Visual Studio Magazine Online
Classic VB Corner

Finding the Right Tool for the Job

Tired of wondering which version of dumpbin or link is executing? Would you like to copy that tool you use all the time to another machine, but can't seem to find it? Here's a little utility that can help.

September 22, 2009 · by Karl E. Peterson

Developers tend to collect command line utilities like squirrels do their nuts. And, just like the tree rats, I think most of us tend to forget where we put our tools sometimes. I certainly do. This afternoon, I needed to use dumpbin in a relatively "pure" Win98 VM. I knew I had it on my development machine -- I could type its name at the command prompt and get the usage. But hell if I knew *where* it was, so I could copy it over to the VM!

Time to write another console app. (A guy can never have enough nuts, eh?) I'm probably worse than many in this regard, which explains why I put together that Console Framework that you can download from my website. With that, there's virtually no effort involved in producing a tool for almost any need, especially this one. I knew that Windows searches the PATH environment variable in order, but wasn't sure how to determine what it considered an executable extension.

It turns out the PATHEXT environment variable is there for us to use. This reduces the problem to native VB, other than outputting to the console of course. In a nutshell:

```
Private Function Find(ByVal FindExe As String, _
   Optional ByVal FindAll As Boolean = False) As Long

   Dim ext() As String
   Dim path() As String
   Dim pathext As String
   Dim i As Long, j As Long
   Dim TryFile As String
   Dim nFound As Long

   ' Break path into separate folders, leaving
   ' room for current directory first.
   path = Split(";" & Environ("PATH"), ";")
   path(0) = CurDir$

   ' Find order of executable file extensions.
   pathext = Environ("PATHEXT")
   If Len(pathext) = 0 Then
      ' Undefined?  Up to us!
      pathext = ".COM;.EXE;.BAT;.CMD"
   End If
   ' Leave empty extension as first option, to cover
   ' situations where full filename was given.
   ext = Split(";" & pathext, ";")

   ' Loop through all possibilities.
```

```
   For i = 0 To UBound(path)
      ' Avoid duplicate path entries.
      If Not Duplicate(path(), i) Then
         ' Try each extension in order.
         For j = 0 To UBound(ext)
            TryFile = path(i) & "\" & FindExe & LCase$(ext(j))
            If FileExists(TryFile) Then
               nFound = nFound + 1
               Con.WriteLine TryFile
               If FindAll = False Then Exit For
            End If
         Next j
         If (FindAll = False) And (nFound > 0) Then Exit For
      End If
   Next i

   ' Return number of files found.
   Find = nFound
End Function
```

We start by breaking the PATH down into its individual elements with a simple Split call. The only twist is the need to leave an empty first element to which we can assign the current directory, as that's always searched first.

Then we go after the PATHEXT variable. Unfortunately, this is only present by default in Windows NT-class systems, so we need to provide a backup plan for Windows 9x systems that lack this information. I chose to default to ".COM;.EXE;.BAT;.CMD" for this situation. The first element of the extensions array needs special attention too. Prepending the semi-colon to the PATHEXT leaves us with an empty first element, which would mean the first test in each folder will be for an exact match.

Now it's just a simple matter of looping through each folder in our path array, and looking first for an exact match. When no match is found, we append each known executable extension in turn and do another search. Optionally, the loop may be exited on the first match, or output all matching files, which can help track down frustrating version issues.

In the process of developing this little tool, I also made an embarrassing discovery. I actually had *three duplicate folders* in my PATH. They were all VS6 related, so I assume the multiple setups over the years somehow did this. There was one each, in the User and System environment variable tables. Hence the urge to add a Duplicate() function to avoid multiple identical results when looking for all possible matches:

```
Private Function Duplicate(sArray() As String, _
   ByVal TestElement As Long) As Boolean

   Dim i As Long
   ' Look for matching string preceeding test element.
   For i = LBound(sArray) To (TestElement -- 1)
      If StrComp(sArray(i), sArray(TestElement), vbTextCompare) _
                                                        = 0 Then
         Duplicate = True
         Exit For
```

```
      End If
   Next i
End Function
```

Granted, that's not highly efficient with large arrays, but in this situation it's entirely adequate.

File Exists?

