Parsing and Building File and Path Names

Tamar E. Granor, PhD

I started using FoxBase+ nearly 20 years ago. In the evolution from that remarkably able product to Visual FoxPro 9, hundreds, perhaps thousands, of new elements have been added to the FoxPro programming language. Each new version has introduced not only new capabilities, but new ways to do old things.

Old habits die hard. Once you know how to do something, changing to a new way takes some effort. But the new approaches are often faster, more readable, or both. Now that the VFP language has stabilized, it's a good time to work on writing the best code we can with the tools at hand.

In this series, I'll look at some of the new approaches that have been introduced over the years and try to make a compelling case for change.

This first article looks at techniques for taking apart and putting together files and paths. These are fairly common tasks, both for developer tools and end-user applications. We may need to put a file in a particular place or create a back-up copy of a file, use the same name but a different extension, or any number of similar tasks.

The structure of files and paths

Filenames consist of a path, a stem and an extension. The path ends with the rightmost backslash. The extension begins after the rightmost period. Everything in between is the stem.

With such a strictly defined structure, it's not hard to take filenames apart and put them back together, using FoxPro's powerful string manipulation language. You can use functions like AT() (or better yet, RAT()), LEFT(), RIGHT() and SUBSTR(), together with easy concatenation using the + operator.

However, there's an easier, more readable, way. Starting in VFP 6, the language includes a set of functions with names like JustPath() and ForceExt() that make file and path manipulation a piece of cake. Not only that, but these functions were in the FoxTools library at least as far back as FoxPro 2.5 for Windows.

Let's look at specific tasks to see why you should learn and use these functions. Before we start, let's define some terminology a little more clearly. In particular, the term "filename" is ambiguous.

A *path* is a sequence of folders, possibly beginning with a drive designator. For example, D:\Fox\VFP9. A path may or may not include a terminating backslash.

An *extension* is the portion of a filename that normally indicates the file type. In the Windows world, it's usually three characters. For example, DOC, PRG or JPG. There are a few commonly used four-character extensions, such as JPEG and HTML. The extension follows a period in the filename.

A *stem* or *filestem* is the main part of the filename that indicates the name of a particular file. For example, in VFP.EXE the stem is VFP.

Filename can refer to either the stem plus the extension, such as VFP.EXE, or the complete reference including the path, such as D:\Fox\VFP9\VFP.EXE. To avoid this ambiguity, for the rest of this article, I'll use filename to refer to only stem plus extension, and use *full name* or *fully-pathed filename* to refer to the complete reference.

Constructing fully-pathed filenames

There are several tasks you might want to do around the idea of putting parts together into full names. Probably the most common is taking a path and a filename to create a fully-pathed filename.

In the examples here, assume that cPath contains the path and cFile contains the filename.

When handling this task manually, the main issue is ensuring you have the final backslash between the path and the stem. Here's one way to do that:

```
cFullName = cPath
IF RIGHT(cPath, 1) <> "\"
  cFullName = m.cFullName + "\"
ENDIF
cFullName = m.cFullName + m.cFile
```

Of course, this is a common enough operation that you might want to write it in a single line and you can do that, thanks to the IIF() function:

```
cFullName = m.cPath +,;
            IIF(RIGHT(m.cPath,1) = "\",;
                "", "\") + m.cFile
```

Both approaches give the same result. However, in both cases, you have to take a good look at the code to figure out what it does. A more maintainable approach is to use the ForcePath() function. It accepts two parameters, a filename and a path, and returns a fully-pathed filename. Here's the same example:

```
cFullName = FORCEPATH(m.cFile, m.cPath)
```

This version is so much more readable that I'd pay a small speed penalty to have it in my code, and it turns out that's what it costs, sort of. In my testing, the five-line version of the old way takes about twice as long as the one-liner. The ForcePath() version takes about 30% longer than the one-liner. However, all three are so fast that speed is essentially a non-issue. On my production machine, a loop with 100,000 passes took between 0.1 and 0.2 seconds, depending on the version. So, unless you're doing intensive file processing with a need to build millions of filenames, speed shouldn't be a consideration. For readability and maintainability, the winner is clear.

