Tool hot to handle

VB makes it easy to add version information to an application – but how do you read it? Tim Anderson explains.

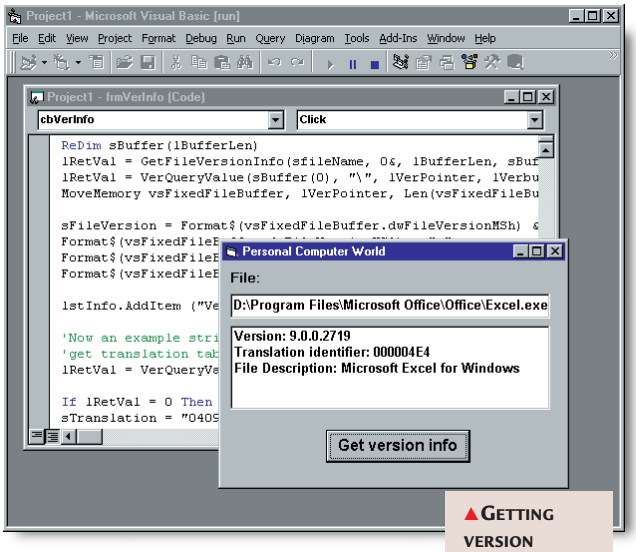
Here's an interesting question. Paul O'Neill wants to know: 'Which API calls do I need to find out the proper program description of an .EXE or .COM file?' Windows can do it, if you right-click a program file and click Properties – Version.

There are three API calls that do this job. GetFileVersionInfo fills a buffer with file version information. You need to specify the size of the buffer, which you can find out with GetFileVersionInfoSize. Then call VerQueryValue to interpret the data. It sounds straightforward, but it isn't, particularly for Visual Basic developers. Here is a more detailed look. Even if you do not want to do this, it is worth investigating, as it shows how awkward the Windows API can be.

The Windows version information feature is flexible. All files stamped with version information contain a VS_FIXEDFILEINFO structure or type, with fields for the file version, the product version (intended to be the version number of the product with which the file was distributed) and a bitfield with flags to indicate debug, patched, pre-release, private or special builds.

There are also fields for the target operating system, the type of the file (for example, application, DLL or driver). It doesn't stop there. VS_FIXEDFILEINFO is only one of the elements in a higher-level VS_VERSIONINFO structure. If you look at typical version information using Windows Explorer, you will notice it varies. WINWORD.EXE has two LegalTrademark fields. NetObjects' FUSION.EXE has a Configuration field. In fact, developers can name these string fields as they want to and the number of available fields is not fixed. To further enhance the system it allows for multiple languages and code pages. You could present different strings for US English and UK English, as well as for French, German and Arabic.

All this flexibility comes at a price in that it makes the data in the version resource more complex to store. This is why there is a VerQueryValue function to unpack it. Here is how it works, with tips for VB developers. First, you need a module with declares for the VS_FIXEDFILENFO structure, the functions GetFileVersionInfo, GetFileVersion InfoSize, and VerQueryValue, and the constants used by these functions.



▲ GETTING VERSION INFORMATION

You need to declare MoveMemory, an alias for a runtime library function RtlCopyMemory. The reason is that obtaining version information makes extensive use of pointers, which means an easy ride for C, C++ or Delphi developers and a rough time for VB. The declares and constants are not printed here for space reasons, but you can find them in the MSDN library articles shown at the end of this feature.

To obtain the version information, first call GetFileVersionInfoSize. If it returns zero, there was either an error or else no version information is stored in that file. Next, use ReDim to allocate memory for a byte array, into which the version information is copied using GetFileVersionInfo [Fig 1]. Plain sailing so far, but using VerQueryValue needs care. The C declaration looks like this:

```
Bool VerQueryValue(
const LPVOID pBlock,
LPTSTR lpSubBlock,
LPVOID *lplpBuffer,
PUINT pulen
);
```

The first parameter is the buffer you filled with GetFileVersionInfo. The second is a string describing the data you want back from VerQueryValue. The third is a pointer to the address of the returned data (a pointer to a pointer). Finally, pulen is a pointer to an integer variable, which is filled by VerQueryValue with the length of the data in lplpBuffer.

If you only need the basic VS_FIXEDFILEINFO structure, the lpSubBlock parameter is easy. Just pass a string containing a single backslash, indicating the root data. Most of the pointers are then handled transparently as long as you get the declaration right, but not the lplpBuffer value. To access the data you need the MoveMemory function mentioned above. This simply copies the data to the address of your VS_FIXEDFILEINFO variable. The snag is that the slightest error on your part will crash your application horribly.

