A File Open Dialog

Doug Hennig

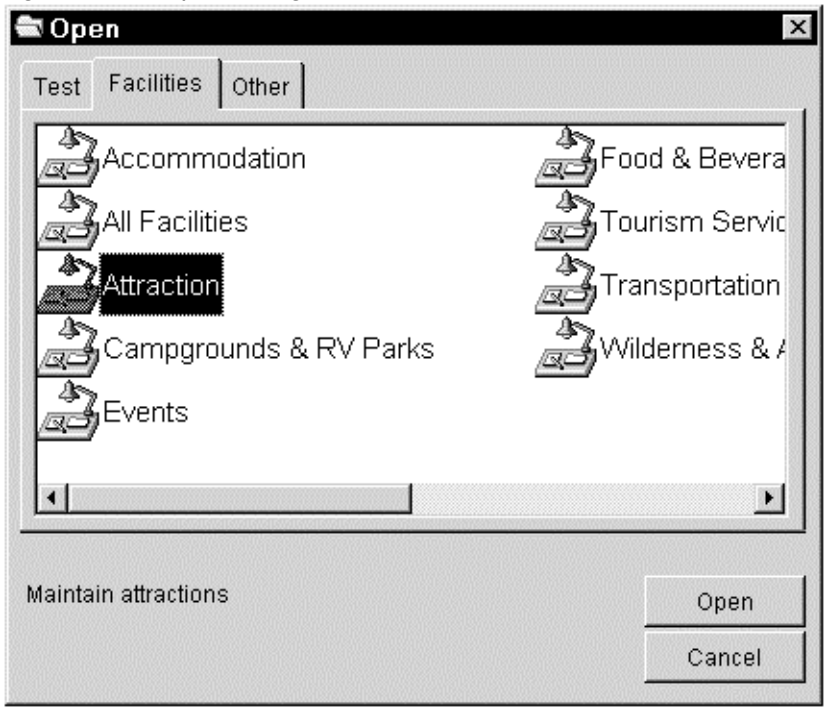
A File Open dialog is a better approach than having a long list of forms appear in the File menu. This article presents a reusable File Open dialog that supports grouping forms into different pages in a pageframe and uses ListView controls for a visually appealing display.

One of the issues sure to cause a flurry of messages on CompuServe these days is: how do you present a menu of which forms a user can run? In the past, our applications had a File pad in the application menu, with bars under this pad listing each of the data-entry forms a user could run (other developers put their forms under different pads, but the idea was the same). The problem with this approach is a complex application with many forms can quickly create a long ugly menu. Also, Windows users are now used to seeing certain standard functions appearing in the File pad (New, Open, Close, etc.) rather than specific form names. So, we've gone with a File Open dialog called from the Open bar in the File menu. This dialog displays a list of forms the user can choose from. While this is generally a great approach, in the case of a complex application with a lot of forms, it simply shifts the list of forms from a long menu to a long scrolling list in a dialog.

One of the things I've always liked about Microsoft Office is the New Office Document dialog. It presents a tabbed display of templates you can base a new document on. Different types of templates are shown on different pages, with each page showing icons representing each template type. This article creates a VFP 5 File Open dialog that reproduces this user interface for database applications.

Figure 1 shows a screen shot from the File Open dialog of the sample application accompanying this article. Notice there are several pages, one for each "group" of forms. As user clicks on a item in the dialog, a edit box shows a more detailed description of the item. Double-clicking on an item or selecting an item and choosing the Open button causes that form to be run.

Figure 1. File Open dialog.



Modules

The File Open dialog displays what I call "modules". A module is anything the application can run: a form, a PRG, an external utility, etc. The modules used by the application are defined in a MODULES table,

which is the source of information used by the File Open dialog. Table 1 shows the structure of this table and Table 2 shows some sample records (some fields are omitted for brevity)

Table 1. Structure of MODULES.DBF.

Field	Type	Size	Description
NAME	Character	40	The name of the module as the user sees it
MODULE	Character	40	The name of the module as the application knows it (e.g. form name)
DESCRIP	Memo	4	A full description of the module
GROUP	Character	40	The group the module belongs to
HOWTOCALL	Character	45	A string that when macro expanded will execute the module
SECURITY	Numeric	1	The type of security used for this module
INOPENFORM	Logical	1	.T. if this module should appear in the File Open dialog
ORDER	Numeric	3	The order the group should appear in the dialog
IMAGEKEY	Character	20	The name of the ICO file used as the image for this module

Table 2. Sample records for MODULES.DBF.