One of *the most* frequently asked questions online is how to determine whether a file exists. Unfortunately, the most common response is to guide the newb to use VB's native Dir function. Bad response! The problem is that will cause VB to release the Find handle it may already have open for another Dir iteration. The most elegant way to test for file existence is to simply query its attributes:

```
Private Function FileExists(ByVal FileName As String) As Boolean
   Dim nAttr As Long
   ' Grab this files attributes, and make sure it isn't a folder.
   ' This test includes cases where file doesn't exist at all.
   On Error GoTo NoFile
   nAttr = GetAttr(FileName)
   If (nAttr And vbDirectory) <> vbDirectory Then
      FileExists = True
   End If
NoFile:
End Function
```

The only thing to be aware of is that an error could be generated with a very small handful of "weird" system files, so that has to be trapped and avoided. If that sort of error trapping is distasteful on esthetic grounds, you can always turn to the API for the same task:

```
Public Function FileExists(ByVal FileName As String) As Boolean
   Dim nAttr As Long
   ' Grab this files attributes, and make sure it isn't a folder.
   ' This test includes cases where file doesn't exist at all.
   nAttr = GetFileAttributes(FileName)
   If (nAttr And vbDirectory) <> vbDirectory Then
      FileExists = True
   ElseIf Err.LastDllError = ERROR_SHARING_VIOLATION Then
      FileExists = True
   End If
End Function
```

Why though? Classic VB is just fine for this.

Reverse Twist

What if you want to know which of your many tools is associated with a given document type? You may have noticed that if you just enter a document name at the command prompt, Windows obliges by essentially calling ShellExecute with the "open" verb on it. Why not support this reverse lookup in our little tool, too? It's extremely easy to do with the FindExecutable API:

```
Private Function FindApplication(ByVal DocName As String) As String
   Dim Result As String
   Const MAX_PATH As Long = 260
   ' Find executable associated with this document.
   Result = Space$(MAX_PATH)
   If FindExecutable(DocName, vbNullString, Result) > 32 Then
      FindApplication = _
         Left$(Result, InStr(Result, vbNullChar) -- 1)
   End If
End Function
```

This API function provides the bonus that if the passed "document" is actually an executable; it returns the name of the passed "document" itself. As it turns out, this is very nearly a functional replacement for all our loopy code above, but for one thing. Remember all those nuts that look alike? The loopy code was written to (optionally) find all instances of an executable anywhere on the path, while FindExecutable will only return the one that actually executes if you just type its name.

Injecting the following code before the path/extension loops solves that problem:

```
   ' Look for an associated application, in case user
   ' passed a document rather than executable filename.
   If Not KnownExtension(FindExe, ext) Then
      TryFile = FindApplication(FindExe)
      If Len(TryFile) = 0 Then
         TryFile = FindApplication(CurDir$ & "\" & FindExe)
      End If
      ' Return results if associated application was found.
      If Len(TryFile) Then
         Con.WriteLine TryFile
         Find = 1
      End If
      Exit Function
   End If
```

Unfortunately, this solution introduces another problem of its own. We now need to do a quick check to see if we were passed a file with a known executable extension. That means extracting the extension, if there is one, and comparing it to all the elements of the PATHEXT. Not difficult, of course, but it is something that make this column a tad longer:

```
Private Function KnownExtension(FileName As String, _
   Extensions() As String) As Boolean

   Dim i As Long
   Dim ext As String
   ' Loop through array, looking for match.
   ext = FileExtension(FileName)
   For i = LBound(Extensions) To UBound(Extensions)
      If StrComp(ext, Extensions(i), vbTextCompare) = 0 Then
         KnownExtension = True
         Exit For
      End If
   Next i
End Function
```

If you'd like to download the completed project, along with VB6 source and an EXE compiled to run on any system with the VB6 runtime installed, see the Which sample on my site.

Now you have one more tool for your hoard. Just don't lose it!

About the Author

Karl E. Peterson wrote Q&A, Programming Techniques, and various other columns for VBPJ and VSM from 1995 onward, until Classic VB columns were dropped entirely in favor of other languages. Similarly, Karl was a Microsoft BASIC MVP from 1994 through 2005, until such community contributions were no longer deemed valuable. He is the author of VisualStudioMagazine.com's new Classic VB Corner column. You can contact him through his Web site if you'd like to suggest future topics for this column.