Manage Your Applications

Doug Hennig

This month's article presents a simple yet useful tool, and discusses several reusable techniques used in this tool.

If you're like me, your hard drive (or your server's hard drive) is littered with project directories: finished applications, in-progress client work, R & D stuff, etc. Managing and switching between all these directories can be a drag ("was that project in T:\Apps\Clients\ACME Rentals\SalesApp or in F:\Projects\Current\SalesApp?").

Recently, I had to do some remedial work on some FoxPro 2.6 applications (I bet we've all been doing some of that in the last year or so <g>), and was reacquainted with an old friend: an application manager I'd written years ago and forgotten about. It installed in the FoxPro system menu, and was a single keystroke away. This application manager was very simple: it just listed, by a descriptive name, each application I'd registered with it, and had functions to select an application, add or edit information about an application, or remove an application from the list. When you selected an application, the application manager made that application's directory the current one and automatically opened the project file. Switching between applications was a piece of cake: Ctrl-L (to invoke the application manager), End (to highlight the last application in the list, for example), and Enter (to make that application's directory the current one and open its project file).

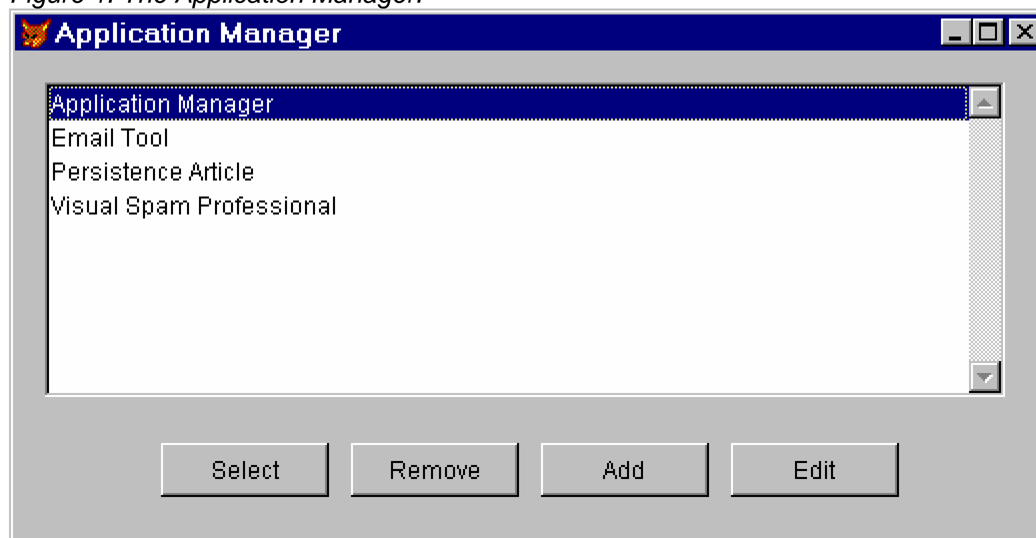
(Other developers take similar approaches that are implemented a little differently. For example, FoxTalk Editor Whil Hentzen adds a new pad to the end of the VFP system menu that lists all of his projects as items in the menu).

Once I started using my FoxPro 2.6 application manager again, I just had to use something similar in VFP. So, this month's article is about my shiny new application manager. "That sounds like kind of a simple topic", you may be thinking. Perhaps, but it's a useful tool and we'll encounter some interesting and useful techniques along the way.

The Application Manager Table and Form

Figure 1 shows the application manager form when it's running. It has a list of applications registered with the manager and buttons to select, add, edit, and remove an application. Double-clicking or pressing Enter in the list acts like clicking on the Select button.

Figure 1. The Application Manager.



Applications are registered with the manager by adding records to APPMGR.DBF. This simple table has two fields: NAME, the descriptive name of the application, and PROJECT, the path and name of the project file for the application.

APPMGR.SCX is the application manager form. Its Load method checks for the existence of APPMGR.DBF, and creates it if necessary before opening it.

```
local lcDirectory
dodefault()
lcDirectory = substr(sys(16), at(' ', sys(16), 2) + 1)
lcDirectory = addbs(justpath(lcDirectory))
if not file(lcDirectory + 'APPMGR.DBF')
  create table (lcDirectory + 'APPMGR') (NAME C(60), ;
    PROJECT C(128))
  index on upper(NAME) tag APPMGR
endif not file(lcDirectory + 'APPMGR.DBF')
use (lcDirectory + 'APPMGR') order APPMGR again shared
```

How do we know where to look for APPMGR.DBF? We expect it's in the same directory as APPMGR.APP, so we can use SYS(16) to give us the name of the currently executing method and the file (including path) it's in. In this case, we'll get "PROCEDURE FRMAPPMGR.LOAD <your path>APPMGR.SCT". So, to get the path, we can't just use JUSTPATH(), since that would give us "PROCEDURE FRMAPPMGR.LOAD <your path>". Instead, we'll use JUSTPATH() on everything after the second space ("<your path>APPMGR.SCT"), and use ADDBS() to ensure there's a trailing backslash.

