Splitting a Procedure File

It's easier to maintain separate program files rather than one monolithic procedure file. This utility makes it easy.

Tamar E. Granor, Ph.D.

Procedure files have been part of FoxPro since the early days. They allow you to bundle a whole group of procedures and functions into a single file. In the years before the Project Manager, a procedure file could make life easier. But since FoxPro 2.0, there's been no reason to use procedure files. In this article, I'll show a utility for deconstructing procedure files into separate PRGs.

I hate procedure files. I always have. They always seemed to me to make things harder rather than easier. When I can, I've always chosen to store each procedure or function in a separate PRG file named with the routine's name.

But most of my work is with existing applications. Often, they arrive with procedure files and usually, I simply leave them alone. But a few years ago, I started working on a project that included a procedure file that was over 3MB and contained more than 134,000 lines of code and over 1,600 routines. Code References choked on it and it confused Document View as to where its various routines began and ended. Aside from all that, there was evidence that many of the routines weren't needed in the project, so it was just cluttering things up.

Breaking up a procedure file of that size manually seemed like an overwhelming task, so I turned to VFP's ability to manipulate files and text.

The arguments against procedure files

Before we look at the code, let me talk a little about why I hate procedure files so much. The reasons are a mix of technical issues and convenience.

The first reason is convenience. With procedure files, I have to look two places to find a given routine. First I look for a PRG with the specified name. If I don't find it, then I have to open the procedure file and look for it there. If there's more than one procedure file (and almost every project I've inherited that uses procedure files has more than one), I may have to look at several before finding the routine of interest.

This issue got a little better with the Document View window, since I can quickly see whether a given routine is in a specified procedure file. (However, for that 3MB procedure file, it takes 10-15 seconds to open Document View or to refresh it when the file gets focus.)

However, now that I've integrated Thor's Go To Definition tool into my workflow, the problem is actually worse. I highlight the name of the routine I'm looking for and hit my hotkey. If the routine is a PRG, it opens; if it's in a procedure file, it doesn't. (To be fair, I just tested and found that if I SET PROCEDURE in the IDE, Go To Definition does find the routine.)

The second issue is dead weight. My experience is that procedure files only ever get bigger; routines that are no longer in use never get removed. So looking for what you need takes longer and longer. In addition, the whole procedure file goes into your EXE, whether every routine is used or not, likely making the EXE larger than it needs to be.

There's another problem with procedure files getting bigger all the time. You may end up with two or more versions of the same routine in the file. When that happens, VFP uses the last version it finds. This month's downloads include a simple demonstration. TestProcFile.PRG contains two versions of a routine called Repeated. Each version sends a different message to the Debug Output window. DemoDupRoutines.PRG contains the code in Listing 1. When you run the code, the message from the second version of the routine appears.

Listing 1. When a procedure file contains multiple routines with the same name, VFP uses the last one it finds.

```
SET PROCEDURE TO TestProcFile
DEBUG
?Repeated()
```

The problem with this, of course, is that in editing, you're likely to find and change the first instance of a given routine.

The final issue I have with procedure files is that it can sometimes be hard to get changes to "take" in testing. I do most of my testing from the

Command Window without building executables, but occasionally, I need to build and test with an EXE. When procedure files are involved, once I've run the EXE from the Command Window in that VFP session, if I then run the main program from the Command Window, changes to the procedure file are not reflected in the running code. The only solutions I've found are to delete the EXE or to close and restart VFP. On one project, I wasted a lot of time until I figured that out.

To demonstrate this problem, build the project DemoProcFile that's included into this month's downloads into an EXE. Then DO DemoProcFile.EXE from the Command Window. Next, open the procedure file and change the output in the second version of Repeated. Then, DO DemoDupRoutines from the Command Window. You'll see that your changed version does not run. Even issuing SET PROCEDURE TO (whether in the project code or from the Command Window) doesn't help.

For all these reasons and given that, in my view, procedure files give you no benefits, I avoid them.

Doing a split

It turned out to be pretty easy to write code to split a procedure file into separate PRG files. Listing 2 shows my initial code. The user is prompted to point to the procedure file. If a file is chosen, the AProcInfo() function is called; that function fills an array with a list of the "things" in the specified file. As called here, the array includes procedures, classes, methods and compiler directives.

The next step is to read the procedure file and break it into lines. The code then loops through the list of things and processes each one classified as "Procedure" (which, in fact, includes procedures, functions, methods and events). The first step in processing a routine is checking whether we already have a PRG with the specified name (which can happen either because it appears twice in the procedure file or because there's already a PRG with that name). If there isn't or the user says to overwrite it, we figure out which lines in the procedure file contain the routine (using the AProcInfo() results), and then build a string containing only those lines and save it as a file.

