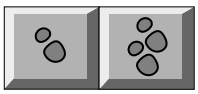

by Karl E. Peterson

CREEPING VERSION-ITIS

BE PREPARED! NEW OPERATING SYSTEMS ARE TAKING OVER.

Over the next year or two, your applications may find themselves running in any one of three Microsoft operating systems. Because the transition from 16-bit to 32-bit has the potential to be much more traumatic for programmers than the DOS-to-Windows shift, the time to prepare is now.

As 32-Bit Bucket section leader in *VBPJ's* CompuServe forum, it's appropriate that my first tip is how to determine if your application is running under Windows NT. While most APIs work just fine, some will return strange values or even fail miserably. As more and more users are discovering the wonders of true 32-bit multitasking, it's not too early to begin adopting alternate strategies to deal with the peculiarities of NT.

Sixteen-bit Visual Basic 3.0 apps run in the Win16-On-Win32 subsystem of NT, which provides a majority of the Win16 API calls. The Win16 GetWinFlags function returns information about your machine and its operating mode. To determine if you are operating in Windows NT, compare the return value from GetWinFlags with the constant value WF_WINNT. Using the And operator tests a specific bit. A nonzero result means that bit is set. The function IsWinNT will return true if NT is running:

```
' Enter this declaration in the General section
Declare Function GetWinFlags Lib "Kernel" () As Long

Function IsWinNT () As Integer
   Const WF_WINNT = &H4000
   '
   ' Use GetWinFlags to test for NT.
   '
   If GetWinFlags() And WF_WINNT Then
      IsWinNT = True
   Else
      IsWinNT = False
   End If
End Function
```

The GetVersion API is the standard method of determining the version of Windows that's running. Since the release of Windows for Workgroups 3.11, however, this function doesn't always return the correct version information.

The VerWin function uses the GetVersion API to retrieve version information for Windows (see Listing 1). GetVersion returns a long integer with the Windows version stored in the low word and the DOS version stored in the high word. Within each word, the major version is in the low byte, and the minor version is in the high byte. Rather than messing with floating-point numbers, the VerWin function returns an integer that equals the true version number multiplied by 100.

Microsoft decided that Windows for Workgroups 3.11 should return 3.10. So when Windows appears to be version 3.10, investigate further. To confirm that it's running under 3.10 or 3.11, VerWin passes USER.EXE to GetFileVersionInfo, which returns the internally coded version information for a file.

Many of you are probably using the Windows 95 Beta-3 public prerelease. Because the Win16 API was written before the new operating system was called Chicago, it contains no API call to detect the operating system. By calling the GetVersion API, you can deduce that Windows 95 is running. When a Win16 app calls GetVersion, Windows 95 returns 3.95 for the sake of compatibility.

The IsWin95 function tests the return value of VerWin against 395 and returns the results accordingly:

```
Function IsWin95 () As Integer
   '
   ' Win95 returns 3.95 to Win16 apps because
   ' returning 4.0 broke too many of them!
   '
   If VerWin() = 395 Then
      IsWin95 = True
   Else
      IsWin95 = False
   End If
End Function
```

Remember, VerWin returns the version number multiplied by 100 to avoid floating-point numbers (see Listing 1).

The VerWin function requires splitting a long integer into individual words, and splitting those words into bytes. Four functions are invaluable for these purposes (see Listing 2); refer to the VerWin function to see them in use (see Listing 1). Basic has always suffered from the lack of an unsigned integer data type. You can overcome this with a little math, temporarily shifting a negative integer into the long-integer range before performing bitwise operations on it.

CALLING WIN32 API FROM VB3

Now that you've determined that you're running under Windows NT or Windows 95, download CALL32.ZIP from the Magazine Library of the *VBPJ* Forum on CompuServe. This archive contains a DLL that provides a method for 16-bit Visual Basic apps to call Win32 API functions.CALL32.DLL was written by Peter Golde, and it has been placed into the public domain.

Karl Peterson is a GIS analyst with a regional transportation planning agency and a member of the VBPJ *Technical Review Board. Karl has coathored a book that is scheduled for publication concurrent with the upcoming release of a major programming language <g>. He's the 32-Bit Bucket Section Leader for the* VBPJ *Forum and a Microsoft MVP in the MSBASIC forum. Contact Karl in either CompuServe location at 72302,3707.*

Hundreds of new API calls are available in Win32. Some replace system functionality that is no longer available, because DOS is no longer an underlying layer. Previously, you could access services in Win16 only by reaching below Windows to invoke DOS interrupts. Some of the new APIs address this.

The information from GetDriveType has been expanded in Win32. This function now detects CD-ROM and RAM disks. To detect free space on a disk, the GetDiskFreeSpace function returns cluster and sector size, as well as the total number of clusters for a drive and how many are free.