Once you have obtained the VS_FIXEDFILEINFO data, reading the fields is straightforward, if tedious. But what if you want to get the string information, such as File Description, Company Name or even LegalTrademark? The procedure is similar, except instead that of a VS_FIXEDFILEINFO structure you need only a string variable. The tricky bit is getting the right value for lpSubBlock. It is a two-stage process. First, you give it the value: "\VarFileInfo\Translation", in order to obtain an array of translation identifiers, one for each language and code page supported by this version resource. For each translation where you want to obtain string values, you then call VerQueryValue, again giving lpSubBlock the value "\StringFileInfo\lang-codepage\string-

[FIG 1]

VB Code for extracting version information. You also need various declarations as given in the example code on MSDN (*see box next page*).

```
Private Sub cbVerInfo_Click
Dim lRetVal As Long
Dim lJunk As Long
Dim sBuffer() As Byte
Dim lBufferLen As Long
Dim lVerPointer As Long
Dim vsFixedFileBuffer As VS_FIXEDFILEINFO
Dim lVerbufferLen As Long
Dim sDescription As String
Dim icountvar As Integer
Dim lpTransTable As Long
Dim lTranslation As Long
Dim iLang As Integer
Dim iCode As Integer
Dim sTranslation As String
Dim sHexBit As String
Dim sfileName As String
Dim sFileVersion As String

sfileName = txtFileName.Text
lBufferLen = GetFileVersionInfoSize(sfileName, ↙
lJunk)
If lBufferLen < 1 Then
MsgBox "No Version Info available!"
Exit Sub
End If

ReDim sBuffer(lBufferLen)
lRetVal = GetFileVersionInfo(sfileName, 0&, lBufferLen, sBuffer(0))
lRetVal = VerQueryValue(sBuffer(0), "\", lVerPointer, lVerbufferLen)
MoveMemory vsFixedFileBuffer, lVerPointer, Len(vsFixedFileBuffer)

sFileVersion = Format$(vsFixedFileBuffer.dwFileVersionMSh) & "." & _
Format$(vsFixedFileBuffer.dwFileVersionMSl) & "." & _
Format$(vsFixedFileBuffer.dwFileVersionLSh) & "." & _
Format$(vsFixedFileBuffer.dwFileVersionLSl)

lstInfo.AddItem ("Version: " & sFileVersion)

'Now an example string value
'get translation table
lRetVal = VerQueryValue(sBuffer(0), "\VarFileInfo\Translation", ↙
lVerPointer, lVerbufferLen)

If lRetVal = 0 Then
sTranslation = "04090000"
Else

If lVerbufferLen < 4 Then
MsgBox "Error retrieving translation"
Exit Sub
End If
```

[FIG 1 contd.]

```
MoveMemory ByVal VarPtr(lTranslation), lVerPointer, ↙
Len(lTranslation)
iLang = LoWord(lTranslation)

iCode = HiWord(lTranslation)

sHexBit = Hex(iLang)
While Len(sHexBit) < 4
sHexBit = "0" + sHexBit
Wend

sTranslation = sHexBit

sHexBit = Hex(iCode)
While Len(sHexBit) < 4
sHexBit = "0" + sHexBit
Wend

sTranslation = sTranslation + sHexBit
End If

lstInfo.AddItem ("Translation identifier: " + sTranslation)

lRetVal = VerQueryValue(sBuffer(0), "\StringFileInfo\" + ↙
sTranslation + "\FileDescription", lVerPointer, ↙
lVerbufferLen)
If lRetVal <> 0 Then
sDescription = String$(lVerbufferLen + 1, Chr$(0))
MoveMemory ByVal sDescription, lVerPointer, lVerbufferLen
lstInfo.AddItem ("File Description: " & sDescription)
End If
End Sub
```

(*Key:* ↙ *code string continues*)

name", where lang-codepage is a string representation of the translation identifier in hexadecimal, and string-name is the name of the string key. Examples are "FileDescription", "CompanyName" and "Comments". There is a little mystery surrounding these key names. The official documentation states that they must be one of a pre-defined range, but later says that these are guidelines only.

Getting the right translation identifier string is just a little tricky in Visual Basic. Multi-language version information appears to be rare, and neither Visual Basic nor Delphi support this in the IDE. Even if there is only one translation identifier, it must still be converted to the right string. The long integer value contains two sub-values, each of which is a Word (unsigned 16bit) value. The code page identifier is stored in the most significant 16bits, and the language identifier in the least significant 16bits. Visual Basic doesn't have built-in

HiWord and LoWord functions, but it is easy to add utility functions to do this. The neatest way is to use the MoveMemory function and simply copy the two bytes required [Fig 2]. Note the use of VarPtr, a semi-documented function that retrieves the address of a VB variable so you can use it in functions that expect pointers.