The Init method calls the custom GetApps method, which does a SQL SELECT from APPMGR.DBF into the aApps array property of the form. The lstApps listbox on the form has this array as its RowSource so it lists the application descriptions.

The DblClick method of the listbox and Click method of the Select button both call Thisform.SelectApp. This method tries to change the current directory to the one containing the project file for the selected application, and then open the project. It tries to handle potential problems, such as the directory not existing or the project file not found. For example, if the directory exists but the project file doesn't, it looks in the directory to see if there are any project files. If there are none, it gives an error message. If there's exactly one, it opens that one (on the assumption that the project file was renamed after it was registered with the application manager). If there's more than one, an Open File dialog is presented so you can pick the project file to open.

The Click method of the Remove button calls Thisform.RemoveApp, which simply deletes the record for the selected application and calls GetApps again to refresh the array and listbox.

The Click methods of the Add and Edit buttons both call Thisform.EditApp, Add passing it .T. to indicate that we're adding rather than editing. EditApp runs a form called EDITAPP.SCX to get the description of the application and its project file. However, notice that we'll need three return values from EDITAPP.SCX: the description, the project, and whether the user chose OK or Cancel. Since there can only be one return value, how do we handle this? My preferred mechanism is to use a parameter object, one whose sole purpose is to be moved between methods with a "package" of parameters in its properties. These parameters are two-way; either the caller or the method being called can examine or change them.

Because both the caller and the method being called must know the names of the properties in the parameter object, you could create a class for the parameter object for this use. However, if you use this technique a lot, you'd quickly find yourself creating a lot of these classes. Being a lazy, er, efficient programmer, I prefer to use a single generic class (SFParameter in SFCTRLS.VCX) for this. This class has a This_Access method that automatically fires whenever anything tries to access a property of the object. The code for that method is as follows:

```
lparameters tcMember
if not pemstatus(This, tcMember, 5)
  This.AddProperty(tcMember)
endif not pemstatus(This, tcMember, 5)
return This
```

This code does something pretty cool: if the property being accessed doesn't exist, it's created. This means that even though this object doesn't have, for example, a cMessage property, we can do the following:

```
loProperties = createobject('SFParameter')
loProperties.cMessage = 'Howdy'
```

In other words, this class exposes a variable interface; it looks like anything anyone wants it to!

OK, back to the EditApp method. The first task is to create an SFParameter object and fill its (created on the fly) properties with either the description and project file name for the selected application or empty values if we're adding an application. Next, if we're editing an application, we position the APPMGR table to the record for the application; this allows EDITAPP.SCX to ensure we don't enter a duplicate record. Then we call EDITAPP.SCX, passing it the parameter object. Upon return, we either do nothing if the user chose Cancel, or create or update the record in APPMGR and call GetApps to refresh the application array and listbox if the user chose OK. In the case of adding an application, we also ensure the new application is highlighted in the listbox.

```
lparameters tlAdd
local lcAppName, ;
  lcProject, ;
  loProperties

* Create a properties object and fill it with either the
* name and project for the selected app or blanks for a
* new app.

lcAppName    = iif(tlAdd, '', ;
  trim(Thisform.aApps[Thisform.lstApps.ListIndex, 1]))
lcProject    = iif(tlAdd, '', ;
  trim(Thisform.aApps[Thisform.lstApps.ListIndex, 2]))
loProperties = MakeObject('SFParameter', 'SFCtrls.vcx')
loProperties.cAppName = lcAppName
loProperties.cProject = lcProject
loProperties.lNew     = tlAdd
loProperties.lSaved   = .F.

* If we're editing an app, position the APPMGR table to
* the record for the selected app.

if not tlAdd
  seek upper(loProperties.cAppName)
endif not tlAdd

* Run the EditApp form. If the user saved their changes,
* either add or edit the app information.

do form EditApp with loProperties
do case
  case not loProperties.lSaved
  case tlAdd
    insert into APPMGR values (loProperties.cAppName, ;
      loProperties.cProject)
    This.GetApps()
    lnApp = ascan(This.aApps, loProperties.cAppName)
    if lnApp > 0
      This.lstApps.ListIndex = asubscript(This.aApps, ;
        lnApp, 1)
    endif lnApp > 0
  otherwise
    replace NAME with loProperties.cAppName, ;
      PROJECT with loProperties.cProject
    This.GetApps()
endcase
```

