Zip it, Zip it Good

Doug Hennig

This month's article presents a class to zip and pack the files in a project, and a class to interface VFP with a zipping utility like WinZip. Combining these classes with a ProjectHook and a project toolbar helps improve your productivity.

In my column in the September 1998 issue of FoxTalk ("The Happy Project Hooker", a article for which I received way more comments about the title than the article itself <g>), I discussed the ProjectHook class, a welcome addition to VFP 6 that makes it easier to build project-related tools for developers. I also presented several ProjectHook classes that provided functionality like build buttons in a toolbar, text searching, and restoring project-specific settings like Intellidrop classes and form templates. I mused that other features would be useful, like packing the table-based source files (SCX, VCX, FRX, and MNX) in a project or creating a zip file for archive or transportation purposes. Since I've actually built these functions, it's time to present these as reusable developer tools.

As a refresher, SFPROJ.VCX contains several ProjectHook classes. SFProjectHook is the base ProjectHook class; it contains core functionality like chaining to hooked ProjectHook (or other) objects and implementing a toolbar if desired. SFProjectHookFind is a subclass that implements text search capabilities; it collaborates with SFFindText, a form that displays where the specified text was found. SFProjectHookRegistry saves and restores project-specific settings, making it easy to switch between projects. SFProjectToolbar, SFProjectToolbarTimer, and several SFToolbarButton classes provide a toolbar that works with a ProjectHook object and automatically releases itself when the project is closed.

SFProjectUtils

Rather than creating another ProjectHook subclass for project packing and zipping, I decided to create a project utilities class (SFProjectUtils) based on Custom instead. Although this class is in SFPROJ.VCX like the ProjectHook classes, I wanted this class to be used for projects that aren't necessarily open in the Project Manager, so I didn't need the functionality of a ProjectHook. The reason I based it on Custom rather than SFCustom (our Custom base class in SFCTRLS.VCX) is that having SFCTRLS.VCX open (which happens when you instantiate a subclass of one of its classes, even when that subclass is in a different VCX) when trying to pack or zip a project that includes this file causes problems. Just because it isn't based on ProjectHook doesn't mean it can't be used from within a ProjectHook; that's in fact what we'll do later in this article.

SFProjectUtils has two main methods, and several supporting methods. The main methods are ZipProject, which creates a zip file of the files in a project, and PackProject, which packs the table-based source files in a project. The supporting methods, which we won't look at due to space limitations, are GetFilesFromProject, which fills an array with the files in a project, GetDateTime, which combines the date and time values from an array filled with ADIR() into a DateTime value, and GetAssociatedFile, which returns the name of a file associated with the specified file (for example, passing TEST.SCX to GetAssociatedFile will return TEST.SCT). SFProjectUtils has several properties, the two important ones being cProject, which contains the name of the project file we're working with, and oZip, which contains a reference to an object that'll actually perform the zipping work for us.

Let's look at PackProject first, since it's the simpler of the two main methods. I like to use this method before I build an EXE or archive a project because, as you probably know, the memo file of table-based source code files (such as VCTs) can get huge with discarded data as you make changes to it. Although the Project Manager will compact these files when you build with the "Recompile All Files" option chosen, I don't use that option often. As a result, the EXE or archive can be bloated with garbage. PackProject provides a fast way to trim the files down to size.

The major code in this method gets a list of files in the project, opens those that are table-based (the file type is K [screen], V [VCX], R [report], B [label], or M [menu]) exclusively, and packs them.

```
lnFiles  = This.GetFilesFromProject(@laFiles, lcProject)
for lnI = 1 to lnFiles
  if laFiles[lnI, 2] $ 'KVRBM'
```

```
    lcName = laFiles[lnI, 1]
    if file(lcName)
      wait window 'Processing ' + lcName + ' ...' nowait
      use (lcName) exclusive
      pack
      use
    endif file(lcName)
  endif laFiles[lnI, 2] $ 'KVRBM'
next lnI
wait clear
```

PACKPROJ.PRG is a simple utility program you can call to pack the files in a project. Pass it the name of the project to pack and it'll let SFProjectUtils take care of the work.