When you have retrieved the language and code page identifiers, you can use VB's Hex function to convert them to a string hexadecimal representation, which then must be padded with zero characters so that each one ends up as four-character string. Finally, combine the two to make an eight-character string to use in VerQueryValue.

[FIG 2]

Declarations for MoveMemory, HiWord and LoWord

```
Declare Sub MoveMemory Lib "KERNEL32" Alias ↙
"RtlMoveMemory" _
(dest As Any, ByVal Source As Long, ByVal ↙
length As Long)
Function HiWord(ByVal dw As Long) As Integer
MoveMemory HiWord, ByVal VarPtr(dw) + 2, 2
End Function
Function LoWord(ByVal dw As Long) As Integer
MoveMemory LoWord, ByVal VarPtr(dw), 2
End Function
```

(*Key:* ↙ *code string continues*)

What does this tell you about VB?

Once you have worked out how to get version information with Visual Basic, you can tuck the code into your utility library and never worry about it again. The thing could easily be wrapped into a class and a web search would probably turn up some example code. There is something strange here though. It is odd that VB lets you write version information into an application with a simple dialog in project properties, but makes retrieving it such a convoluted effort. The information can be of critical importance. An example is when you need to interoperate with a third-party application. It is quite possible some detail of the file format, DDE or COM automation interface might change from one version to another. You cannot rely on date stamps, so retrieving version information is the best solution.

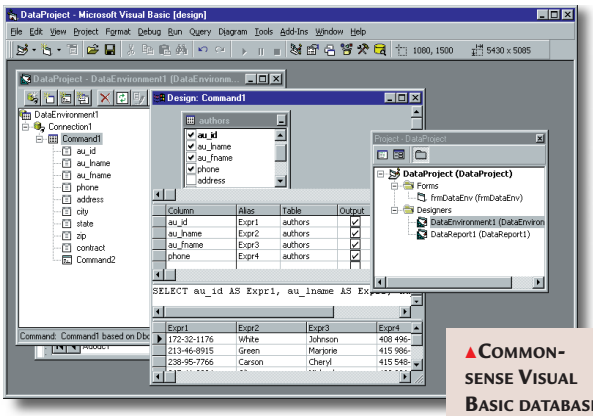
There is another way of looking at this. It may be tricky, but it can be done in VB. Arguably, VB developers get the best of both worlds, an easy language for most tasks, and a workaround when things get tough. If you want to know more about VB and pointers, see box below.

Common-sense VB databases

Although there are numerous books on database development, few are as useful as they should be. Many concentrate on going blow-by-blow through some product or other, which is useful, but tends to duplicate what is already in the official documentation. How you set about designing and implementing database solutions is tackled less often.

Charles Williams' new book, *Professional Visual Basic 6 Databases*, is a good one. Although it covers some advanced material, it does not assume a lot of knowledge and there are chapters on database basics, client-server basics, an overview of SQL (Structured Query Language) and a section on database normalisation, which is the art of storing data efficiently and consistently.

If you know all this, you can skip to the material on ADO (Advanced Data Objects), which does seem to be catching on as the best route to data when using Microsoft tools. There is a particularly impressive section on three-tier solutions, including sections on MTS (Microsoft Transaction Server), the Distributed Transaction Coordinator for transactions that involve more than one database, security, and data warehousing.



▲ COMMON-SENSE VISUAL BASIC DATABASES

There is a certain mystery surrounding how you are meant to implement multi-tier solutions using VB, so a common-sense guide is welcome. There is an emphasis on SQL Server rather than Access, which makes sense now that the SQL data engine can be used for free in small-scale projects, but little coverage of other databases such as Oracle and DB2, which have a larger market share than Microsoft's SQL database. Nevertheless, this is a useful new resource.

BOOKS AND DOWNLOADS

The classic book on Visual Basic and the API is Daniel Appleman's *Visual Basic Programmer's Guide to the Win32 API* (SAMS ISBN 06723 15904) £53.99 inc VAT. On the accompanying CD the author supplies some handy utility functions for working with pointers. Another classic is Bruce McKinney's *Hardcore Visual Basic*, which is distributed on the MSDN library CD (supplied with VB or contact Microsoft on 0345 002 000). Look up the chapter called *Dealing with Pointers*, and the section *Bring your Hatchet*.

The MSDN CD also has example code for extracting version information in articles Q139491 which covers VS_FIXEDFILEINFO and Q160042 which covers string information. These are also available from www.microsoft.com.

You can download the example code in this feature from www.only connect.co.uk.

Professional Visual Basic 6 Databases by Charles Williams (Wrox, ISBN 1861002025) is £35.99.

PCW CONTACTS

Tim Anderson welcomes your Visual Programming comments and queries. Contact him at visual@pcw.co.uk or via the PCW editorial office