Working with text

Tamar E. Granor, Ph.D.

VFP's tools for working with text have improved as the importance of text files has grown. In a world where we need to parse and create HTML, XML, and other text formats, using the best VFP has to offer makes the job a lot easier.

When I started working with FoxBase+ in the last 1980's, I rarely had to do anything with text files. But as FoxPro entered the Windows world, text, such as .INI files, assumed increasing importance. By the mid-90's, with the explosion of the Internet, the ability to handle text files became an important part of application development.

Mirroring those changes, FoxPro's tools for working with text files and the text within have improved over time. Even early versions of Fox products had the ability to parse and format text strings, and we've had a fairly straightforward way to read and write text files since FoxPro 1.0.

As with other areas, it's easy to keep using the techniques you learned long ago. Since VFP 6, however, there's been a veritable explosion of text handling techniques. This article looks at the first step in working with text, reading and writing it.

Reading and writing text files

When the low-level file functions (LLFFs) were introduced in FoxPro 1.0, I was intimidated by their name. "Low-level" sounded like something for the same people who used the LCK (Library Construction Kit) to build FLLs, not for mere mortals like me. Eventually, I needed to work with some external files and I learned that, in fact, the LLFFs were pretty much the same kind of mechanism I'd used to read and write data in other languages.

Even so, working with the LLFFs is tedious. You have to open the file with the right function, hang onto the handle that function returns, and make a series of function calls to actually read the data. Listing 1 shows code that reads the contents of a text file into a variable:

Listing 1. To read a text file with the low-level file functions, you open it and get a handle, then loop through until you run out of file.

```
* This code reads the file in blocks of
* 254 characters
nHandle = FOPEN(m.cFileName)
IF m.nHandle <> -1
   cContents = ""
   DO WHILE NOT FEOF(m.nHandle)
      cContents = m.cContents + ;
                  FREAD(m.nHandle, 254)
   ENDDO
   FCLOSE(m.nHandle)
ENDIF
```

Writing a text file is a little simpler because the FWRITE() function accepts the length of the string as a parameter, and can handle arbitrarily large strings. Listing 2 shows one way to write a text file with the LLFFs.

Listing 2. Writing a text file with the low-level file functions is simpler than reading one.

```
nHandle = FCREATE(m.cFileName)
IF m.nHandle <> -1
   nResult = FWRITE(m.nHandle, ;
             m.cContents, LEN(m.cContents))
ENDIF
FCLOSE(m.nHandle)
```

While these approaches work, VFP 6 introduced a pair of functions that virtually eliminate the need to use the LLFFs: FileToString() and StrToFile(). As their names suggest, they read a file into a string and write a string to a file, respectively. They also convert the code blocks above into single lines of code. Listing 3 shows how to use FileToString(), while Listing 4 demonstrates StrToFile().

Listing 3. With FileToString(), reading in a text file takes just one line.

```
cContents = FILETOSTR(m.cFileName)
```

Listing 4. StrToFile() turns writing a text file into a one-liner.

```
nResult = STRTOFILE(m.cContents, ;
          m.cFileName, .F.)
```

Why switch?

If you need to read text files regularly, by now, you've probably created your own wrappers for the LLFFs, so that you can read and write text files with a single call, so why would you switch to the newer functions?

For reading files, the answer is simple: speed. I tested the loop in Listing 1 against the single line in Listing 3 on a file with 710,000 characters. My test read the file in 1000 times. For the LLFFs, I also tested with a variety of block sizes (the second parameter to FREAD()) from 254 bytes to 2540. While a larger block size made a difference (with the largest block size, 1000 passes took about 80% of the time as with the smallest block size), FileToStr() was more than 30 times faster than the fastest LLFF attempt.

The speed advantage of FileToStr() also varies with the size of the target file. For tiny files, FileToStr() has almost no advantage, but even in the vicinity of 40KB files, FileToStr() is about 30% faster than the LLFFs.

Listing 5 shows my code for testing read speed.

Listing 5. FileToStr() is just about always faster than reading with the LLFFs. For large files, it's an order of magnitude or more faster.

```
* Compare LLFF with FileToStr()
#DEFINE PASSES 1000

* For LLFF, open file and read one
* line at a time.

LOCAL cFileName, nHandle, cContents
LOCAL nStart, nEnd, nPass
LOCAL nBlockPass, nBlockSize

cFileName = GETFILE("TXT;LOG")

* Try different block lengths
FOR nBlockPass = 1 TO 10
   nBlockSize = m.nBlockPass * 254
   nStart = SECONDS()

   FOR nPass = 1 TO PASSES
      nHandle = FOPEN(m.cFileName)
      IF m.nHandle <> -1
         cContents = ""
         DO WHILE NOT ;
            FEOF(m.nHandle)
            cContents = ;
            m.cContents + ;
            FREAD(m.nHandle,;
            m.nBlockSize)
         ENDDO
         FCLOSE(m.nHandle)
      ENDIF
   ENDFOR
   nEnd = SECONDS()

   ? " Using LLFF, result has ", ;
   LEN(m.cContents), " characters"
   ? " With block size = ", ;
   m.nBlockSize, ", total time = ",;
nEnd - nStart
ENDFOR
* For FileToStr(), one-liner
nStart = SECONDS()

FOR nPass = 1 TO PASSES
        cContents = FILETOSTR(m.cFileName)
ENDFOR
nEnd = SECONDS()

? " Using FILETOSTR(), result has ",;
  LEN(m.cContents), " characters"
? " Total time = ", nEnd - nStart
```

