Extending the VFP 9 IDE

Doug Hennig

One of the key themes in VFP 9 is extensibility. You can extend the VFP 9 Report Designer through report events and the reporting engine through the new ReportListener base class. You can even extend the IDE by trapping system and shortcut menu hits. This month, Doug Hennig shows you how to customize the IDE in ways you never thought possible.

VFP has long provided ways to hook into various aspects of the interactive development environment (IDE), and it gets better with each version. FoxPro 2.x allowed us to replace the functionality of certain Xbase components by changing the name of the applications pointed to by system variables, such as _GENSCRN and _GENMENU. VFP 3 gave us builders, which allow us to automate or simplify working on classes and forms. VFP 6 added project hooks; they allow us to receive events when the user does things in the Project Manager, such as adding or removing a file. IntelliSense was one the major new features in VFP 7, and the data-driven approach the VFP team used means we can customize how it works.

In VFP 9, Microsoft has blown the lid off the IDE. We can completely change the appearance and behavior of the dialogs used by the Report Designer because the Report Designer raises report events that we can react to. The new ReportListener base class allows us to receive events as a report is run, providing features such as label rotation, dynamic formatting, and custom rendered objects such as graphs. We'll be able to react to Windows events; this feature isn't available as of this writing, but will give us nearly complete control over both the VFP IDE and run-time environment.

The focus of this article, though, is hooking into system menu hits (that is, when the user selects an item from the VFP system menu bar) and system shortcut menus (those displayed by right-clicking in various places, such as the Properties window or Database Designer). We'll start by looking at how to do this, then we'll see some practical examples.

MENUHIT

To trap a system menu hit, create a script record in the IntelliSense table, with TYPE = "S" (for "script"), ABBREV = "MENUHIT," and the code to execute in the DATA memo field. (The IntelliSense table is pointed to by the _FOXCODE system variable; by default, it's FOXCODE.DBF in the HOME(7) directory.) The code should accept an object as a parameter. This object has several properties, but the important ones as far as MENUHIT is concerned are shown in **Table 1**.

Table 1. The properties of the parameter object passed to the MENUHIT script provide information about the menu hit.

Property	Description
UserTyped	The prompt of the pad the menu hit is from
MenuItem	The prompt of the bar the user selected
ValueType	A return value to VFP: blank means continue with the default behavior and "V" or "L" means prevent the default behavior (similar to using NODEFAULT in class code).

Here's a simple example (taken from SimpleMenuHit.PRG included in this month's Subscriber Downloads):

```
text to lcCode noshow
lparameters toParameter
wait window 'Pad: ' + toParameter.UserTyped + chr(13) + ;
  'Bar: ' + toParameter.MenuItem
endtext
delete from (_foxcode) where TYPE = 'S' and ;
  ABBREV = 'MENUHIT'
insert into (_foxcode) ;
    (TYPE, ABBREV, DATA) ;
  values ;
    ('S', 'MENUHIT', lcCode)
```

This simply displays the pad and bar prompts for the selected menu item and then carries on with the default behavior for that item. This is fine for testing code, but in "real" code, you may want to disable the default behavior and replace it with your own. In that case, set the ValueType property of the parameter object to "V" or "L". (In case you're wondering why there are two values, it's because these values are used for other purposes by the IntelliSense manager, and the VFP team decided to use the same mechanism for MENUHIT.)

Here's an example, taken from DisableExit.PRG, that disables the Exit function in the File menu (perhaps you could use this as an April Fools joke on a co-worker):

```
text to lcCode noshow
lparameters toParameter
if toParameter.UserTyped = 'File' and ;
  toParameter.MenuItem = 'Exit'
  toParameter.ValueType = 'V'
endif toParameter.UserTyped = 'File' ...
endtext
delete from (_foxcode) where TYPE = 'S' and ;
  ABBREV = 'MENUHIT'
insert into (_foxcode) ;
    (TYPE, ABBREV, DATA) ;
  values ;
    ('S', 'MENUHIT', lcCode)
```

While this looks correct, when you choose Exit from the File menu, VFP exits anyway. It turns out that in addition to setting ValueType to "V" or "L", some functions require that the MENUHIT code returns .T. to prevent the default behavior. Add RETURN .T. just before the ENDIF statement to prevent Exit from closing VFP.

Another MENUHIT issue: what if you're using a localized version of VFP, such as the German version? In that case, checking for "File" and "Exit" won't work because those aren't the prompts that appear in the menu. This is something you'll have to be aware of, or will have to make your target audience aware of, if you distribute a MENUHIT script to other VFP developers.