NAME	GROUP	HOWTOCALL	INFORM	ORDER	IMAGEKEY
Reindex		oMeta.oSDTMgr.Reindex()	.F.	0	
Security		oApp.OpenForm('SFSecur')	.F.	0	
Accommodation	Facilities	oApp.OpenForm('Facility', 'cSectorId = "1"')	.T.	2	desktop.ico
All Facilities	Facilities	oApp.OpenForm('Facility')	.T.	2	desktop.ico
Clients	Other	oApp.OpenForm('Client')	.T.	3	drive.ico
Test Program	Test	do TEST	.T.	1	audio.ico

By macro expanding the contents of the HOWTOCALL field to call a particular module, the module can be called in a more complex manner than simply using DO. For example, many of the records in Table 2 call a form by passing parameters to the OpenForm() method of an oApp object (which would likely be an application manager object but isn't part of the simple sample application). Other modules are called differently.

Notice this table is not completely normalized: GROUP contains the name of the group as the user sees it, and ORDER is the order the group's page appears in the pageframe (this allows you to order the pages in some way other than alphabetical). Records in the same group will have the same values in GROUP and ORDER. This results in redundant information in the table, but we felt this was less important than simplifying the design and avoiding having a table just to store groups.

In the sample application that accompanies this article, SECURITY only contains two values: 0 for no security (all users can access the module) or 1 for security (only some users can access it). A real application would likely make more use of this field, such as indicating the type of security used. For example, we use 1 for "yes/no" security (meaning the user can either access the module or not) and 2 for modules that support "read/write" security (some users can add, edit, or delete records while others can only view them). The CheckAccess() method of a "user manager" class (SFUser, defined in SFAPPLIC.VCX) is used to determine if the current user can access a particular module. Since security issues go beyond the scope of this article, this method simply returns .T. for every module. If you want to see how the File Open dialog supports security, though, edit this method to return .F., rebuild the sample application, and run it. Notice that only certain modules now appear in the dialog.

Not all modules in the application appear in the File Open dialog. For example, utility functions like Reindex and Import might appear in the application menu but not in the File Open dialog. Since we use MODULES for security information about modules in addition to the description of modules for the File Open dialog, the INOPENFORM field allows us to define if a module will appear in the File Open dialog or not. If INOPENFORM is .F., this module is in the MODULES table for security needs only. A real application might use these records to decide if the Skip For clause of a menu item evaluates to .T. for a particular user.

Design Details

Let's look at the code for this reusable tool. SFOPEN.SCX is the File Open dialog called when the user chooses Open from the File menu. It's based on SFForm (a subclass of Form defined in SFCTRLS.VCX, our class library of subclassed VFP base classes) and has a private datasession (so opening the MODULES

doesn't interfere with any other data session) and an appropriate icon and caption. This form is very simple: it just has an SFFileOpen object on it and no special properties or methods. All of the work is done in the SFFileOpen object, which we'll look at shortly.

SFOPEN.VCX is a class library containing two classes that provide the functionality for the File Open dialog: SFFileOpenListView and SFFileOpen. SFFileOpenListView is an OLEControl subclassed from the ListView ActiveX control. If you haven't worked with a ListView control before, it's very similar to the TreeView control I discussed in the June 1997 issue of FoxTalk. We'll look at its properties and methods in more detail when we discuss the SFFileOpen class, but for now, only a few properties are set at design time (the rest are set at runtime):

- View: controls whether the ListView displays large icons, small icons, a list, or a report. The right pane of the Windows Explorer is a ListView control and the Explorer includes buttons in its toolbar that switch between these four views. We'll set View to 2 (List) in this subclass.
- LabelEdit: the default is 0, which allows the user to edit the label of an item in the ListView simply by clicking on it and typing a new label. Because this behavior would be confusing, we'll set it to 1 (Manual).
- HideSelection: the default is .T., which means when another object gets focus, the selected item in the ListView is no longer highlighted. I prefer to set this property to .F. so the selected item is always highlighted.
- Height: 250
- Width: 360

SFFileOpenListView has code in two methods. DblClick() has the following code:

```
This.Parent.Parent.Parent.DoModule()
```

This calls the DoModule() method of the SFFileOpen control this object will sit on to run the module the user double-clicked. ItemClick(), which is called when the user clicks on an item, has the following code:

```
This.Parent.Parent.Parent.ItemClick(Item)
```

This calls the ItemClick() method of the SFFileOpen control, passing it an object reference to the item the user clicked on. We'll take a look at the methods being called in a moment.