When writing text files, the case is murkier. For small files, StrToFile() is about a third faster than the LLFFs. However, when the string to write is more than 327,680 (which is 320 * 1024) characters, StrToFile() has a significant slowdown, and the LLFFs are faster. According to VFP MVP Christof Wollenhaupt, the difference is in the way VFP translates the calls to API calls. He says that StrToFile uses:

> "a single call to the WriteFile() API function passing the string as a parameter.
>
> "Hence, I assume that this is an issue with the API or a driver. VFP makes synchronous API calls. What I guess is happening behind the scenes is that the driver stores data in a buffer and then performs an asynchronous file operation. When the content exceeds the size of the buffer, the driver would have to complete the first operation, before adding the second part to the async buffer. Alternatively, large blocks might be written with a lower priority as to not to slow down the system too much.
>
> "You don't see the same issue with FWRITE(), because FWRITE() splits the string into blocks of 0x20000 bytes and is therefore always below the limit. The single FWRITE() call results in three calls to the WriteFile API function."

Breaking the string up into pieces no more than 327,680 characters and issuing multiple calls to StrToFile() doesn't improve matters (given Christof's explanation, this makes sense). In fact, in my tests, that approach was slower than the single call. The bottom line, therefore, is that if you might be writing large text files, you may want to stick with the LLFFs. If that's your choice, wrapping the write process up into a single function method is a good idea. Listing 6 shows my code for testing write speed.

Listing 6. StrToFile() has only a small advantage over LLFFs for small to medium files. For large files, FWRITE() is a better choice.

```
* Compare LLFF with StrToFile()
#DEFINE PASSES 100

LOCAL cFileName, cContents, nHandle
LOCAL nPass, nSTart, nEnd

cFileName = FORCEPATH("TestOutput.TXT",;
            SYS(2023))
SET SAFETY OFF

SET ALTERNATE TO StrToFileTiming.TXT
SET ALTERNATE ON

FOR nLength = 1 TO 10
   cContents = REPLICATE("Now is "+;
   "the time for all good men to "+;
   "come to the aid of their country." ;
   + CHR(13) + CHR(10), 1000 * m.nLength)

   ? * For LLFF, have to create file
   * and then send a bit at a time
   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      nHandle = FCREATE(;
            m.cFileName)
      IF m.nHandle <> -1
         nResult = FWRITE(;
         m.nHandle, ;
         m.cContents, ;
          LEN(m.cContents))
      ENDIF
      FCLOSE(m.nHandle)
   ENDFOR
```

```
  nEnd = SECONDS()

? " With LLFF, characters " + ;
  "written: ", m.nResult
? " Total time: ", ;
  m.nEnd - m.nStart

* For StrToFile(), one line
nStart = SECONDS()

FOR nPass = 1 TO PASSES
   nResult = STRTOFILE(;
         m.cContents, ;
         m.cFileName, .F.)
ENDFOR
nEnd = SECONDS()

? " With STRTOFILE(), "+ ;
"characters written: ", m.nResult
? " Total time: ";
   m.nEnd - m.nStart

* Try StrToFile() in loop,
* if past the magic number
IF LEN(m.cContents) > 327680
   LOCAL nPart, cPart
   nStart = SECONDS()

      FOR nPass = 1 TO PASSES
       DELETE FILE ;
       (m.cFileName)
       FOR nPart = 1 TO ;
       CEILING(;
       LEN(m.cContents);
       /327680)
       cPart = SUBSTR(;
       m.cContents, ;
       (nPart-1) * ;
       327680 + 1, 327680)
       nResult = ;
       STRTOFILE(;
       m.cPart, ;
       m.cFileName, .T.)
       ENDFOR
   ENDFOR
   nEnd = SECONDS()

      ? " With STRTOFILE() in "+;
      "a loop, characters " + ;
      written: ", m.nResult
      ? " Total time: ", m.nEnd -
m.nStart
   ENDIF
ENDFOR

SET ALTERNATE off
SET ALTERNATE TO
SET SAFETY ON
```

My test programs are included in this month's downloads as LLFFvsFileToStr.PRG and LLFFvsStrToFile.PRG.

What next?

Getting text data in and out of files is only the first step, of course. Once we have the data, we need to manipulate it in various ways. I'll look at old and new approaches for doing so over the next few issues.

Author Profile

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. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. 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