ForcePath() is actually much more powerful than this examples demonstrates. Later in this article, I'll show you how you can use it to avoid a whole lot of parsing.

Adding the backslash

The tedious step in the old versions of the previous example is ensuring that the path ends with a backslash. This is actually another thing we can do better with a built-in function. As the earlier code demonstrates, the old way to do this is with IF or IIF(), like this:

```
* More readable old version
m.cPathWithBackSlash = m.cPath
IF RIGHT(m.cPath, 1) <> "\"
  m.cPathWithBackSlash = ;
        m.cPathWithBackSlash + "\"
ENDIF

* One-line old version
cPathWithBackSlash = m.cPath + ;
         IIF(RIGHT(m.cPath,1) = "\", "", "\")
```

But the AddBS() function provides a more readable alternative:

```
cPathWithBackSlash = ADDBS(m.cPath)
```

The timing for this one is interesting. If the path already has the backslash, the three approaches all take essentially the same time. Over 10,000 passes, the differences were thousandths of a second on my machine. However, if the path needs the backslash added, my tests found the one-liner and the AddBS() versions essentially identical, but the more verbose version, using IF, took about twice as long.

Adding an extension

Just as you might combine a path and a filename to get a fully-pathed filename, you can combine a filestem and an extension to get a filename. Here, the key issue in the old version is ensuring you have the period between the stem and the extension.

For these examples, assume cStem contains the filestem and cExt contains the extension.

Here's the old way, using concatenation. Although it's unlikely that the filestem includes the trailing period, the code checks for it anyway. (For completeness, you might even want to check the extension for a leading period, so you don't end up with two.)

```
cFile = m.cStem
IF RIGHT(m.cFile,1) <> "."
  cFile = m.cFile + "."
ENDIF
cFile = m.cFile + m.cExt
```

As with files and paths, there is a less readable, one-line of this:

```
cFile = m.cStem +;
        IIF(RIGHT(m.cFile,1)= ".",;
            "",".") + m.cExt
```

But the ForceExt() function supersedes both versions, providing a clear, readable line of code:

```
cFile = FORCEEXT(m.cStem, m.cExt)
```

Once again, the one-liner and the new way take about the same time, while the longer version using IF takes about 2.5 times as long. Also, as with ForcePath(), ForceExt() is much more powerful than this example shows. I'll show you where it really shines later in this article.

All of the examples so far may seem a little cooked, since we're often handed fully-pathed filenames that we need to manipulate. That usually involves parsing them into their components and, frequently, putting them back together differently. We'll look next at the parsing tasks.

Separating the path and filename

Probably the most common parsing task is separating the path from the filename. That is, given a fully-pathed filename, extract the path and the filename into separate variables. Sometimes, you only need one or the other. For these examples, we'll assume cFullName contains a fully-pathed filename.

Using VFP's string functions, you can separate the path and filename like this:

```
* The old way
LOCAL nPathEnds
nPathEnds = RAT("\", m.cFullName)
cPath = LEFT(m.cFullName, m.nPathEnds-1)
cFile = SUBSTR(m.cFullName, m.nPathEnds + 1)
```

That's not too much code, but it certainly doesn't tell you what it's doing. The JustPath() and JustFName() functions let you understand the code immediately. Here's the new way:

```
cPath = JUSTPATH(m.cFullName)
cFile = JUSTFNAME(m.cFullName)
```

The timing for this code surprised me. I expected the new way to be much faster. In fact, the comparison depends on the length of the path. With no path, just a filename, the new way is the same or faster. After that, the longer the path, the faster the old way is in comparison to the new. When I saw this result, I expected it to be a function of the number of levels (how many folders to traverse), but in fact, it seems to be based on the number of characters. Regardless, all of this is quite fast. The slowest test I saw took less than 0.4 seconds for 10,000 passes.