The other class in SFOPEN.VCX, SFFileOpen, is the object in the File Open dialog that provides the majority of the functionality of this form. SFFileOpen is based on SFContainer (a subclass of Container defined in SFCTRLS.VCX). It has two custom properties: cSelectedModule, which holds the name of the selected module, and cHowToCall, a protected property containing the command string to call the selected module. SFOpen has several member objects, shown in Table 3.

Table 3. SFOpen member objects.

Name	Based On	Properties	Methods
pgfOpen	SFPageFrame	Height: 250 Width: 410 Top: 0 Left: 0	
oImageList	OLEControl (ImageList)	Top: 0 Left: 310	
oSmallImageList	OLEControl (ImageList)	Top: 0 Left: 360	
edtDescription	SFReadOnlyEditBoxFlat (SFRO.VCX)	Height: 49 Left: 0 Top: 270 Width: 310	Refresh: This.Value = MODULES.DESCRIP
cmdOpen	SFCommandButton	Caption: Open	Click:

		Left: 325 Top: 270	This.Parent.DoModule()
cmdCancel	SFCancelButton (SFBUTTON.VCX)	Left: 325 Top: 299	

The Init() method of SFFileOpen opens the MODULES table and calls LoadList() to do the majority of the setup work:

```
dodefault()
use MODULES order ORDER again shared
return This.LoadList()
```

LoadList() is a protected method that loads the modules to display to the user. Here's the code:

```
local lcKey, ;
  laGroups[1], ;
  laImages[1], ;
  lcImage, ;
  lcImageKey, ;
  lcImageFile, ;
  loPicture, ;
  lnImages1, ;
  lnImages2, ;
  llClassLib, ;
  lnHeight, ;
  lnWidth, ;
  llTabs, ;
  lnI, ;
  lnPage, ;
  loItem
with This.pgfOpen

* Get the current key for MODULES.

  select MODULES
  lcKey = key()

* Grab those records from MODULES that should appear
* in the Open dialog and this user has rights to.

  select * ;
    from MODULES ;
    where INOPENFORM and ;
      not empty(GROUP) and ;
      (SECURITY = 0 or ;
        not empty(oUser.CheckAccess(MODULE))) ;
    into cursor LOADLIST ;
    order by ORDER, ;
      NAME

* If there aren't any modules this user has rights to,
* give an error and exit.

  if _tally = 0
    messagebox('You do not have access to any  ' + ;
      modules in this application.', 16, ;
      _screen.Caption)
    return .F.
  endif _tally = 0

* Load the images used by the modules.

  select IMAGEKEY from LOADLIST group by 1 ;
    into array laImages
  for each lcImage in laImages
    lcImageKey  = trim(lcImage)
    lcImageFile = fullpath(lcImageKey)
```

```
    if file(lcImageFile)
      loPicture = LoadPicture(lcImageFile)
      if type('loPicture') = 'O' and ;
        not isnull(loPicture)
        This.oImageList.ListImages.Add(, lcImageKey, ;
          loPicture)
        This.oSmallImageList.ListImages.Add(, ;
          lcImageKey, loPicture)
      endif type('loPicture') = 'O' ...
    endif file(lcImageFile)
  next lcImage
  lnImages1 = This.oImageList.ListImages.Count
  lnImages2 = This.oSmallImageList.ListImages.Count

* We'll use one page for each group of modules, so
* figure out how many pages we need and what their
* captions should be. If there's only one page, hide
* the tabs.

  select GROUP, ;
      ORDER ;
    from LOADLIST ;
    into array laGroups ;
    group by 2, 1
  .PageCount = alen(laGroups, 1)
  if .PageCount = 1
    .Tabs = .F.
  endif .PageCount = 1

* If necessary, open the class library we're in
* because we need another class there.

  llClassLib = This.ClassLibrary $ set('CLASSLIB')
  if not llClassLib
    set classlib to (This.ClassLibrary) additive
  endif not llClassLib

* Add a ListView control to each page and set its
* properties.

  lnHeight = .Height
  lnWidth  = .Width
  llTabs   = .Tabs
  for lnI = 1 to .PageCount
    if llTabs
      .Pages[lnI].Caption = trim(laGroups[lnI, 1])
    endif llTabs
    .Pages[lnI].AddObject('oList', ;
      'SFFileOpenListView')
    with .Pages[lnI].oList
      .Top        = 5
      .Left       = 5
      .Width      = lnWidth  - 15
      .Height     = lnHeight - iif(llTabs, 40, 20)
      .Visible    = .T.

* Set the Icons and SmallIcons properties if we have
* any images loaded.

      if lnImages1 > 0
        .Icons = This.oImageList.Object
      endif lnImages1 > 0
      if lnImages2 > 0
        .SmallIcons = This.oSmallImageList.Object
      endif lnImages2 > 0

* Create some columns for the ListView in case we
* want to display in Report view.

      .ColumnHeaders.Add(, 'Name', ;
        'Name', int(.Width/2))
```

```
      .ColumnHeaders.Add(, 'Description', ;
        'Description', int(.Width/2))
    endwith
  next lnI

* Close the class library if we opened it.

  if not llClassLib
    release classlib (This.ClassLibrary)
  endif not llClassLib

* Add each module to the ListView control for the
* group the module belongs to.

  scan
    lnPage = asubscript(laGroups, ascan(laGroups, ;
      GROUP), 1)
    do case
      case lnImages1 = 0 and lnImages2 = 0
        loItem = .Pages[lnPage].oList.ListItems.Add(, ;
          sys(2015), trim(NAME))
      case lnImages1 = 0
        loItem = .Pages[lnPage].oList.ListItems.Add(, ;
          sys(2015), trim(NAME), , trim(IMAGEKEY))
      case lnImages2 = 0
        loItem = .Pages[lnPage].oList.ListItems.Add(, ;
          sys(2015), trim(NAME), trim(IMAGEKEY))
      otherwise
        loItem = .Pages[lnPage].oList.ListItems.Add(, ;
          sys(2015), trim(NAME), trim(IMAGEKEY), ;
          trim(IMAGEKEY))
    endcase
    loItem.SubItems(1) = trim(DESCRIP)
    loItem.Tag = evaluate(lcKey)
  endscan
  select MODULES

* Select the first node.

  .Pages[1].oList.SelectedItem = .Pages[1].oList.ListItems[1]
  This.ItemClick(.Pages[1].oList.SelectedItem)
endwith
```