Listing 2. It doesn't take much code to split a procedure file into separate PRGs.

```
LOCAL cProcFile, aProcs[1], nProcs, nProc
LOCAL cProcName, cPath, cContent, nStartProc
LOCAL nEndProc, aProcLines[1], nTotalLines
LOCAL cProcText, nLine, cFileName, lProceed
LOCAL cMessage

cProcFile = GETFILE("prg","File name", ;
   "Split", 0, ;
   "Select procedure file to split")
IF NOT EMPTY(m.cProcFile) AND ;
   FILE(m.cProcFile)
  cPath = JUSTPATH(m.cProcFile)
  nProcs = APROCINFO(aProcs, m.cProcFile)

  * Read the whole file and split it in lines
  cContent = FILETOSTR(m.cProcFile)
  nTotalLines = ALINES(aProcLines, ;
     m.cContent)

  FOR nProc = 1 TO m.nProcs
    IF aProcs[m.nProc, 3] = "Procedure"
      cProcName = aProcs[m.nProc, 1]
      cFileName = FORCEPATH( FORCEEXT( ;
        m.cProcName, "prg"), m.cPath)
      * Prompt if we have an existing file
      IF FILE(m.cFileName)
        cMessage = m.cFileName + ;
          "already exists. Overwrite it?"
        IF MESSAGEBOX(m.cMessage, 4 + 32,;
          "Overwrite existing program?") = 6
          lProceed = .T.
        ELSE
          lProceed = .F.
        ENDIF
      ELSE
        lProceed = .T.
      ENDIF

      IF m.lProceed
        nStartProc = aProcs[m.nProc, 2]
        IF m.nProc < m.nProcs
          nEndProc = aProcs[m.nProc+1, 2] - 1
        ELSE
          nEndProc = m.nTotalLines
        ENDIF

        * Now grab the relevant lines
        cProcText = ''
        FOR nLine = m.nStartProc TO m.nEndProc
          cProcText = m.cProcText + ;
            aProcLines[m.nLine] + ;
            CHR(13) + CHR(10)
        ENDFOR

        * Save
        STRTOFILE(m.cProcText, m.cFileName, 0)
      ENDIF
    ENDIF
  ENDFOR

ENDIF

RETURN
```

You can probably think of lots of bells and whistles to add here. A few that came to mind as I wrote the description are displaying the existing routine when there's a duplicate, keeping a list of the routines skipped because of duplication, and keeping a list of routines created. None of those would be hard to add.

In addition, this code doesn't properly handle classes defined in a procedure file. (That was a conscious design decision, since I rarely inherit procedure files that include class definitions.)

More importantly, it ignores compiler directives. If the procedure file uses #INCLUDE or #DEFINE to make constants available, there will be problems with the new PRGs.

But for all these weaknesses, the code works pretty well, and it's very quick.

However, for the procedure file for which I wrote it, it failed in two ways. The first turns out to be a bug in AProcInfo(), which mirrors the bug that file demonstrates in Document View. When I run AProcInfo() on that file, the starting positions it shows for some routines are wrong. Both Document View and AProcInfo() get off by a line (that is, show the second line of the routine as the first line) nearly 10,000 lines into this monster, get off by another line after more than 35,000 lines total. By the time they both entirely gave up on this file after more than 87,000 lines, they're missing the first line of the routine by four lines. There are quite a few more routines after that, but neither the function nor the tool sees them. (This turned out not to be a bug in VFP. The file contained CHR(0); removing it allowed this code, as well as Document View, to see the whole file and eliminate the crash in Code References.)

When I first wrote the tool, I spent some time trying to figure out whether it was something other than the size of the file causing the problem, as well as trying to code around the problem. While I was trying to solve the problem of picking up the wrong lines, Jim Nelson suggested I also convert it into a Thor tool.

Splitting via Thor

I wrote about creating your own Thor tools in the March, 2013 issue, so I won't go back over what's necessary for that here. The complete code, including the part that tells Thor about the tool, is included in this month's downloads as Thor_Tool_Split.PRG. I'll describe how to add it to Thor later in this article.

The key portion of any Thor tool code is a procedure called ToolCode; that's what runs when the user chooses the tool. Much of the ToolCode procedure for this tool is the same as the code in Listing 2. However, it includes several improvements.

