Long Live PRGs!

Doug Hennig

Although classes get most of the attention today, there's still a place for PRGs. This month's article looks at some library routines packaged as PRGs.

Although I have a class that provides a large number of general library routines, I also have some library routines that exist as PRGs. Why do PRGs still have a place in an object-oriented environment like VFP?

- PRGs are easier to use since there's nothing to instantiate.
- Functions are easier to call, since you leave off the "Object." syntax.
- Functions in PRGs execute a little faster than methods of objects.
- User-defined function (UDF) calls "look" like native function calls (the next paragraph explains why this can be a good thing).

We'll look at nearly a dozen little PRGs that provide useful enough functionality that I use them in nearly every application and tool I write. Many of these functions are named the same as native VFP 6 functions so they implement the same behavior in VFP 5. Why do I care about VFP 5? Believe it or not, a lot of developers haven't upgraded from VFP 5 to 6 yet, and as a developer of VFP tools, I don't want to limit my market to just those that have the latest version. Thus, these routines means I don't need to have one version of my code for VFP 5 and one for VFP 6; when a function like VARTYPE() is called in VFP 6, the native function will be used, but in VFP 5, VARTYPE.PRG will be called instead.

FOXTOOLS Functions

VFP 6 added as native functions a number of filename processing routines built into FOXTOOLS.FLL, eliminating the need to load this library. These functions are very useful anytime you need to do something with a filename or path. For example, I often use temporary files for certain processes, and then want to delete them afterward, so I use code like this:

```
lcTempFile = sys(3) + '.DBF'
* create and do something with this file
erase (lcTempFile)
erase forceext(lcTempFile, 'CDX')
erase forceext(lcTempFile, 'FPT')
```

FORCEEXT() forces the specified extension onto a filename, so the CDX and FPT files for the table can be deleted.

I've created PRG versions of most of the filename processing functions so I can use these functions without worrying about whether someone will use my code in VFP 5. We've already discussed FORCEEXT(); here's the code:

```
lparameters tcName, ;
  tcExtension
local lcPath, ;
  lcName
lcPath = addbs(justpath(tcName))
lcName = juststem(tcName) + iif(empty(tcName) or ;
  empty(tcExtension), '', '.' + tcExtension)
return lcPath + lcName
```

FORCEEXT.PRG uses three other "FOXTOOLS" functions: ADDBS() (if you've read my articles before, you're probably thinking I use this function a lot <g>), which adds a backslash to the end of a path; JUSTPATH(), which returns just the drive and path of a filename (without a trailing backslash, which is why ADDBS() is used here); and JUSTSTEM(), which returns just the stem (the part of the name before the period and with no drive and path) of a filename. Here's the code for ADDBS.PRG:

```
lparameters tcName
local lcName
lcName = alltrim(tcName)
return lcName + iif(right(lcName, 1) <> '\' and ;
  not empty(lcName), '\', '')
```

Here’s JUSTPATH.PRG:

```
lparameters tcName
return left(tcName, rat('\', tcName))
```

This is the code for JUSTSTEM.PRG:

```
lparameters tcName
local lcName
lcName = substr(tcName, rat('\', tcName) + 1)
return iif('.' $ lcName, ;
  left(lcName, rat('.', lcName) - 1), lcName)
```

JUSTFNAME() returns a filename (stem and extension) without drive and path:

```
lparameters tcName
return substr(tcName, rat('\', tcName) + 1)
```

The last one, JUSTEXT.PRG, returns just the extension of a filename:

```
lparameters tcName
local lcName
lcName = justfname(tcName)
return iif('.' $ lcName, ;
  substr(lcName, rat('.', lcName) + 1), '')
```

In case you’re wondering why the call to JUSTFNAME() is necessary in this routine, think about what would be returned if the path had a period in it but the filename had no extension.

