If IsMissing(delimiter) Then
         delimiter = " "
      ElseIf Len(CStr(delimiter)) = 0 Then
         delimiter = ""
      Else
         delimiter = CStr(delimiter)
      End If
      For i = LBound(arr) To UBound(arr)
         sRet = sRet & arr(i) & delimiter
      Next i
   End If
   Join = Left(sRet, Len(sRet) - Len(delimiter))
End Function
```

—G. Ajay Kumar, Chennai, India

VB3 and up
Level: Beginning

Store Primary Key in ItemData
Loading a combo/listbox is pretty easy, and determining what the combo/listbox Text property selects is even easier. But if you load a table that might contain duplicate values, you might run into a problem—for example, many people might share the same last name.

Here's the solution. First, load your combo box with a table from your database. A sub such as this works fine, by loading the list with names and storing a lookup key in each item's ItemData property:

```
Public Sub FillComboBox(ctrControl As Control)
   Set rs = db.OpenDatabse("Contact", _
      dbReadOnly)
   If Not rs.EOF Then
      With ctrControl
         Do Until rs.EOF
            .AddItem rs("LastName")
            .ItemData(.NewIndex) = rsTemp("ContactID")
            rs.MoveNext
         Loop
      End With
   End If
   rs.Close
   Set rs = Nothing
End Sub
```

You can now easily determine exactly which name is selected:

```
strSQL = "SELECT * FROM Contact Where " & _
   "ContactID = " & cboMyComboBox.ItemData( _
   cboMyComboBox.ListIndex)
```

—Ken Kilar, Los Angeles, California

VB4 32, VB5, VB6
Level: Intermediate

Show Non-Modal Forms From ActiveX DLLs
If your VB ActiveX DLL includes a non-modal form, you can't summon it from a VC++ client if you call the native VB Show method of the form from within the DLL:

```
Public Sub Show()
   ' Exposed method uses .Show
   Dim frm As New frmMyForm
   ' Show can generate Error 406
   frm.Show
End Sub
```

Instead, use this technique in your exposed interface:

```
Private Declare Function ShowWindow Lib _
   "user32" Alias "ShowWindow" (ByVal hWnd As _
   Long, ByVal nCmdShow As Long) As Long
Private Const SW_SHOW = 5
Public Sub Show()
   ' Exposed method uses API call
   Dim frm As New frmMyForm
   Call ShowWindow(frm.hWnd, SW_SHOW)
End Sub
```

—Alexander Meissel, Colorado Springs, Colorado

VB5, VB6
Level: Beginning

Retrieve Additional File Properties
Several new properties associated with files were not available when the original VB file I/O statements and functions were designed. To easily access these new properties—DateLastAccessed, Type, DateCreated, DateLastModified, Path, ShortPath, ShortName, and ParentFolder—you need to set a project reference to the Microsoft Scripting Runtime. Set the reference by selecting Project from the VB main menu, then select References and check the Microsoft Scripting Runtime item. With the reference established, you can add this code to access and display the new file properties:

```
Dim objFSO As New FileSystemObject
Dim objFileDetails as File
' Identify the file for which you want
' to display properties
Set objFileDetails = _
   objFSO.GetFile("C:\config.sys")
' Move file properties associated with
' the above selected file into labels
' on your property form
lblFileType=objFileDetails.Type
lblDateCreated=objFileDetails.DateCreated
lblDateModified=objFileDetails.DateLastModified
lblDateAccessed=objFileDetails.DateLastAccessed
```

—January Smith, Houston, Texas

VB4, VB5, VB6
Level: Intermediate

Determine Control's Membership in a Control Array
To determine whether a control is a member of a control array, you can reference its Index property and handle the generated error when the control is not in an array. Alternatively, you can use the TypeName function, which returns "Object" for members of a control array. The trick to using it is to reference the control array, not just one of its members. You can do this using the Controls collection, keying on the control's name:

```
Public Function IsCntlArray(cntl As Control) _
   As Boolean
   IsCntlArray = _
      (TypeName(cntl.Parent.Controls(cntl. _
      Name)) = "Object")
End Function
```

—Bill McCarthy, Barongarook, Victoria, Australia

VB5, VB6
Level: Beginning

Ascertain OK or Cancel From InputBox
When the user presses Cancel on a VB InputBox, the string returned is a vbNullString. If the user inputs a zero-length string and presses OK, the return string is empty (""). Unfortunately, in VB, you can't compare an empty string to vbNullString because VB equates "" to be equal to vbNullString even though the two are quite different.

However, you can use the undocumented StrPtr function to determine whether the return string is indeed a vbNullString. A vbNullString's pointer is, by definition, zero:

```
Dim strReturn as String
strReturn = InputBox("Enter in a value")
If StrPtr(strReturn) = 0 Then
   ' User pressed Cancel
End If
```

—Bill McCarthy, Barongarook, Victoria, Australia

VB6
Level: Advanced

☆☆☆☆☆ Five Star Tip

Avoid Copying Data
You can use the name of a Function or Property Get procedure as a local variable anywhere in the procedure. If your procedure returns a String or UDT type, writing directly to the function name instead of a temporary variable saves you from making a full copy of your data at the end of a function. Unfortunately, you can't leverage this technique if your function returns an array, because VB interprets any parentheses after the function name as a call to the function, not as an index into the array. The overloaded parentheses force you to use a local variable and make an expensive array copy at the end of the function. However, if the assignment to the function name happens on the statement before an [End|Exit] [Function|Property], then VB simply transfers ownership of the local variable to the function name instead of copying it. Any intervening statements (including End If) preclude the compiler from making this optimization.

—Matt Curland, Redmond, Washington

VB3 and up
Level: Beginning

Force Tri-State Checkbox Cycling
The CheckBox control in VB supports three positions: Checked, Unchecked, and Grayed. Unfortunately, the default behavior for the control is to cycle between Checked and Unchecked. To set it to Grayed, you must do it programatically.

This code shows you how to cycle between the three positions (the order is Checked->Unchecked->Grayed->Checked ...):

```
Private Sub Check1_Click()
   Static iState As CheckBoxConstants
   Static bUserClick As Boolean
   ' Trap if the user clicked on the control
   ' or if the event was fired because you
   ' changed the value below
   bUserClick = (iState <> Check1.Value)
   ' Prevents you from entering an infinite
   ' loop and getting an Out of Stack Space error
   If bUserClick Then
      Select Case iState
         Case vbChecked
            iState = vbUnchecked
         Case vbUnchecked
            iState = vbGrayed
         Case vbGrayed
            iState = vbChecked
      End Select
      ' This will raise another click event but
      ' your boolean check prevents you from looping
      Check1.Value = iState
   End If
End Sub
```

—Eric Litwin, Thousand Oaks, California

VB3 and up
Level: Intermediate

Use the Immediate Window to Write Repetitive Code
You can stop a program's execution and use the debug window to generate code you can paste into your program. For example, you have a recordset called rs and you wish to manually move the contents into controls on your form or into declared variables. Place a breakpoint after you open the recordset, press Ctrl-G to open the Immediate window, and type this:

```
for each x in rs.Fields : ?"= rs.Fields(""" & _
   x.name & """)" : next
```

When you press Enter, you get one line per field. The output should resemble this:

```
= rs.Fields("Edition")
= rs.Fields("Num")
= rs.Fields("Title")
= rs.Fields("ReaderName")
= rs.Fields("ReaderFrom")
= rs.Fields("Bits16")
= rs.Fields("Bits32")
= rs.Fields("Level")
= rs.Fields("Tip")
```

Copy and paste this output into your code. Now you only need to enter the destination control or variable's name on the left side of the equal signs. If you have a recordset with a large number of fields, this tip is worth its weight in gold. It prevents typing errors and saves time because the field names are pulled right from the recordset.

—Larry Johnson, Trenton, Georgia

VB5, VB6
Level: Beginning

Format Your Version Info
Many professional applications are required to display a version number on all screens to indicate to users which version of the app is currently running. This also helps with configuration management. Here's a function that appends the VB project's version number to a text description passed to the function as input. The version information is embedded in a project by assigning major, minor, and revision values on the Make tab of the Project Properties dialog. Then when you right-click on the resulting EXE file in Windows, go to Properties, and click on the Version tab, the version number matches those on your screens, providing a nice consistency. Putting the function in a standard module—particularly one made of generic reusable functions and subprocedures—allows other developers to plug the module into their projects and use the routine:

```
Public Function GetVersion(strApp As String) _
   As String
   ' Pass in the application name you want
   ' displayed as part of the form's caption. A
   ' blank character and the version number are
   ' appended to the application name
   ' completing the caption.
   GetVersion = strApp & " " & _
      Format(App.Major, "#0") & "." & _
      Format(App.Minor, "#00") & "." & _
      Format(App.Revision, "0000")
End Function
```

Here's a sample call to this function:

```
Dim strVersion As String
strVersion = "Application XYZ Version"
frmMain.Caption = GetVersion(strVersion)
' Set form's caption
```

—Michael T. Hutman, Germantown, Maryland

VB4 32, VB5, VB6
Level: Intermediate

Bind Option Buttons to Data Controls
The Option button is a convenient way to display multiple options from which only one can be selected. One problem is that the Option button cannot be bound to a data control. Here's an easy workaround: Create an array of Option buttons and also create a hidden text field and bind it to your data control. Place this code in your form:

```
Private Sub Option1_Click(Index As Integer)
   Text1.Text = Index
End Sub
Private Sub Text1_Change
   Option1(Val(Text1.Text)).Value = True
End Sub
```

Whenever the value in Text1 is changed by the data control, it sets the Option button of the corresponding index value to True. Whenever the Option button is changed, it stores the corresponding Index in the textbox. Because the textbox is bound to the data control, the value is saved in the database.

—Chris Schneider, Newark, Delaware

VB4 32, VB5, VB6
Level: Beginning

Change Appearance Property at Run Time
Here's a way of changing the "read-only at run time" Appearance property for numerous types of controls. It works especially well with the ListBox, TextBox, and PictureBox controls, without re-creating the control.

Paste this code in a module and pass references to controls you want to alter. By default, this routine changes the style from 3-D to Flat, but you can change it back to 3-D by passing True for the optional second parameter:

```
Option Explicit
Private Declare Function GetWindowLong Lib _
   "user32" Alias "GetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long) As Long
Private Declare Function SetWindowLong Lib _
   "user32" Alias "SetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long, ByVal _
   dwNewLong As Long) As Long
Private Declare Function SetWindowPos Lib _
   "user32" (ByVal hWnd As Long, ByVal _
   hWndInsertAfter As Long, ByVal x As Long, _
   ByVal y As Long, ByVal cx As Long, ByVal _
   cy As Long, ByVal wFlags As Long) As Long
Private Const WS_BORDER = &H800000
Private Const WS_EX_CLIENTEDGE = &H200
Private Const GWL_STYLE = (-16)
Private Const GWL_EXSTYLE = (-20)
Private Const SWP_NOSIZE = &H1
Private Const SWP_NOMOVE = &H2
Private Const SWP_NOZORDER = &H4
Private Const SWP_NOACTIVATE = &H10
Private Const SWP_FRAMECHANGED = &H20
Public Sub ChangeStyle(ctrl As Control, _
   Optional ByVal ThreeD As Boolean = False)
   Dim nStyle As Long
   Dim nStyleEx As Long
   Const swpFlags = SWP_NOSIZE Or _
      SWP_NOMOVE Or SWP_NOZORDER Or _
      SWP_NOACTIVATE Or SWP_FRAMECHANGED
   ' Get current styles
   nStyle = GetWindowLong(ctrl.hWnd, GWL_STYLE)
   nStyleEx = GetWindowLong(ctrl.hWnd, _
      GWL_EXSTYLE)
   If ThreeD Then
      ' Turn Border off, ClientEdge on
      nStyle = nStyle Or WS_BORDER
      nStyleEx = nStyleEx And Not _
         WS_EX_CLIENTEDGE
   Else
      ' Turn Border on, ClientEdge off
      nStyle = nStyle And Not WS_BORDER
      nStyleEx = nStyleEx Or WS_EX_CLIENTEDGE
   End If
   ' Set new styles and force redraw
   Call SetWindowLong(ctrl.hWnd, GWL_STYLE, _
      nStyle)
   Call SetWindowLong(ctrl.hWnd, GWL_EXSTYLE, _
      nStyleEx)
   Call SetWindowPos(ctrl.hWnd, 0, 0, 0, 0, _
      0, swpFlags)
End Sub
```

You might observe the control shrinking when you click on the button several times. You can reproduce this behavior by repeatedly changing the Appearance property of the control from the design-time Properties window. You can inhibit this resizing with ListBox controls by setting IntegralHeight to False. With other controls, you can consider a sizing correction after changing the border style.

—Gilbert R. Rosal, Miami, Florida

VB3 and up
Level: Beginning

Use the Erl Function to Debug
When you're faced with difficult debugging chores or when you want to enhance the value of error logs produced by production code, line numbers can help determine exactly where errors are occurring. The Erl function, not documented since VB3, pinpoints the problem. Here's an example of how Erl returns the line number of an error:

```
Public Sub DivideByZero()
   On Error GoTo HandleError
10    Dim x As Long
20    Dim y As Long
30    y = 5
40    MsgBox y / x 'error
   Exit Sub
HandleError:
   Debug.Print "Error on line " & Erl
End Sub
```

—Aaron Crandall, Boise, Idaho

VB5, VB6
Level: Intermediate

Allow Interval Greater Than Timer Controls
When you need a timer for a larger interval than the Timer control allows, insert this code into a BAS module. The procedure starts when the timer interval has passed:

```
Dim lTimerId As Long
Private Declare Function SetTimer Lib "user32" _
   (ByVal hWnd As Long, ByVal nIDEvent As _
   Long, ByVal uElapse As Long, ByVal _
   lpTimerFunc As Long) As Long
Private Declare Function KillTimer Lib _
   "user32" (ByVal hWnd As Long, ByVal _
   nIDEvent As Long) As Long

Private Sub TimerProc(ByVal lHwnd As Long, _
   ByVal lMsg As Long, ByVal lTimerId As Long, _
   ByVal lTime As Long)
   Dim lResult As Long
   lResult = StopTimer(lTimerId)
   Call InsertYourProcessNameHere
   'code to be executed after interval
End Sub

Public Sub StartTimer(lInterval As Long) _
   'convert interval to milliseconds prior to
   'passing
   lTimerId = SetTimer(0, 0, lInterval, _
      AddressOf TimerProc)
End Sub

Public Function StopTimer(lTimerId As Long) _
   As Long
   'must pass the TimerId returned by SetTimer
   StopTimer = KillTimer(0, lTimerId)
End Function
```

This call executes the procedure:

```
Call StartTimer(5000) '5 seconds
```

You can stop the timer before the interval by calling the StopTimer function.

—Alex Whyte, Como, Australia

VB4, VB5, VB6, DAO 3.x
Level: Intermediate

Execute Parametrized QueryDefs Simultaneously in DAO
In Microsoft Access, you can execute a parameterized query that uses other parameterized queries, as long as their parameter names are the same. Save these queries in an Access database:

```
"QueryOne"
PARAMETERS MyDate DateTime;
SELECT Date1 FROM TableOne WHERE Date1>MyDate;
"QueryTwo"
PARAMETERS MyDate DateTime;
SELECT Date1 FROM TableTwo WHERE Date1>MyDate;
"QueryUnion"
PARAMETERS MyDate DateTime;
SELECT * FROM QueryOne
UNION
SELECT * FROM QueryTwo;
```

You can execute QueryUnion from VB code by passing the MyDate parameter. This example is for DAO 3.5:

```
Sub ExecuteQuery()
Dim db As Database
Dim rs As Recordset
Dim qd As QueryDef
Set db = OpenDatabase("<database name>")
Set qd = db.QueryDefs("QueryUnion")
qd.Parameters(0).Value = CDate("3/1/00")
Set rs = qd.OpenRecordset(dbOpenSnapshot)
' <.....>
rs.Close
db.Close
End Sub
```

—Pavel Maksimuk, Brooklyn, New York

VB4, VB5, VB6
Level: Advanced

Save Expensive Heap Allocations
Fixed-size arrays in local variables use a stack-allocated descriptor as expected, but all the data for an array is allocated on the heap. However, fixed-size arrays embedded in structures are fully stack-allocated. This means that you can save yourself expensive heap allocations by defining a (Private) type with a single fixed-size array element and using a UDT-typed variable in place of the local fixed-size array. You can optimize the number of allocations you need to load a standard module or create a class instance using the same technique at module-level.

—Matt Curland, Redmond, Washington

VB5, VB6
Level: Intermediate

Allow Context-Sensitive Help for Disabled Controls
If you want a form to support context-sensitive help, set the WhatsThisButton and WhatsThisHelp properties on the form to True, and set the WhatsThisHelpID property to a corresponding help-file topic ID for any control on that form for which you want help to be displayed.

Unfortunately, the help isn't shown if the control's Enabled property is set to False. To solve this problem, create a label under the control with the same dimensions, and clear its caption to make it invisible. Set the WhatsThisHelpID property to the same value as the disabled control's property.

—Frank Addati, Melbourne, Australia

VB4, VB5, VB6
Level: Intermediate

Use the ListIndex Property to Store Primary Keys From a Recordset
Here's an easy way to fill a listbox or combobox with names, then retrieve the UserID of that name. This example loads names into a listbox from a SQL Server stored procedure. When you click on a name in the listbox, the Key value is stored in the lngUserID variable. Then you can use the lngUserID variable in other parts of the program to retrieve related information for the selected name. The names are set up as character fields with the UserID being an AutoNumber field and also the primary key. This tip is valid only if you can translate the field value to a number:

```
Private Sub Form_Load()
   Call LoadData(List1)
End Sub
Private Sub List1_Click()
   If List1.ListIndex>=0 Then
      lngUserID = _
         List1.ItemData(List1.ListIndex)
   End If
End Sub
Private Sub LoadData(ByRef obj As Object)
' Assumes the Object is either a ListBox or ComboBox
   Dim com as ADODB.Command
   Dim rs as ADODB.Recordset
   Set com=CreateObject("ADODB.Command")
   Set rs=CreateObject("ADODB.Recordset")
   com.CommandText = "procGetData"
   com.CommandType = adCmdStoredProc
   com.ActiveConnection = strConnect
   Set rs=com.Execute
   obj.Clear
   Do While Not rs.EOF
      obj.AddItem rs!Name
      obj.ItemData(obj.NewIndex) = rs!UserID
      rs.MoveNext
   Loop
   rs.Close
   set rs=Nothing
   set com=Nothing
End Sub
```

—Steve Ramsey, Tyrone, Pennsylvania

VB5, VB6
Level: Intermediate

Use Bitwise Comparison in SQL Server Queries
The newsgroups offer a lot of discussion about bitwise comparison in SQL statements. VB supports true bitwise arithmetic with And, but SQL supports only a logical AND and returns only TRUE or FALSE. Here's a quick way to test against a single bit in SQL:

```
SELECT MyField
FROM MyTable
WHERE (MyTable.MyField \ 2 ^ (MySingleBit - 1) _
   MOD 2 = 1)
```

The \ operator specifies integer division, although you could have used INT (MyTable.MyField / ...) just as easily. MySingleBit is the bit you want to test: 1,2,3,4,5, and so on. More complicated ways of doing this—such as with table joins—might be faster, but this is about as simple as it gets.

—Merv Pate, Houston, Texas

VB4 32, VB5, VB6
Level: Intermediate

Convert Values for Collection
If you're adding something to a collection, you might want to convert it before adding it. This code adds a reference to the recordset, *not* the value of the field returned by this syntax normally:

```
' (col is a collection, rs is a recordset)
col.Add rs("Some Field")
```

When you try to get the value from the collection, you get an error if the recordset is gone—not the best coding practice. If you want a simple collection, you want to convert the value:

```
col.Add CStr(rs("Some Field"))
```

Here's another way around the problem:

```
col.Add rs("Some Field").Value
```

The default property of a recordset is the .Value property, which is why you see a value when you code things such as str = rs("Some Field"). You should be coding with rs("Field").Value, but it's more work, so you don't. The penalty for shortcutting the default property comes when things don't behave properly—for example, with a collection reference. If you have other object values that you want to put into a collection, be sure you know what you're referencing—the object or the value.

—Dan Kobelt, Abbotsford, British Columbia, Canada

VB5, VB6
Level: Intermediate

Use OLE Drag-and-Drop With a DBGrid Control
It's easy to drag-and-drop from a DBGrid to Word or Excel, even though DBGrids don't support OLE Automation. Just add a rich textbox to your form and make it invisible. Also set its OLEDragMode to 0 - rtfOLEDragManual. Then in the DBGrid's MouseMove event, add this code:

```
Private Sub DBGrid1_MouseMove(Button As _
   Integer, Shift As Integer, X As _
   Single, Y As Single)
   ' start the drag from here...
   If Button = vbLeftButton Then
      rtfDrag.OLEDrag
   End If
End Sub
```

In the rich textbox's OLEStartDrag event, set up the allowed format and effect:

```
Private Sub rtfDrag_OLEStartDrag(Data As _
   RichTextLib.DataObject, AllowedEffects As Long)
   ' valid formats and effects
   Data.SetData , vbCFRTF
   Data.SetData , vbCFText
   AllowedEffects = vbDropEffectCopy
End Sub
```

In the OLESetData event for the rich textbox, you must generate the data you want to export. You can include tabs, or you can create a table in Word, select it, and drop it on the rich textbox to see what code it generates.

—William Wensley, received by e-mail

VB4 32, VB5, VB6; SQL Server 6.5 and up; ADO 2.1 and up
Level: Intermediate

Updatable Join Recordset Using ADO and SQL Server
Contrary to popular thought, you *can* add new records to an ADO 2.1 Recordset object that is the result of a Join operation executed on multiple-base tables. You must specify the UniqueTable, or the name of the base table upon which updates, insertions, and deletions are allowed. This example uses the SQLOLEDB.1 provider and a disconnected ADO recordset as a bonus:

```
Private Sub Form_Load()
   ' You could combine the following 5 steps in
   ' the .Open method
   objRecordset.ActiveConnection = _
      objConnection  'An ADO connection object
   objRecordset.CursorLocation = adUseClient
   ' Must use client-side server with this
   ' property!
   objRecordset.CursorType = adOpenStatic
   ' Must use this with client-sided server cursor
   objRecordset.LockType = adLockBatchOptimistic
   ' Hooking this up to a bound grid in a
   ' disconnected mode using the Northwind
   ' database
   objRecordset.Open "SELECT * " & _
      "FROM Customers JOIN Orders ON " & _
      "Customers.CustomerID = " & _
      "Orders.CustomerID WHERE city = " & _
      "'London' ORDER BY CustomerID"
   objRecordset.Properties("Unique Table"). _
      Value = "Orders"
   objRecordset.Properties("Resync " & _
      Command").Value = "SELECT * FROM " & _
      "(SELECT * FROM Customers JOIN " & _
      "Orders ON Customers.CustomerID = " & _
      "Orders.CustomerID WHERE city = " & _
      "'London' ORDER BY CustomerID) " & _
      "WHERE Orders.OrderID = ?"
   objRecordset.ActiveConnection = Nothing
   ' A disconnected ADO recordset
   Set grdTest.Datasource = objRecordset
End Sub
Private Sub Save()
   objRecordset.ActiveConnection = _
      objConnection
   ' Reconnect for the purpose of saving only
   objRecordset.UpdateBatch
   ' Don't forget to check the ADO errors
   ' collection!
   objRecordset.ActiveConnection = Nothing
   ' Disconnect again
End Sub
```

It's also imperative that you use the Resync command method to instruct ADO how to uniquely identify all rows being refreshed. In this method, you must supply the original Select statement and append all primary keys of the UniqueTable in a Where clause with open-ended ("?") tokens as R-values.

—Alexander Meissel, Colorado Springs, Colorado

VB4 32, VB5, VB6
Level: Intermediate

Toggle Min/Max Buttons at Run Time
Here's some code that toggles the MinButton and MaxButton properties of a form at run time. To demonstrate, start a new project, add two command buttons to the default form, and paste this code into the form's code window. One button toggles the form's MaxButton, and the other toggles the form's MinButton. The code works by flipping the WS_MAXIMIZEBOX and WS_MINIMIZEBOX window-style bits back and forth. Make a final call to SetWindowPos to force a redraw of the nonclient area. It will be useful for Microsoft to drop the notion of "read-only at run time" properties in VB:

```
Private Declare Function GetWindowLong Lib _
   "user32" Alias "GetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long) As Long
Private Declare Function SetWindowLong Lib _
   "user32" Alias "SetWindowLongA" (ByVal _
   hWnd As Long, ByVal nIndex As Long, _
   ByVal dwNewLong As Long) As Long
Private Declare Function SetWindowPos Lib _
   "user32" (ByVal hWnd As Long, ByVal _
   hWndInsertAfter As Long, ByVal x As Long, _
   ByVal y As Long, ByVal cx As Long, ByVal _
   cy As Long, ByVal wFlags As Long) As Long
Private Const WS_MINIMIZEBOX = &H20000
Private Const WS_MAXIMIZEBOX = &H10000
Private Const GWL_STYLE = (-16)
Private Const SWP_FRAMECHANGED = &H20
Private Const SWP_NOMOVE = &H2
Private Const SWP_NOZORDER = &H4
Private Const SWP_NOSIZE = &H1
Private Sub Command1_Click()
   Dim nStyle As Long
   nStyle = GetWindowLong(Me.hWnd, GWL_STYLE)
   Call SetWindowLong(Me.hWnd, GWL_STYLE, _
      nStyle Xor WS_MAXIMIZEBOX)
   SetWindowPos Me.hWnd, 0, 0, 0, 0, 0, _
      SWP_FRAMECHANGED Or SWP_NOMOVE Or SWP_NOSIZE
End Sub
Private Sub Command2_Click()
   Dim nStyle As Long
   nStyle = GetWindowLong(Me.hWnd, GWL_STYLE)
   Call SetWindowLong(Me.hWnd, GWL_STYLE, _
      nStyle Xor WS_MINIMIZEBOX)
   SetWindowPos Me.hWnd, 0, 0, 0, 0, 0, _
      SWP_FRAMECHANGED Or SWP_NOMOVE Or SWP_NOSIZE
End Sub
```

—S. Partha Sarathy, Chennai, India

VB5, VB6
Level: Advanced

Pass Error Message Throughout Nested Components
When you use components in your VB projects, sometimes it's hard to determine which component has an error when the application stops running. You might especially have this problem with nested components.

Here's a structure for developing your nested components that lets you obtain detailed error information from your application. You can determine which component and which method has the error without having to work through your code.

Create your component using this structure:

```
'/
' Component1
'
Public Sub GetName(ByRef strName As String)
On Error GoTo EndOfSub
Dim objComponent2 As New Component2
'Create Component2
   Dim strErrMsg As String
   Dim lYourErrorNumber As Long
   strErrMsg = ""
   ' Write your code to something else
   ' Record any error to strErrMsg and set lYourErrorNumber
   ' Get name from Component2
   Call objCompnent2.GetName(strName)
   If strErrMsg <> "" Then
      ' Use "GetName::Component1" as Err.Source
      ' here for any error that occurred inside this method
      Err.Raise vbObjectError + lYourErrorNumber, _
         "GetName::Component1", strErrMsg
   End If
EndOfSub:
   If Err.Number <> 0 Then
      ' raise error to component's client program
      Err.Raise Err.Number, Err.Source, Err.Description
   End If
End Sub
'/
'/
' Component2
'
Public Sub GetName(ByRef strName As String)
On Error GoTo EndOfSub
   Dim strErrMsg As String
   Dim lYourErrorNumber As Long
   strErrMsg = ""
   ' Write your code to get name
   ' Record any error to strErrMsg and set lYourErrorNumber
   If strErrMsg <> "" Then
      ' Raise error
      Err.Raise vbObjectError + _
         lYourErrorNumber, , strErrMsg
   End If
EndOfSub:
   If Err.Number <> 0 Then
      ' Raise error to component's client program
      ' Use "GetName::Component2" as Err.Source
      ' here for any error ocurred inside this method
      Err.Raise Err.Number, _
         "GetName::Component2", Err.Description
   End If
End Sub
'/
```

Use this code in your application program:

```
'/
' Client program
'
Sub DoMyJob()
On Error GoTo EndOfSub
   Dim objComponent1 As New Component1
   'Create Component1
   Dim strMyName As String
   ' Get name from Component1
   Call objComponent1.GetName(strMyName)
   ' Write your code to do your job
EndOfSub:
   If Err.Number <> 0 Then
      ' If any error occurred in the component1
      ' or component2, it can be catched here,
      ' including detail information about
      ' error number, source and description
      Err.Raise Err.Number, Err.Source, Err.Description
   End If
End Sub
'/
```

—Peter Luo, Calgary, Alberta, Canada