To use CALL32.DLL with these functions, start a new project, and place the declarations and constants shown in the General section of the default form (see Listing 3). Use the Declare32 function to set up a thunking mechanism for the intended 32-bit API calls. The next two declarations are for those Win32 functions, but they have been modified slightly to work with CALL32. Note that the Lib argument has been set to "call32.dll." Also, a parameter has been added to the end of the call. This ID parameter identifies which function CALL32 should dispatch. Declare one persistent ID variable for each Win32 API function at either the module or global level. You must use this ID, retrieved upon initialization, whenever the call is made.

In the Form_Load event, or wherever your application is initialized, retrieve a persistent ID from Declare32 for each of the Win32 APIs you intend to call. The parameters to Delcare32 include the function name, the library where it resides, and a string that indicates what sort of parameters it expects (one letter for each parameter):

```
Sub Form_Load ()
   '
   ' Obtain function ID's for Win32 calls
   '
   idFreeSpace = Declare32("GetDiskFreeSpaceA", _
      "kernel32", "ppppp")
   idDriveType = Declare32("GetDriveTypeA", "kernel32",
"p")
End Sub
```

I've listed the data types supported in the last parameter (see Table 1). The Form_Click event initiates a loop that tests drive letters from A to Z for drive type (see Listing 4). If the program detects a valid drive, it collects statistics on free and used space and calculates byte totals. Output is directed to the form. IDs retrieved in Form_Load are passed with the Win32 calls.

KEYWORD OF THE MONTH: IIF

The IIf function, which you can use to evaluate whether an expression is true or false, is troublesome because it's not implemented in VBRUN300.DLL. For some reason, IIf is found in MSAFINX.DLL—a separate DLL that programmers often overlook when making distribution disks. If you use temporary string variables in an IIf parameter, you'll probably get a GPF. The GPF often occurs at the conclusion of the procedure containing the offense, or even at apparently random locations, making it almost impossible to locate the cause:

```
' Recipe for disaster!
Label1 = IIf(x > 100, "Warning: " & x
& " too high!", _
   "In Range: " & x)
```

IIf doesn't properly release the handles of temporary string variables used as one of its arguments, such as the second and third arguments I've shown. But that alone does not cause the GPF. If a large string is allocated and there is not enough contiguous space for it, VB will try to compact the heap, either immediately or upon exit of the procedure containing IIf. In either of these situations, a GPF will occur the next time the program allocates a temporary string that happens to use the string handle utilized by the IIf function.

The good news is there's absolutely no compelling reason to use the IIf function. Use a block if-then-else structure whenever you're tempted to use IIf. This approach does not risk a GPF, nor does it jeopardize your distribution with missing files:

```
' The *sane* way to do it.
If x > 100 Then
   Label1 = "Warning: " & x & " too high!"
Else
   Label1 = "In Range: " & x
End If
```

```
' Enter these declarations in the General section
Declare Function GetVersion Lib "Kernel" () As Long
Declare Function GetFileVersionInfo Lib "VER.DLL" _
  (ByVal lpszFileName$, ByVal handle As Any, ByVal _
  cbBuf&, ByVal lpvData$) As Integer

Function VerWin% ()
  Dim nRet%
  Dim Ver$
  '
  ' Use GetVersion to find initial answer, which
  ' is byte-swapped in the low word of return.
  '
  nRet = WordLo(GetVersion())
  nRet = ByteLo(nRet) * 100 + ByteHi(nRet)
  '
  ' For "compatibility", v3.11 returns v3.10, check it.
  '
  If nRet = 310 Then
     Ver = Space$(255)
     nRet = GetFileVersionInfo("user.exe", 0&, _
        Len(Ver), Ver)
     '
     ' Find position in Ver$ of "FileVersion" stamp.
     '
     nRet = InStr(Ver, "FileVersion")
     '
     ' Look just beyond stamp for version string.
     '
     If Mid$(Ver, nRet + 12, 4) = "3.11" Then
        VerWin = 311
     Else
        VerWin = 310
     End If
  Else
     VerWin = nRet
  End If
End Function
```

LISTING 1 ***Retrieving Version Information.*** *The VerWin function uses GetVersion to retrieve version information. Even if it appears that version 3.10 is running, check for version 3.11.*

C Data Type	Visual Basic Declare Type	Declare32Type
int, UINT	ByVal Long	i
LONG, DWORD	ByVal Long	i
HANDLE	ByVal Long	i
WORD, short	(not supported by CALL32.DLL)	(not supported by CALL32.DLL)
HWND	ByVal Long	w (i for no 16-bit to 32-bit translation)
LPSTR, LPCTSTR	ByVal String	p
LPDWORD, LPUINT, int FAR *	Long	p
LPWORD	Integer	p

TABLE 1 ***Parameter Type Support in CALL32.*** *The first column shows likely parameters defined in Win32 API references. The second column shows how these types translate to a Visual Basic Declare, and the third column shows how each type is defined for CALL32.DLL in the Declare32 function call.*