The ZipProject method zips the files in a project. I use it both to archive the project once it's completed and for transportation purposes (taking a copy of the files home or giving them to another developer). ZipProject accepts two parameters: the name of the zip file to create and, because you may not want to zip all the files in a project, an expression to filter the files to zip. We'll look at how the filter expression is used in a moment. We won't examine all the code in this method, just the important stuff. First, we get a list of files in the project, the directory the project is in, and the name of the APP or EXE file created from the project (if it exists).

```
with This
  lcProject = .cProject
  if empty(lcProject)
    return .F.
  endif empty(lcProject)

* Get a list of files in the project.

  lnFiles = .GetFilesFromProject(@laFiles, lcProject)

* Get the directory and name of the APP file.

  lcDirectory = addbs(justpath(lcProject))
  lcAppFile   = forceext(lcProject, 'APP')
  if not file(lcAppFile)
    lcAppFile = forceext(lcProject, 'EXE')
  endif not file(lcAppFile)
```

Next, we put information about the project file into an object and pass that object to the ShouldFileBeZipped method, which returns .T. if the filter expression is either empty (meaning there is no filter) or evaluates to .T. If it does, the AddFile method of the oZip object adds the PJX and PJT files to the list of files to zip. If the APP or EXE file created from the project exists, it's treated the same way.

```
  adir(laDir, lcProject)
  loFile.cFileName   = lcProject
  loFile.cType       = 'p'
  loFile.nFileSize   = laDir[2]
  loFile.tModified   = .GetDateTime(laDir[3], laDir[4])
  loFile.cAttributes = laDir[5]
  if .ShouldFileBeZipped(loFile, tcFilterExpr)
    .oZip.AddFile(lcProject)
    .oZip.AddFile(forceext(lcProject, 'PJT'))
  endif .ShouldFileBeZipped(lcProject, ...
  if file(lcAppFile)
    adir(laDir, lcAppFile)
    loFile.cFileName   = lcAppFile
    loFile.cType       = ''
    loFile.nFileSize   = laDir[2]
    loFile.tModified   = .GetDateTime(laDir[3], laDir[4])
    loFile.cAttributes = laDir[5]
    if .ShouldFileBeZipped(loFile, tcFilterExpr)
      .oZip.AddFile(lcAppFile)
    endif .ShouldFileBeZipped(loFile, tcFilterExpr)
  endif file(lcAppFile)
```

Each of the files in the project is also added to the list of files to zip if it passes the filter criteria. Because a file may have one or two associated files (for example, DCX and DCT files accompany a DBC file), the GetAssociatedFile method is called to ensure all necessary files are included.

```
  for lnI = 1 to lnFiles
    lcFile               = laFiles[lnI, 1]
    lcType               = laFiles[lnI, 2]
    loFile.cFileName     = lcFile
    loFile.cType         = lcType
    loFile.nFileSize     = laFiles[lnI, 3]
    loFile.tModified     = laFiles[lnI, 4]
    loFile.cAttributes   = laFiles[lnI, 5]
    if .ShouldFileBeZipped(loFile, tcFilterExpr)
      .oZip.AddFile(lcFile)
      for lnJ = 1 to 2
        lcFile = .GetAssociatedFile(lcFile, lcType, lnJ)
        if not empty(lcFile)
          .oZip.AddFile(lcFile)
        endif not empty(lcFile)
      next lnJ
    endif .ShouldFileBeZipped(lcFile, ...
  next lnI
```

Finally, if any of the files passed the filter criteria, we close the project file if necessary (we can't zip it if it's open in the Project Manager), have the oZip object do the zipping work, and reopen the project if we closed it.

```
  if .oZip.nFiles > 0
    loProject = iif(type('_vfp.Projects[lcProject].' + ;
      'Name') = 'C', _vfp.Projects[lcProject], .NULL.)
    llOpen = vartype(loProject) = 'O'
    if llOpen
      loProject.Close()
    endif llOpen
    llReturn = .oZip.ZipFiles(tcZipFile)
    if llOpen
      modify project (lcProject) nowait
    endif llOpen
  endif .oZip.nFiles > 0
endwith
return llReturn
```

Before we look at the zipping object, let's take a look at an example of how ZipProject might be called. I like to create two zip files for each project. One, called SOURCE, contains the project file, APP or EXE, and all the project-specific source code. The other, called LIBRARY, contains the library (or non-project-specific) source code used in the project. The reason for zipping LIBRARY is that sometimes you may need to restore an old project to do maintenance work on it but you've since updated the library files it uses to a later version. Rather than trying to figure out what changes in the library files will cause havoc in your old project, it's easier to just use the snapshot of the library files as they were when the project was archived.