First, it handles compiler directives at the top of the procedure file correctly, adding them to the start of each new PRG. That's handled by the new DO WHILE loop that precedes the main FOR loop.

Second, it includes a fix for the AProcInfo() bug related to line numbers. The new FindDefLine function, discussed later in this article, handles this bug.

Third, because most of the procedure files I've worked with have comment blocks describing the routine before the PROCEDURE or FUNCTION line, it captures those lines and moves them to the new PRG as well. That's addressed by the two new DO WHILE loops inside the main FOR loop. The first goes backward from the line containing the PROCEDURE or FUNCTION, looking for empty lines and comment lines (those beginning with an asterisk—if you use one of the other comment notations, you'll need to modify the code). The default end position for the routine is the line immediately before the beginning of the next item in the file. The second DO WHILE loop works backward from that line, so that any trailing comments are omitted from this routine, as they're assumed to belong to the next item.

Listing 3 shows the ToolCode procedure.

Listing 3. The ToolCode procedure of the Thor version of the tool is similar to the original code.

```
LOCAL cProcFile, aProcs[1], nProcs, nProc
LOCAL cProcName, cPath, cContent
LOCAL nStartProc, nEndProc
LOCAL aProcLines[1], nTotalLines
LOCAL cProcText, nLine, cFileName, lProceed
LOCAL cMessage
LOCAL cDirectives
Local cLine, cWord2, nAdjust

cProcFile = GETFILE("prg","File name", ;
  "Split", 0, ;
  "Select procedure file to split")
IF NOT EMPTY(m.cProcFile) AND ;
   FILE(m.cProcFile)
  cPath = JUSTPATH(m.cProcFile)

  nProcs = APROCINFO(aProcs, m.cProcFile)

  * Read the whole file and split it in lines
  cContent = FILETOSTR(m.cProcFile)
  nTotalLines = ALINES(aProcLines, m.cContent)

  * Collect all compiler directives at top of
  * file for insertion into all new files
  cDirectives = ''
  nProc = 1
  DO WHILE nProc <= m.nProcs AND ;
    aProcs[m.nProc,3] = "Directive"
    cDirectives = ;
      aProcLines[aProcs[m.nProc,2]] + ;
      CHR(13) + CHR(10)
    nProc = m.nProc + 1
  ENDDO

  FOR nProc = 1 TO m.nProcs
    * Look only at procs and functions.
    * Don't include methods
    IF aProcs[m.nProc, 3] = "Procedure" AND ;
       NOT ("." $ aProcs[m.nProc, 1])
      cProcName = aProcs[m.nProc, 1]
      cFileName = FORCEPATH( FORCEEXT( ;
        m.cProcName, "prg"), m.cPath)
      * Prompt if we have an existing file
      IF FILE(m.cFileName)
        cMessage = m.cFileName + ;
          " already exists. Overwrite it?"
        IF MESSAGEBOX(m.cMessage, 4 + 32, ;
          "Overwrite existing program?") = 6
          lProceed = .T.
        ELSE
          lProceed = .F.
        ENDIF
      ELSE
        lProceed = .T.
      ENDIF
```

```
      IF m.lProceed
        nStartProc = aProcs[m.nProc, 2]

        * Make sure we have the actual PROC or
        * FUNC line. There's a bug in
        * AProcInfo() that sometimes specifies
        * the first line as too low down.
        nStartProc = FindDefLine( ;
          m.cProcName, m.nStartProc, ;
          aProcs[m.nProc,3], @aProcLines)

        * Search backward for comment lines
        DO WHILE nStartProc > 1 AND ;
          (EMPTY(aProcLines[m.nStartProc-1]) ;
          OR LEFT(aProcLines[m.nStartProc-1],;
                  1) = "*")
          nStartProc = m.nStartProc - 1
        ENDDO

        IF m.nProc < m.nProcs
          * Find actual start of next proc
          nEndProc = FindDefLine( ;
            aProcs[m.nProc + 1, 1], ;
            aProcs[m.nProc+1, 2], ;
            aProcs[m.nProc+1, 3], ;
            @aProcLines) - 1

        ELSE
          nEndProc = m.nTotalLines
        ENDIF

        * Search backward to ignore trailing
        * comment lines
        DO WHILE nEndProc > m.nStartProc AND ;
          (EMPTY(aProcLines[m.nEndProc]) OR ;
          LEFT(aProcLines[m.nEndProc],1) = ;
            "*")
          nEndProc = m.nEndProc -1
        ENDDO


        * Now grab the relevant lines
        cProcText = m.cDirectives
        FOR nLine = m.nStartProc TO m.nEndProc
          cProcText = m.cProcText + ;
            aProcLines[m.nLine] + ;
            CHR(13) + CHR(10)
        ENDFOR

        * Save
        STRTOFILE(m.cProcText, m.cFileName, 0)
      ENDIF
    ENDIF
  ENDFOR

ENDIF

RETURN
```