Finding the extension

Another common task is grabbing the extension from a filename, whether there's a path or not. This allows you to find out what kind of file you're dealing with. As with path, sometimes you want to grab both filestem and extension, while at others, you only need the extension.

Here's the old way to grab just the extension:

```
LOCAL nPeriodAt

nPeriodAt = RAT(".", m.cFullName)
IF nPeriodAt = 0
  cExtension = ""
ELSE
  cExtension = RIGHT(m.cFullName, ;
               LEN(m.cFullName) - m.nPeriodAt)
ENDIF
```

The new way uses the JustExt() function:

```
cExtension = JUSTEXT(m.cFullName)
```

Interestingly, in this case, the new way is faster, except in the case of a file with a long path and no extension. I suspect what's going on there is that JustExt() is walking backward from the end of the string looking for a period and not finding one. In all the other cases I tested, using JustExt() took from one-half to two-thirds as long as the old way. As with all the other tests, though, both versions were so fast as to make the question moot. In this case, 10,000 passes in under 0.2 seconds.

Extracting just the stem

The last in the set of "pulling filenames apart" tasks is extracting just the filestem, the part in between the path and the extension. Like the other tasks, the old way involves parsing to find the final backslash and the final period, while the new way is a single function call, in this case, to JustStem().

The code for the old way is more complicated than the other cases because it has to deal with filenames with no path, filenames with no extension, filenames with neither and filenames with both:

```
LOCAL nPathEnds, nPeriodAt

nPathEnds = RAT("\", m.cFullName)
nPeriodAt = RAT(".", m.cFullName)
DO CASE
CASE nPathEnds = 0 AND nPeriodAt = 0
  * All we have is the stem
  cFileStem = m.cFullName

CASE nPathEnds = 0
  * No path, just filename
  cFileStem = LEFT(m.cFullName, m.nPeriodAt-1)

CASE nPeriodAt = 0
  * No extension
  cFileStem = SUBSTR(m.cFullName, ;
              m.nPathEnds + 1)

OTHERWISE
  * Have all parts, extract
  nFullLen = LEN(m.cFullName)
  nExtLen = nFullLen - m.nPeriodAt
  nStemLen = m.nFullLen - m.nPathEnds ;
             - m.nExtLen - 1
  cFileStem = SUBSTR(m.cFullName, ;
              m.nPathEnds + 1, m.nStemLen)

ENDCASE
```

From a code point of view, it's easy to see why the new way is an improvement:

```
cFileStem = JUSTSTEM(m.cFullName)
```

Not surprisingly, given how much code is involved in the old way, the new way was faster in every case I tested. The factor ranged from half the time of the old way down to a quarter of the time.

The Full Monty

Of course, what you sometimes want is to combine all of this and pull a filename into its component parts. Since getting the filestem is more complex using the old way, my code for the complete parse is based on that and simply adds the code to set the path and extension to each of the four cases.

```
LOCAL nPathEnds, nPeriodAt

nPathEnds = RAT("\", m.cFullName)
nPeriodAt = RAT(".", m.cFullName)
DO CASE
CASE nPathEnds = 0 AND nPeriodAt = 0
  * All we have is the stem
  cFileStem = m.cFullName
  cPath = ""
  cExtension = ""
```

```
CASE nPathEnds = 0
  * No path, just filename
  cFileStem = LEFT(m.cFullName, m.nPeriodAt-1)
  cPath = ""
  cExtension = RIGHT(m.cFullName, ;
               LEN(m.cFullName) - m.nPeriodAt)

CASE nPeriodAt = 0
  * No extension
  cFileStem = SUBSTR(m.cFullName, ;
               m.nPathEnds + 1)
  cPath = LEFT(m.cFullName, m.nPathEnds - 1 )

OTHERWISE
  * Have all parts, extract
  nFullLen = LEN(m.cFullName)
  nExtLen = nFullLen - m.nPeriodAt
  nStemLen = m.nFullLen - m.nPathEnds ;
               - m.nExtLen - 1
  cFileStem = SUBSTR(m.cFullName, ;
               m.nPathEnds + 1, m.nStemLen)
  cPath = LEFT(m.cFullName, m.nPathEnds - 1 )
  cExtension = RIGHT(m.cFullName, ;
               LEN(m.cFullName) - m.nPeriodAt)
ENDCASE
```