ZIPPROJ.PRG (included with this month's Subscriber downloads) accomplishes this goal. Pass it the name of the project to zip and it'll create SOURCE.ZIP and LIBRARY.ZIP. The interesting part of this program is the following code (loProject is a reference to the SFProjectUtil object and lcProject contains the name of project to process):

```
pcDirectory = addbs(justpath(lcProject))
loProject.ZipProject(pcDirectory + 'source.zip',  ;
  'ZipSourceFile(toFile)')
loProject.ZipProject(pcDirectory + 'library.zip', ;
  'ZipLibraryFile(toFile)')
```

Notice that for the filter expressions, this code passes function names. These functions are located in ZIPPROJ.PRG, which means that the SFProjectUtil object in effect calls back to ZIPPROJ to figure out which files should be zipped. The ZipSourceFile function will allow the project file (toFile.cType = "p"), application file (toFile.cType is blank), or any file in the project directory or a subdirectory of it to be

zipped, while ZipLibraryFile will only allow files that aren't in the project directory or a subdirectory of it to be zipped:

```
function ZipSourceFile
lparameters toFile
return toFile.cType = 'p' or empty(toFile.cType) or ;
  addbs(justpath(toFile.cFileName)) = pcDirectory

function ZipLibraryFile
lparameters toFile
return addbs(justpath(toFile.cFileName)) <> pcDirectory
```

What if you don't want this functionality? Easy: change the filter conditions. Mike Yearwood (who I owe thanks to, both for the following code and for helping make the classes in this article better) wanted a single zip file containing only files changed since the last time the project was zipped so he could send just those files to someone else. MIKEZIPPER.PRG uses a table that contains the date and time the project was last zipped (which is placed into the ptLastTimeZipWasMade variable), and uses this as the filter function:

```
function ZipSourceFile
lparameters toFile
return toFile.tModified > ptLastTimeZipWasMade
```

SFZipUtils and SFZipUtilsWinZip

Now let's look at the classes that handle the zipping process for us. Because the exact process of how to zip files depends on what tool you use (several tools are available, including WinZip, PKZip, DynaZip, and several free or shareware utilities), I created an abstract class called SFZipUtils (based on Custom rather than SFCustom for the same reason I outlined earlier) that has the framework for a zipping class and leaves the implementation to a subclass.

SFZipUtils has an AddFile method that adds a filename to the protected aFiles array. Call this method, passing it the name and path of the file, for each file to be zipped. The nFiles property contains the number of files to be zipped; it has an assign method that makes it read-only and an access method that calculates the value for the property. The Clear method simply clears the aFiles array; call this method after zipping some files when you want to zip a different set of files. The cZipFile property contains the name of the zip file to create; it has an assign method to ensure a valid filename is entered. The lUseRelativePaths property indicates whether relative paths should be used in the zip file; set it to .F. to use absolute paths.

The ZipFiles method uses a Template design pattern to create the zip file. It doesn't do a lot of work itself; rather, it directs other methods to do the work. The following is a stripped down version of the code for space reasons (see the source code for the complete version):

```
lparameters tcZipFile
with This

* Hook method before the list of files to zip has been
* created.

  .BeforeAddFilesToZip()

* Add each file to the zip file.

  lnFiles = .nFiles
  for lnI = 1 to lnFiles
    lcFile = .aFiles[lnI]
    if .lUseRelativePaths
      lcFile = sys(2014, lcFile, lcZipFile)
    endif .lUseRelativePaths
    .AddFileToZip(lcFile)
  next lnI

* Hook method after the list of files to zip has been
* created.

  .AfterAddFilesToZip()
```

```
* Hook method before the zip file has been created.

  .BeforeCreateZipFile()

* Ensure we're in the directory where the zip file
* should be created (it works better if we're using
* relative file paths) and zip the files.

  lcCurDir = sys(5) + curdir()
  cd (justpath(lcZipFile))
  llReturn = .CreateZipFile(lcZipFile)
  cd (lcCurDir)

* Hook method after the zip file has been created.

  .AfterCreateZipFile()
endwith
return llReturn
```