FindDefLine returns the line number in the file on which the specified information (whether it's a routine, a class definition, or some kind of compiler directive) actually begins, correcting for the bug in AProcInfo(). The function receives the name of the item, the line it's supposed to start on, the type of item, and the array containing all the lines in the procedure file. It builds a logical condition to identify the correct line (or, more accurately, to identify lines that cannot be the correct line) and then loops backwards from the specified line until it finds a line that qualifies as the right one. The code is shown in Listing 4.

Listing 4. This function corrects for the bug in AProcInfo() by searching backward to find the real first line of the routine.

```
PROCEDURE FindDefLine(cProcName, ;
  nStartsOn, cType, aProcLines)
* Find the actual line on which the specified *
proc starts. It may be nStartsOn, but due to
* a bug in AProcInfo, might be an earlier
* line.

LOCAL cLine, cWord2, nAdjust
LOCAL cDefinitionCondition

DO CASE
CASE m.cType = 'Procedure'
  cDefinitionCondition = ;
   [NOT INLIST(LEFT(m.cLine,4),"PROC","FUNC");
   OR NOT (m.cWord2 == UPPER(m.cProcName))]

CASE m.cType = 'Class'
  cDefinitionCondition = ;
   [NOT ("DEFI"$GETWORDNUM(m.cLine,1) ;
   AND m.cWord2 == "CLASS") ];
   + [OR NOT GETWORDNUM(m.cLine,3) == ;
   UPPER(ALLTRIM(STREXTRACT(m.cProcName,'', ;
   'AS')))]

CASE m.cType = 'Directive'
  cDefinitionCondition = ;
   [NOT (LEFT(m.cLine,1) = "#" ] + ;
   [OR NOT INLIST(GETWORDNUM(m.cLine,1), ;
   "INCLUDE", "IF", "ELIF", "ELSE", "ENDIF", ;
   "IFDEF", "IFNDEF", "UNDEF"))]

CASE m.cType = 'Define'
  cDefinitionCondition = ;
   [NOT (LEFT(m.cLine,1) = "#" OR ;
   NOT (m.cWord2 == UPPER(m.cProcName))]
ENDCASE

nAdjust = 0
cLine = ;
  UPPER(ALLTRIM(aProcLines[m.nStartsOn]))
cWord2 = GETWORDNUM(m.cLine,2)
IF "(" $ m.cWord2
  cWord2 = STREXTRACT(m.cWord2, '', '(')
ENDIF
DO WHILE &cDefinitionCondition
  nAdjust = m.nAdjust + 1
  cLine = ;
    UPPER(aProcLines[m.nStartsOn - m.nAdjust])
  cWord2 = GETWORDNUM(m.cLine,2)
  IF "(" $ m.cWord2
    cWord2 = STREXTRACT(m.cWord2, '', '(')
  ENDIF
ENDDO

RETURN m.nStartsOn - m.nAdjust
```

Adding the tool to Thor

Adding this tool to Thor is easy, assuming you have Thor installed. Make any changes you want to the tool's PRG (such as what submenu you want it on or the prompt you'll see). Then from the Thor menu, choose More | Open Folder | My Tools. Drop the PRG into that folder and restart VFP (or just restart Thor) and tool will be available.

If you're not using Thor (why not?), you can extract the code from the ToolCode routine into a standalone PRG and use the tool directly. There are no dependencies on theThor framework.

Improving the tool

In addition to the items I mentioned in "Doing a split" earlier in this article, you might want to modify the tool to operate on an open file rather than having to point to it. (That would a good use of the Thor framework.)

You might also want to modify the code so that it only puts the directives you need into a given PRG. If you're dealing only with #DEFINE, that's pretty easy; just search the code in the routine to see if the specified constant appears. However, for include files, you'd have to do more work; AProcInfo() can give you a hand there, as you can ask it only for directives, so you could apply it to the Include file and then search the code for each routine to see whether any of those definitions appear.

Let me know if you think of any other useful extensions.

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. Tamar is author or co-author of a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro and Taming Visual FoxPro's SQL. Her latest collaboration is VFPX: Open Source Treasure for the VFP Developer, available at www.foxrockx.com. Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.