If you can't avoid using the IIf function (although it's certainly hard to imagine why you couldn't), be absolutely sure to avoid the use of temporary string variables in its arguments. If you don't, it is highly likely that you will face a general protection fault. And just as with earthquakes in California, you won't know when the "big one" will hit. ■

```
Function ByteHi% (WordIn%)
  '
  ' Lop off low byte with divide. If less than
  ' zero, then account for sign bit (adding &h10000
  ' implicitly converts to Long before divide).
  '
  If WordIn < 0 Then
     ByteHi = (WordIn + &H10000) \ &H100
  Else
     ByteHi = WordIn \ &H100
  End If
End Function

Function ByteLo% (WordIn%)
  '
  ' Mask off high byte and return low.
  '
  ByteLo = WordIn And &HFF
End Function

Function WordHi% (LongIn&)
  '
  ' Mask off low word then do integer divide to
  ' shift right by 16.
  '
  WordHi = (LongIn And &HFFFF0000) \ &H10000
End Function

Function WordLo% (LongIn&)
  '
  ' Low word retrieved by masking off high word.
  ' If low word is too large, twiddle sign bit.
  '
  If (LongIn And &HFFFF&) > &H7FFF Then
     WordLo = (LongIn And &HFFFF&) - &H10000
  Else
     WordLo = LongIn And &HFFFF&
  End If
End Function
```

LISTING 2 ***Splitting Words***. *Windows and other function libraries may pack fields into one return value. These functions can split long integers into individual words and words into bytes.*

```
' API Declarations
Declare Function Declare32 Lib "call32.dll" (ByVal _
  Func As String, ByVal Library As String, ByVal Args _
  As String) As Long
Declare Function GetDiskFreeSpace Lib "call32.dll" _
  Alias "Call32" (ByVal lpRootPathname As String, _
  lpSectorsPerCluster As Long, lpBytesPerSector _
  As Long, lpNumberOfFreeClusters As Long, _
  lpTotalNumberOfClusters As Long, ByVal id As Long) _
  As Long
Declare Function GetDriveType Lib "call32.dll" Alias _
  "Call32" (ByVal lpRootPathname As String, ByVal id _
  As Long) As Long

' GetDriveType return values
Const DRIVE_UNKNOWN = 0
Const DRIVE_NOTPRESENT = 1
Const DRIVE_REMOVABLE = 2
Const DRIVE_FIXED = 3
Const DRIVE_REMOTE = 4
Const DRIVE_CDROM = 5
Const DRIVE_RAMDISK = 6

' Module-level ID variable for each Win32 API
Dim idFreeSpace As Long
Dim idDriveType As Long
```

LISTING 3 ***API Declarations for CALL32.*** *Place these declarations and constants in the General section of the default form to build a test project that uses GetDriveType and GetDiskFreeSpace.*

```
Sub Form_Click ()
   Dim i&                  ' loop counter
   Dim drv$                ' string parameter for drive
   Dim dType&              ' long function returns
   ' longs for disk space function:
   Dim dfsSectorsPerCluster&
   Dim dfsBytesPerSector&
   Dim dfsFreeClusters&
   Dim dfsTotalClusters&
   ' calculated values:
   Dim FreeSpace&
   Dim TotalSpace&
   '
   ' Clear the form, and loop through the alphabet.
   '
   Me.Cls
   For i = 65 To 90
      '
      ' Construct string used to identify root.
      '
      drv$ = Chr$(i) & ":\"
      Me.Print " "; drv$,
      '
      ' Display type of drive for each letter.
      '
      dType = GetDriveType(drv$, idDriveType)
      Select Case dType
         Case DRIVE_UNKNOWN
            Me.Print "Cannot be determined."
         Case DRIVE_NOTPRESENT
            Me.Print "Root directory does not exist."
         Case DRIVE_REMOVABLE
            Me.Print "Can be removed from the drive.",
         Case DRIVE_FIXED
            Me.Print "Cannot be removed from the drive.",
         Case DRIVE_REMOTE
            Me.Print "Remote (network) drive.", ,
         Case DRIVE_CDROM
            Me.Print "CD-ROM drive.", ,
         Case DRIVE_RAMDISK
            Me.Print "RAM disk.", ,
      End Select
      '
      ' Get and display free/total space.
      '
      If dType > DRIVE_NOTPRESENT Then
         If GetDiskFreeSpace(drv$, dfsSectorsPerCluster, _
            dfsBytesPerSector, dfsFreeClusters, _
            dfsTotalClusters, idFreeSpace) Then
            FreeSpace = dfsBytesPerSector * _
               dfsSectorsPerCluster * dfsFreeClusters
            TotalSpace = dfsBytesPerSector * _
               dfsSectorsPerCluster * dfsTotalClusters
            Me.Print Format(FreeSpace, "#,##0"); "/"; _
                  Format(TotalSpace, "#,##0"); " _
                    Bytes Free";
            Me.Print " ("; Format(FreeSpace / _
               TotalSpace * 100, "0.0"); "%)"
         Else
            Me.Print "Media missing."
         End If
      End If
   Next i
End Sub
```

LISTING 4 ***Getting to the Root of Drive Types***. *Place this code in the Form_Click event to determine the drive type of each potential disk on the system. If the program detects a valid drive, it collects the statistics on free and used space and calculates the byte totals.*