The BeforeAddFilesToZip, AfterAddFilesToZip, BeforeCreateZipFile, and AfterCreateZipFile methods called from ZipFiles are hook methods; they don't have any code in this class, but can be used to place additional code at the appropriate place in the process in a subclass. AddFileToZip and CreateZipFile are also empty in this class, but they're not hook methods; they're abstract methods that must be coded in a subclass to provide the behavior for the zipping mechanism used.

Since I have WinZip (from Nico Mak Computing Inc, www.winzip.com), I created a subclass of SFZipUtils called SFZipUtilsWinZip that uses WinZip to create the zip file. This subclass creates a file containing the list of files to be zipped (the name of this file is passed to WinZip as we'll see later), so the BeforeAddFilesToZip and AfterAddFilesToZip methods were overridden to create and close a textmerge file, respectively. AddFileToZip was overridden to output the name of the specified file to the textmerge file:

```
lparameters tcFile
\\<<tcFile + chr(13) + chr(10)>>
```

The CreateZipFile method, which actually creates the zip file, calls WINZIP.EXE (the actual name and path to the program are stored in the cZipProgram property) to do the work. WinZip is passed the following parameters: "-min" to run minimized, "-a" to add files, "-p" to save directory information (rather than just the filename itself), the name of the zip file, and "@<filename>", where <filename> is the name of the textmerge file (stored in the cZipList property) containing the list of files to zip. Initially, I had this code call WINZIP.EXE using the VFP RUN command. However, the problem with this is that RUN returns immediately even though WinZip hasn't finished executing. I wanted to wait until it was done to be sure the zip file was created, so I used a great public domain class called Process (included in this month's Subscriber downloads), written by Ed Rauh of eSolutions Services, LLC. Set the icCommandLine property of this class to the command line to execute and the icWindowMode property to the mode for the application ("HID" means hidden; see the Process class for other choices), then call the LaunchAppAndWait method to execute the application and wait until it terminates before continuing. The Init method of SFZipUtilsWinZip instantiates Process into the oProcess property. OK, now that this description took way more space than the actual code for CreateZipFile <g>, here's the code:

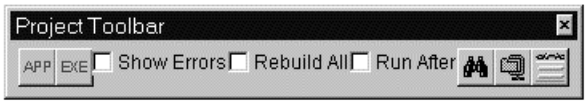
```
lparameters tcZipFile
with This
  .oProcess.icCommandLine = .cZipProgram + ;
    ' -min -a -p ' + tcZipFile + ' @' + .cZipList
  .oProcess.icWindowMode  = 'HID'
  .oProcess.LaunchAppAndWait()
endwith
```

Before using SFZipUtilsWinZip, don't forget to set the cZipProgram property to the path to WINZIP.EXE on your system.

Adding to a ProjectHook

To make it easier to access the methods in SFProjectUtils, I created a subclass of SFProjectHookFind (I used this class because I also want text search capability) called MyProjectHook (in SFPROJECT.VCX) that instantiates SFProjectUtils into its oUtils property and has PackProject and ZipFiles methods that call oUtils.PackProject and oUtils.ZipProject, respectively. It also has ZipSourceFile and ZipLibraryFile methods with the same code as the functions in ZIPPROJ.PRG so file filtering creates the desired zip files. I created a subclass of SFProjectToolbar called MyProjectToolbar (also in SFPROJECT.VCX) that has buttons for building an APP or EXE, checkboxes for various build options, and buttons to perform a text search or zip or pack the project (see Figure 1). These buttons call the appropriate methods in MyProjectHook, which instantiates MyProjectToolbar because its cToolbarClass property contains "MyProjectToolbar".

Figure 1. MyProjectToolbar provides fast access to project functions.



Conclusion

Project utilities like those I presented in my September 1998 article and this article make working with project files a lot easier. I use these utilities all the time, and hope you find them as useful as I do.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.