EDITAPP.SCX is a simple form: it just has a couple of textboxes so you can enter the application description and project filename. Its Init method accepts the parameter object passed in and stores a reference to it in This.oProperties. The Click method of the OK button (which is an instance of SFOKButton in SFBUTTON.VCX so it has a standard appearance) puts the description and project filename into the cAppName and cProject properties of the parameter object, sets its lSaved property to .T. so the EditApp method will know the user chose OK, and then releases the form. There's no code in the Cancel button, since it's an instance of SFCancelButton (also in SFBUTTON.VCX), which has Thisform.Release() in its Click method.

One useful thing in EDITAPP.SCX is a button that displays an Open File dialog (using GETFILE()) so the user can select a file. This button is an instance of SFGetFile (in SFBUTTON.VCX). Rather than having to remember all the parameters for GETFILE() and code what to do after the user selects a file, you simply drop this class on a form and set a few properties. The cExtensions, cText, cOpenButton, nButtonType, and cCaption properties represent the appropriate parameters for GETFILE(), cResult contains the name of the location to put the filename the user selected into, and cAfterDone contains the name of a method or function to execute after the file dialog is closed. In the case of the instance used in EDITAPP.SCX, cExtensions contains "PJX" (since we only want to select project files), cResult contains "Thisform.txtProject.Value" (since that's where the filename should be placed), and cAfterDone contains "Thisform.RefreshForm()" (which, while refreshing the form, will enable the OK button if the application name was also entered).

Main Program

APPMGR.PRG is the main program for the application manager APP file. It consists of a main routine and a few subroutines. The purpose of the main routine is two-fold: to install the application manager in the VFP system menu, and to run the application manager (I prefer to have an APP be self-contained rather than having the installation and execution code in separate places). Which of these tasks is performed depends on how the program is called. If no parameters are passed, or if "INSTALL" is passed (in other words, you used DO APPMGR.APP or DO APPMGR.APP WITH "INSTALL"), this routine will call the InstallAppMgr subroutine to install the application manager in the menu. If "RUN" is passed (which is what happens when you choose "Application Manager" from the menu, as we shall soon see), the RunAppMgr subroutine is called to run APPMGR.SCX. Here's the code for the main routine:

```
lparameters tcAction, ;
  tcPopup
local lcCurrTalk, ;
  lcAction

* Save the current TALK setting and set it off.

if set('TALK') = 'ON'
  set talk off
  lcCurrTalk = 'ON'
else
  lcCurrTalk = 'OFF'
endif set('TALK') = 'ON'

* Handle the action parameter: if it isn't passed, we'll
* assume "INSTALL".

lcAction = iif(vartype(tcAction) <> 'C' or ;
  empty(tcAction), 'INSTALL', upper(tcAction))
do case

* If this is an "install" call, do the installation.

  case lcAction = 'INSTALL'
    do InstallAppMgr with tcPopup

* Run the application manager.

  case lcAction = 'RUN'
    do RunAppMgr
endcase
```

```
* Cleanup and return.

if lcCurrTalk = 'ON'
  set talk on
endif lcCurrTalk = 'ON'
return
```

Notice that this routine can accept two parameters: the "action" ("RUN", "INSTALL", or nothing, which means "INSTALL") and the name of the menu popup into which to install the application manager. As we'll see in a moment, the default popup is _MSM_TOOLS, the popup for the Tools menu, but you can pass the name of another popup (note it must be the popup name, not the prompt that appears in the menu; see "System Menu Names" in the VFP help file for a list of the names) if you want it installed under a different menu item. There's another good reason for passing the popup, even _MSM_TOOLS, as I'll discuss shortly.

The RunAppMgr subroutine, called when "RUN" is passed to APPMGR.APP, is very simple: it just consists of DO FORM APPMGR. I could've placed this code directly into the main routine rather than calling a subroutine, but this approach is more modular.