The new way is just three lines, like this:

```
cFileStem = JUSTSTEM(m.cFullName)
cPath = JUSTPATH(m.cFullName)
cExtension = JUSTEXT(m.cFullName)
```

My tests show the new way to be faster, except when there's a long path and no extension. No doubt that's the same slowdown encountered when extracting the extension from such a filename.

Forcing paths and extensions

Finally, it's time to look at the real power of ForcePath() and ForceExt(). In the earlier examples in this article, we assumed that the filename didn't include a path or extension, respectively. In fact, these functions do as their names suggest, and remove an existing path or extension before adding the new one.

So, using ForcePath(), you can start with a fully-pathed filename and end up with a fully-pathed filename, but pointing to a different path. Similarly, ForceExt() lets you start with a filename (with or without path) including extension and returns a filename that's the same except that the extension has been changed.

Here's the old way of forcing a path:

```
nPathEnds = RAT("\", m.cFullName)
cFile = SUBSTR(m.cFullName, m.nPathEnds + 1)

cNewName = cNewPath
IF RIGHT(cPath, 1) <> "\"
  cNewName = m.cNewName + "\"
ENDIF
cNewName = m.cNewName + m.cFile
```

Here's the new way, once again, a one-liner:

```
cNewName = FORCEPATH(m.cFullName, m.cNewPath)
```

In my tests, the new way takes from one-third to one-half the time of the old way. The old way takes about the same time no matter what the original filename is, and whether or not it has a path. The new way varies with the length of the original path.

Here's the old way of forcing an extension:

```
nPeriodAt = RAT(".", m.cFullName)
IF nPeriodAt = 0
  cNewName = m.cFullName
ELSE
  cNewName = LEFT(m.cFullName, ;
               m.nPeriodAt - 1)
ENDIF
cNewName = m.cNewName + "." + m.cNewExtension

Here's the new way:

cNewName = FORCEEXT(m.cFullName, ;
               m.cNewExtension)
```

Again, the new way is faster across the board. As in the other tests, the slowest case is with a long path and no extension, but even in that case, the new way took only about two-third the time of the new way.

Put these functions to work

The functions for parsing and constructing filenames are not only faster in most cases, but make your code far more readable. Plan to use them in all new code.

What about older code? Should you go back and rewrite to use these functions? This is a question that will occur throughout this series. My general feeling is that you can leave older code alone unless it's giving you a problem or you have some other reason to touch the code in question. That is, when you're making changes to a program anyway, whether to fix a bug or just for refactoring, updating it to use the newer techniques makes sense. But if code is working and performing satisfactorily, leave it alone.

All of the code shown in this article is included in a single program file, UsingPathAndFileFunctions.PRG. My timing tests are in a second program, TimingPathAndFileFunctions.PRG.

Tamar E. Granor, Ph.D., is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of nine books including the award winning Hacker's Guide to Visual FoxPro and Microsoft Office Automation with Visual FoxPro. Her most recent books are Taming Visual FoxPro's SQL and What's New in Nine: Visual FoxPro's Latest Hits. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Certified Professional and a Microsoft Support Most Valuable Professional. Tamar speaks frequently about Visual FoxPro at conferences and user groups in North America and Europe, including every FoxPro DevCon since 1993. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.