Other VFP 6 Replacement Functions

I use two other functions new to VFP 6 a lot: NEWOBJECT(), which instantiates an object even when the class library it’s in isn’t open, and VARTYPE(), a faster and simpler version of TYPE(). As with the “FOXTOOLS” functions, I’ve created PRGs that provide the same functionality in VFP 5. Here’s NEWOBJECT.PRG; I arbitrarily chose six as the number of parameters that can be passed to the new object.

```
lparameters tcClass, ;
  tcLibrary, ;
  tcInApp, ;
  tuParm1, ;
  tuParm2, ;
  tuParm3, ;
  tuParm4, ;
  tuParm5, ;
  tuParm6
local lcLibrary, ;
  lcInApp, ;
  loObject

* If the class library is specified, SET CLASSLIB to it
* if necessary.

do case
  case type('tcLibrary') <> 'C' or empty(tcLibrary) or ;
    upper(tcLibrary) $ upper(set('CLASSLIB'))
    lcLibrary = ''
  case file(tcLibrary)
    lcLibrary = upper(tcLibrary)
  otherwise
    lcLibrary = upper(locfile(tcLibrary, ;
      'Visual Class Library (*.vcx):VCX;Program ' + ;
      '(*.prg):PRG', tcLibrary))
endcase
if not empty(lcLibrary)
  lcInApp = iif(type('tcInApp') = 'C' and ;
    not empty(tcInApp), 'in ' + tcInApp, '')
  set classlib to (lcLibrary) &lcInApp additive
endif not empty(lcLibrary)

* Create the object.

do case
  case pcount() < 4
    loObject = createobject(tcClass)
  case pcount() = 4
    loObject = createobject(tcClass, @tuParm1)
  case pcount() = 5
    loObject = createobject(tcClass, @tuParm1, ;
      @tuParm2)
  case pcount() = 6
    loObject = createobject(tcClass, @tuParm1, ;
      @tuParm2, @tuParm3)
  case pcount() = 7
    loObject = createobject(tcClass, @tuParm1, ;
      @tuParm2, @tuParm3, @tuParm4)
  case pcount() = 8
    loObject = createobject(tcClass, @tuParm1, ;
      @tuParm2, @tuParm3, @tuParm4, @tuParm5)
  case pcount() = 9
    loObject = createobject(tcClass, @tuParm1, ;
      @tuParm2, @tuParm3, @tuParm4, @tuParm5, @tuParm6)
endcase

* Release the library if we opened it (and it's still
* open) and return a reference to the object we created.

if not empty(lcLibrary) and ;
  lcLibrary $ upper(set('CLASSLIB'))
  release classlib (lcLibrary)
endif not empty(lcLibrary) ...
return loObject
```

The code for VARTYPE.PRG is quite simple. Notice that VARTYPE() returns "X" for a .NULL. value and the data type for anything else.

```
lparameters tuObject
return iif(isnull(tuObject), 'X', type('tuObject'))
```

A Better NEWOBJECT()

The native VFP 6 NEWOBJECT() function has one behavior that annoys me, to the point of making it essentially useless for my purposes: if you pass it a class library, the function ignores any open class libraries and insists on looking for the specified library file. Why is this a problem? Well, I frequently run code from the Command window for testing purposes (and the tools I build are often designed to work within the VFP development environment). I don't want to hard-code a path in my calls to NEWOBJECT() (or else the code would only work on my machine, and only if I don't move things around) so I just specify the class library without a path, which works perfectly in a runtime environment since all the files are built into the EXE. To help locate the library in a development environment, and for performance reasons, I use SET CLASSLIB to open the class library prior to instantiating classes from it, but can't expect that others who use my code have done the same thing. So, if NEWOBJECT() would simply look at which class libraries were already open, it would do exactly what I need under all conditions: if the class library is in the path or EXE and not already open, it'll be temporarily opened, and if the class library isn't in the path but is already open, it'll be used. Unfortunately, in the latter case, it doesn't use the open library; it gives an error that the class library cannot be found.