This is a complex method, so we'll discuss it a little further. LoadList() gets a list of which modules the user has rights to using a SQL SELECT from the MODULES table, including a call to the CheckAccess() method of an oUser object (instantiated from the SFUser class, which is defined in SFAPPLIC.VCX as described earlier). LoadList() then creates one page in the pgfOpen pageframe per unique group of modules the user will see. Each page contains an SFFileOpenListView object to display the icons for the modules. Items are added to the appropriate ListView control, depending on which group the module belongs to. After all the modules have been added, the first module in the first page is selected.

LoadList() has some interesting features:

- Images for the items in the ListView controls must come from ImageList controls (one for the small icon and one for the large icon for each item). We use the new VFP 5 LoadPicture() function to load an image from a icon file (the IMAGEKEY field in the MODULES table specifies which icon file to use for each module) into the ImageList controls at runtime. This is one of the keys to making this tool reusable; without LoadPicture(), we'd have to pre-load the ImageList controls at design time with the images we want, which would require creating a customized version of these classes for each application. (One caveat: LoadPicture() appears to have a bug: it cannot load a file built into an APP or EXE, so you'll need to provide the graphics as separate files). Although the same icon file is used for both the small and large icon images, you could create separate fields in MODULES to contain small and large icon file names and modify this code to use these separate fields.

- If the user only has access to one group of modules, the tabs of the pageframe are hidden so only a single page is seen.
- As each ListView control is added to each page, several properties are set. Icons and SmallIcons must contain an object reference to an ImageList control, so these properties are set to the two ImageList controls. Columns are added to the ListView control using the Add() method of the ColumnHeaders collection that's part of the control. This allows you to change the View property to 3 (Report) if desired so the user can see the name and description of the module side-by-side in the control.
- Items are added to the ListView controls using the Add() method of the ListItems collection that's part of the control. This method needs a unique string for a key value (SYS(2015) is used), the text to display, and the image keys for the large and small icons for the item. In addition, each item has a "subitem" (used for columns beyond the first one in the Report view of the ListView control) which contains the description of the module.
- If you want to see how ColumnHeaders and Subitems are used in the ListView control, change the SFFileOpenListView class so the View property is set to 3 (Report) by right-clicking on the control, choosing the ListView property sheet from the context menu, and choosing this value in the View combo box.. When you rebuild and run the sample application, you'll notice a two-column list with the name of each module in the first column and the description in the second. You can resize these columns as you wish.
- The Tag property of each item is set to the key value for the record in the MODULES table. This way, we can later use the Tag property for the selected item in the ListView to find the matching record in MODULES.