The main routine calls InstallAppMgr to add the application manager to the system menu. This routine defines a couple of constants that specify the prompt to appear in the menu (including the hot key for it) and the popup to install in if one isn't specified (_MSM_TOOLS). The first task of this routine is to determine what directory this program is running in. We need it so we can hard-code the path to find APPMGR.APP into the menu; after all, we don't want to have the directory in the VFP path and it certainly won't be in the current directory. This uses the same technique involving SYS(16) I discussed earlier.

```
lparameters tcPopup
local lcDirectory, ;
  lnBar, ;
  lcBar, ;
  llPopup, ;
  lcPad, ;
  lnI
#define ccMENU_BAR_PROMPT   'Appl\<ication Manager'
#define ccMENU_PAD          '_MSM_TOOLS'
lcDirectory = substr(sys(16), at(' ', sys(16), 2) + 1)
lcDirectory = addbs(justpath(lcDirectory))
```

The next thing to do is to see if an item for the application manager has already been added to the menu. This might happen if, for example, you run this program more than once. We wouldn't want to have two items for the application manager in the menu, so if we find one, we'll just reuse it. We'll use CNTBAR() to get the number of bars in the popup, then step backwards through them (since the application manager item is added at the bottom of the menu, this is faster than starting at the top and working down). GETBAR() gives us the number of a bar in a menu based on its position, and PRMBAR() gives us the prompt of the specified bar. Since PRMBAR() gives us the prompt without any hot key characters (\<), we'll compare the prompt against the desired prompt stripped of these characters.

```
lnBar   = 0
lcBar   = strtran(ccMENU_BAR_PROMPT, '\<')
llPopup = vartype(tcPopup) = 'C' and not empty(tcPopup)
lcPad   = iif(llPopup, tcPopup, ccMENU_PAD)
for lnI = cntbar(lcPad) to 1 step -1
  if prmbar(lcPad, getbar(lcPad, lnI)) = lcBar
    lnBar = lnI
    exit
  endif prmbar(lcPad ...
next lnI
```

If we didn't find the application manager in the menu, we'll add a separator line and a new bar at the end of the menu. We'll then define what to do when the item is chosen. Notice that the directory is hard-coded into the menu using macro expansion; this is necessary because we can't use the lcDirectory variable when the menu item is chosen since that variable won't exist. We also use LOCFILE() if for some reason

VFP can't find APPMGR.APP where it was originally located. APPMGR.APP will be passed "RUN" so the main routine in APPMGR.PRG knows to run the application manager rather than install it.

```
if lnBar = 0
  lnBar = cntbar(lcPad) + 1
  define bar lnBar     of &lcPad after _mlast prompt '\-'
  define bar lnBar + 1 of &lcPad after _mlast ;
    prompt ccMENU_BAR_PROMPT
else
  lnBar = lnBar - 1
endif lnBar = 0
on selection bar lnBar + 1 of &lcPad ;
  do locfile('&lcDirectory.APPMGR.APP', 'APP', ;
  'Locate APPMGR.APP') with 'Run'
if not llPopup
  set sysmenu save
endif not llPopup
return
```

Notice something interesting at the end of this code: if no popup name is specified, we use SET SYSMENU SAVE to ensure that SET SYSMENU TO DEFAULT doesn't wipe out the application manager item in the menu. However, we don't do this if a popup name is passed as a parameter. The reason for doing this is something Toni Feltman of F1 Technologies pointed out to me: if you use more than one SET SYSMENU SAVE, your chance of getting a "menu manager inconsistency error" (which terminates VFP faster than you can say "GPF") at some point in your VFP session skyrockets. So, if you want to add several items to your VFP system menu, it's best to leave the SET SYSMENU SAVE command until after you've added them all. Although I don't like to mention products I've developed in my articles, a feature I recently added to Stonefield Database Toolkit (SDT) is pertinent here: SDT.APP (which adds two items to the Tools menu) accepts the "action" and popup name parameters using almost the same code as APPMGR.PRG (in fact, I shoplifted the code for APPMGR.PRG from SDT <s>). So, if you want to add both the application manager and SDT to your system menu, here's the best way to do that:

```
do APPMGR.APP with 'Install', '_MSM_TOOLS'
do SDT.APP    with 'Install', '_MSM_TOOLS'
set sysmenu save
```

This defers SET SYSMENU SAVE until all the custom items have been added to the menu. An example of where you might do this is in some program automatically executed when you start VFP (for example, I have _STARTUP = "VFPSTART.PRG" in my CONFIG.FPW and have code I want executed every time VFP starts in this PRG).

Conclusion

Although this is a simple tool, I find myself using it at least a dozen times a day (am I the only one who's usually at about four levels of interrupt deep at any one time <g>?), so it saves me lots of time trying to remember the exact name of a directory, typing it correctly, and then remembering (and typing correctly) the name of the project file to open. In addition to a useful tool, I hope you also picked up a few useful techniques.

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.