Necessity is the mother of invention, so I created MAKEOBJECT.PRG. This function uses the same set of parameters as NEWOBJECT() but has the behavior I need. If you pass it a class library and that library is already open, it'll be used (it does this by passing an empty class library name to NEWOBJECT(), which in that case acts just like CREATEOBJECT(), expecting the class library is already open). Otherwise, it tries to find the class library and presents a dialog for you to locate it if it can't. Here's the code:

```
lparameters tcClass, ;
  tcLibrary, ;
  tcInApp, ;
  tuParm1, ;
  tuParm2, ;
  tuParm3, ;
  tuParm4, ;
  tuParm5, ;
  tuParm6
local lcLibrary, ;
  llLibrary, ;
  lnParms, ;
  loObject
lcLibrary = iif(empty(tcLibrary) or ;
  upper(tcLibrary) $ set('CLASSLIB') or ;
  upper(tcLibrary) $ set('PROCEDURE'), '', tcLibrary)
llLibrary = empty(lcLibrary) or file(tcLibrary) or ;
  file(tcLibrary + '.VCX') or ;
  file(tcLibrary + '.PRG') or file(tcLibrary + '.FXP')
if not llLibrary
  lcLibrary = locfile(lcLibrary, ;
    'Visual Class Library (*.vcx):VCX;Program ' + ;
    '(*.prg):PRG', lcLibrary)
  llLibrary = not empty(lcLibrary)
endif not llLibrary
lnParms = pcount()
do case
  case lnParms = 1
    loObject = createobject(tcClass)
  case not llLibrary
    loObject = .NULL.
  case lnParms = 2
    loObject  = newobject(tcClass, lcLibrary)
  case lnParms = 3
    loObject  = newobject(tcClass, lcLibrary, tcInApp)
  case lnParms = 4
    loObject  = newobject(tcClass, lcLibrary, tcInApp, ;
      @tuParm1)
  case lnParms = 5
    loObject  = newobject(tcClass, lcLibrary, tcInApp, ;
      @tuParm1, @tuParm2)
  case lnParms = 6
    loObject  = newobject(tcClass, lcLibrary, tcInApp, ;
      @tuParm1, @tuParm2, @tuParm3)
  case lnParms = 7
    loObject  = newobject(tcClass, lcLibrary, tcInApp, ;
      @tuParm1, @tuParm2, @tuParm3, @tuParm4)
  case lnParms = 8
    loObject  = newobject(tcClass, lcLibrary, tcInApp, ;
      @tuParm1, @tuParm2, @tuParm3, @tuParm4, @tuParm5)
  case lnParms = 9
    loObject  = newobject(tcClass, lcLibrary, tcInApp, ;
      @tuParm1, @tuParm2, @tuParm3, @tuParm4, @tuParm5, ;
      @tuParm6)
endcase
return loObject
```

Locating Application Pieces

Often, an application needs to find additional files, such as FRX, FLL, or graphic files, in the same directory as the application is located. The problem is that the application directory may not be the current directory, and the method to determine the directory the application is located in depends on whether the application is an in-process DLL server, out-of-process EXE server, or standalone EXE. So, I use GETAPPDIRECTORY.PRG to determine the directory the application is in. This routine is adapted from code posted in a forum by Rick Strahl.

```
local lcProgram, ;
  lcPath, ;
  lcFileName, ;
  lnBytes
lcProgram = sys(16, 0)
do case

* In-process DLL server or Active Document.

  case atc('.VFD', lcProgram) > 0 or ;
    Application.StartMode = 3
    lcPath = home()

* Out-of-process EXE server.

  case Application.StartMode = 2
    declare integer GetModuleFileName in Win32API ;
      integer hInst,;
      string @lpszFileName,;
      integer @cbFileName
    lcFileName = space(256)
    lnBytes    = 255
    GetModuleFileName(0, @lcFilename, @lnBytes)
    lnBytes    = at(chr(0), lcFilename)
    lcFileName = iif(lnBytes > 1, ;
      substr(lcFileName, 1, lnBytes - 1), '')
    lcPath     = justpath(lcFileName)

* Standalone EXE or VFP development.

  otherwise
    lcPath = justpath(lcProgram)
    if atc('PROCEDURE', lcPath) > 0
      lcPath = substr(lcPath, rat(':', lcPath) - 1)
    endif atc('PROCEDURE', lcPath) > 0
endcase
return addbs(upper(lcPath))
```

Loading Images

If, like me, you instantiate ActiveX controls at runtime rather than design time (see my June 1998 article for a class that can help with this), you know that most ActiveX controls need an image object rather than the name of the graphic file. So, you use the VFP LOADPICTURE() function to open a graphic file and return an object reference to the image. However, LOADPICTURE() has a bug: if the graphic file is included in the EXE, LOADPICTURE() gives an OLE error even if the graphic file is also available on disk. So, if you use an ICO file as the icon for a form (in which case VFP will automatically include it in the EXE) and also use that same image in an ActiveX control, you'll have a problem. There are a couple of solutions to this—have two copies of the graphic file (one built in and one not) or mark the graphic file as excluded from the project—but they still require you to ship a bunch of graphic files with your application. Note that a solution proposed by Microsoft—copy the graphic file out from the EXE to a temporary file on disk, then use LOADPICTURE() on that temporary file—doesn't work because VFP stores the full name of the file *as it existed on your machine* in the EXE. Thus, the command COPY MYICON.ICO TO TEMP.ICO will fail on someone else's machine because VFP passes the original path (such as F:\GRAPHICS\MYICON.ICO) to the Windows API function that copies a file, and it's unlikely the user will have this path on their system.