The ItemClick() method of SFFileOpen is called when the user clicks on an item in any of the ListView objects (it's called from the ItemClick() method of SFFileOpenListView). It's also called from the LoadList() method to ensure the first module on the first page is selected. ItemClick() calls FindModule() to find the record in MODULES for the selected item and then refresh the form.

```
lparameters toItem
This.FindModule(toItem.Tag, .T.)
Thisform.Refresh()
```

FindModule() is a protected method that finds the specified module in the MODULES table and sets the cSelectedModule and cHowToCall properties from fields in the table. In addition to the name of the module, FindModule() accepts a flag parameter indicating which tag of MODULES should be used for the SEEK (.T. means a complete key value was passed to the method, so it should use the ORDER tag; otherwise, it uses the MODULE tag since only the name of the module was passed).

```
lparameters tcModule, ;
  tlCompleteKey
local luValue
with This
  luValue = evaluate(key(tagno(iif(tlCompleteKey, ;
    'ORDER', 'MODULE'))))
  = seek(padr(upper(tcModule), len(luValue)), ;
    'MODULES', iif(tlCompleteKey, 'ORDER', 'MODULE'))
  .cSelectedModule = MODULES.MODULE
  .cHowToCall      = MODULES.HOWTOCALL
endwith
```

DoModule() is called from the Click() method of the Open button and from the DblClick() method of the SFFileOpenListView objects. It hides the File Open dialog and then runs the selected module by macro expanding the contents of the cHowToCall property (set in FindModule() from the HOWTOCALL field in MODULES). Finally, it releases the File Open dialog since it's no longer needed.

```
local lcModule
Thisform.Hide()
lcModule = This.cHowToCall
```

```
if not empty(lcModule)
  &lcModule
endif not empty(lcModule)
Thisform.Release()
```

Error() handles errors in DoModule specially to avoid problems with syntax or other errors if HOWTOCALL isn't specified properly or if a form or program is missing. It passes any other errors to the Error() method of its super class.

```
LPARAMETERS nError, cMethod, nLine
if upper(cMethod) = 'DOMODULE'
  messagebox('This module cannot be executed.', 16, ;
    _screen.Caption)
else
  return dodefault(nError, cMethod, nLine)
endif upper(cMethod) = 'DOMODULE'
```

The sample application accompanying this article consists of a few other pieces besides SFOPEN.SCX, SFOPEN.VCX, and SFCTRLS.VCX:

- SFRO.VCX contains read-only classes that were described in the December 1996 issue of FoxTalk. One of its classes, SFReadOnlyEditBoxFlat, is used in SFFileOpen.
- SFBUTTON.VCX contains a couple of CommandButton classes: SFOKButton, an OK button, and SFCancelButton, which closes the form it's on. SFCancelButton is used in SFFileOpen.
- SFAPPLIC.VCX contains the definition of a very simple user manager class (SFUser). It's only present here to show how its CheckAccess() method could be used by SFFileOpen to prevent users from seeing modules they don't have rights to.
- SYSMAIN.PRG is the main program. It instantiates SFUser into oUser, installs the application menu (defined in SYSMAIN.MNX), and issues a READ EVENTS. It also contains EXIT_APP, a procedure called from the Exit bar of the File pad that issues a CLEAR EVENTS to end the READ EVENTS loop.
- Several ICO files (all from the VFP sample files) provide sample images for the modules.

Note that only the "Test Program" module can be run; the other items in this sample application are not included and give a "this module cannot be executed" message.

Summary

A File Open dialog makes your application look and act more like other Windows applications than having a list of forms appear in the File menu. With a reusable File Open dialog, you simply have to maintain a MODULES table specific to your application. Since this table can have other uses as well, including defining module-level security, this is a simple and easy approach to this problem. Enjoy!

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 for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. 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 user groups and regional conferences all over North America, and spoke at the 1997 Microsoft FoxPro Developers Conference. He is a Microsoft Most Valuable Professional (MVP). 75156.2326@compuserve.com or dhennig@stonefield.com.