My preference is to use a table with all the graphic files used in an application stored in memo fields, and have this table built into the EXE so it isn't a separate file to ship (or accidentally get deleted). I created a routine called LOADIMAGE that accepts the name of a graphic file and optionally the name of a table containing that file, and it returns an object for that image. If a table name is specified, or if a table called _GRAPHICS.DBF can be found, this routine looks for the name of the image in the table, and if found, copies the contents of the GRAPHIC memo field to a temporary file. If a table isn’t used, this routine then looks for the specified graphic file in various places: the directory specified with the graphic file (if there is one), the current directory, the VFP path, the same directory as the application, or a GRAPHICS subdirectory of the application directory. If the graphic file can be found (or was copied out from the table), it’s loaded using LOADPICTURE() (and the temporary file is deleted). Otherwise, the user will get an error that the graphic file couldn’t be found.

```
lparameters tcFile, ;
  tcTable
local lcTable, ;
  lcAlias, ;
  llUsed, ;
  lnSelect, ;
  lcDirectory, ;
  llDelete, ;
  laFiles[1], ;
  lcFile, ;
  loPicture

* Ensure if tcTable is specified that it's open or
* exists.

assert pcount() = 1 or empty(tcTable) or ;
  (vartype(tcTable) = 'C' and (used(tcTable) or ;
  file(tcTable))) ;
  message 'LoadImage: improper table specified'
do case

* If a table name was specified, use it.

  case vartype(tcTable) = 'C'
    lcTable = tcTable

* If a file called _GRAPHICS.DBF can be found, use it.

  case file('_GRAPHICS.DBF')
    lcTable = '_GRAPHICS.DBF'

* We don't have a table, so we'll look for a file on
* disk.

  otherwise
    lcTable = ''
endcase

* Figure out what directory we're in.

lcDirectory = GetAppDirectory()
llDelete    = .F.
do case

* If we have a table to look in, open it if necessary
* and try to find the image file there.

  case not empty(lcTable)
    lcAlias  = juststem(lcTable)
    llUsed   = used(lcAlias)
    lnSelect = select()
    if llUsed
      select (lcAlias)
    else
      select 0
      use (lcTable) again shared alias (lcAlias)
    endif llUsed
    lcFile = upper(justfname(trim(tcFile)))
    do case

* We can't open the table.
```

```
      case not used(lcAlias)
        lcFile = ''

* We can't find the specified image.

      case not seek(lcFile, lcAlias, 'NAME')
        error 1, tcFile
        lcFile = ''

* We did find it, so copy it to a temporary file.

      otherwise
        lcFile = lcDirectory + sys(2015) + '.' + ;
          justext(lcFile)
        copy memo GRAPHIC to (lcFile)
        llDelete = .T.
    endcase
    select (lnSelect)
    if not llUsed and used(lcAlias)
      use in (lcAlias)
    endif not llUsed ...

* If the file exists in the specified path, we'll use it.

  case adir(laFiles, tcFile) = 1
    lcFile = tcFile

* If the file exists in the application directory, we'll
* use it.

  case adir(laFiles, lcDirectory + tcFile) = 1
    lcFile = lcDirectory + tcFile

* If the file exists in a GRAPHICS subdirectory of the
* application directory, we'll use it.

  case adir(laFiles, lcDirectory + 'Graphics\' + ;
    tcFile) = 1
    lcFile = lcDirectory + 'Graphics\' + tcFile

* We can't find the file.

  otherwise
    lcFile = ''
endcase

* If we can't find the file, raise an error. Otherwise,
* load it.

if empty(lcFile)
  error 1, tcFile
  loPicture = .NULL.
else
  loPicture = loadpicture(lcFile)
endif empty(lcFile)

* If we copied the file to disk, erase it now.

if llDelete
  erase (lcFile)
endif llDelete
return loPicture
```

If you wish to use a table containing the graphic files, create it with two columns: NAME C(40) (or some reasonable width) and GRAPHIC M. Create a tag called NAME on UPPER(NAME). To load an image into the table, add a record to the table, specify a value for the NAME field, and use APPEND MEMO GRAPHIC FROM <name of the graphic file> to load the graphic file into the memo field.

LOADIMAGE.PRG makes it easy to load images, since they can exist on disk in one of several places or can be loaded into a table built into the EXE (either explicitly named in calls to this function or named

_GRAPHICS.DBF). This versatility means I can use it in a variety of ways any time I need to load an image.

Conclusion

Even though we're in an object-oriented world, PRGs still have their place. The eleven routines presented here represent almost all of the functions I use in applications and tools today.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. 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). He can be reached at dhennig@stonefield.com.