The MENUHITs just keep coming

While we're on the topic of distributing a MENUHIT script to others, what if you have a script that does something cool and so does someone else, and you want to use both of them? The problem is that there can only be one MENUHIT record (if more than one is present, VFP will use the first one in the IntelliSense table). For this reason, I think it's better to have the MENUHIT record delegate to something else rather than perform the desired IDE customization directly. The simplest way to do this is to add additional records to the IntelliSense table and have the MENUHIT record use them to perform the actual tasks. Although this is arbitrary, it seems to me that a record with TYPE = "M" would be suited to this, because "M" stands for "menu" and isn't a record type currently used in the table. For example, to handle the Exit function, you could add a record with TYPE = "M", ABBREV = "Exit", and the code to execute in DATA.

To make this work, a "standard" MENUHIT record is needed. The code for this record looks in the table for another record with TYPE = "M" and ABBREV set to prompt for the bar the user chose, and if it exists, executes the code in the DATA memo. Here's the code that handles this:

```
lparameters toParameter
local lnSelect, ;
  lcCode, ;
  llReturn
try
  lnSelect = select()
  select 0
  use (_foxcode) again shared order 1
  if seek('M' + padr(upper(toParameter.MenuItem), ;
    len(ABBREV)))
    lcCode = DATA
  endif seek('M' ...
  use
  select (lnSelect)
  if not empty(lcCode)
```

```
    llReturn = execscript(lcCode, toParameter)
    if llReturn
      toParameter.ValueType = 'V'
    endif llReturn
  endif not empty(lcCode)
catch
endtry
return llReturn
```

Run StandardMenuHit.PRG to install this code.

There are a few interesting things about this code. First, I normally use ORDER <tag name> rather than ORDER <n> to set the order for a table. However, the IntelliSense table is unusual: if you open the table and use ATAGINFO() to retrieve information about the indexes for this table, you'll see that there are two tags, both marked as primary and both without tag names. So, you have to use ORDER 1 or ORDER 2 to set the order for this table. The second thing is that the code is wrapped in a TRY structure to prevent any errors, such as problems opening the table or errors that may exist in the code in the other record. The third issue is that this code doesn't check for the prompt of the pad the user's menu selection is from, only the bar prompt. That's because I decided to do a SEEK for performance reasons and the tag used is UPPER(TYPE + ABBREV). Since this is a small table, you could probably get away with putting the pad prompt into the EXPANDED column and using LOCATE FOR TYPE = "M" AND ABBREV = toParameter.MenuItem AND EXPANDED = toParameter.UserTyped to ensure the exact record is found if you wish.

At the time of writing, Microsoft hasn't decided whether such a "standard" MENUHIT record will exist in the IntelliSense table by default. If not, they'll likely provide a simple way to add such a record, such as through a Solution Sample. (Solution Samples are samples that show off various VFP features, and are easily accessed through the Task Pane Manager.)

One last MENUHIT issue: if the code in the DATA memo has any compile errors, you won't get an error message but instead VFP will simply ignore the code and use the default behavior. This can be annoying, of course, since you may not be exactly sure why your code fails to do what it's supposed to.

What's it good for?

Okay, so we can hook into a VFP menu item. What can we use that for?

The first thing that came to my mind was a replacement for the New Property and New Method dialogs. Since we now have the ability to display custom properties and methods in any case we desire (see my June 2004 FoxTalk column entitled "MemberData and Custom Property Editors"), I'd rather have VFP use the case I enter into the New Property or New Method dialog than to have to bring up the MemberData Editor and change the case for a new member. Also, it always annoys me that I have to click on the Add button to add a new property or method and then click on the Close button to close the dialog. Since I often just add one property or method at a time, I'd like to see a button that both adds the member and closes the dialog.

Well, since we can now trap the New Property and New Method items in the Form and Class menus, we should be able to create a replacement dialog that behaves exactly as we wish. **Figure 1** shows such a dialog. This dialog has the following features:

- It automatically updates the _MemberData property (adding that property if necessary) so the case entered for the new member is used (even for access and assign methods if they're created as well) and the member is displayed on the Favorites tab if that option is turned on in the dialog.
- It's non-modal. That means you can keep it open, add some properties or methods, switch to other things, come back, and add some more members.
- It's dockable: try tab-docking it with Properties window. This is very cool!
- It's resizable and persists its size and position to your resource (FOXUSER) file.
- It has an Add & Close button to perform both tasks with one click.
- The default value for a property is automatically set to a value based on the data type of the property if you use Hungarian names. For example, lTest would be logical, so the default value is .F. For nTest, the default is 0.

- It hides rather than disables non-applicable controls. Since this one dialog is used for both New Property and New Method for both the Class and Form Designers, some options may not be applicable for a given instance.
- It disallows invalid names when you enter them rather than when you click on Add or Add & Close.
- The Add buttons are only enabled if a name is entered.

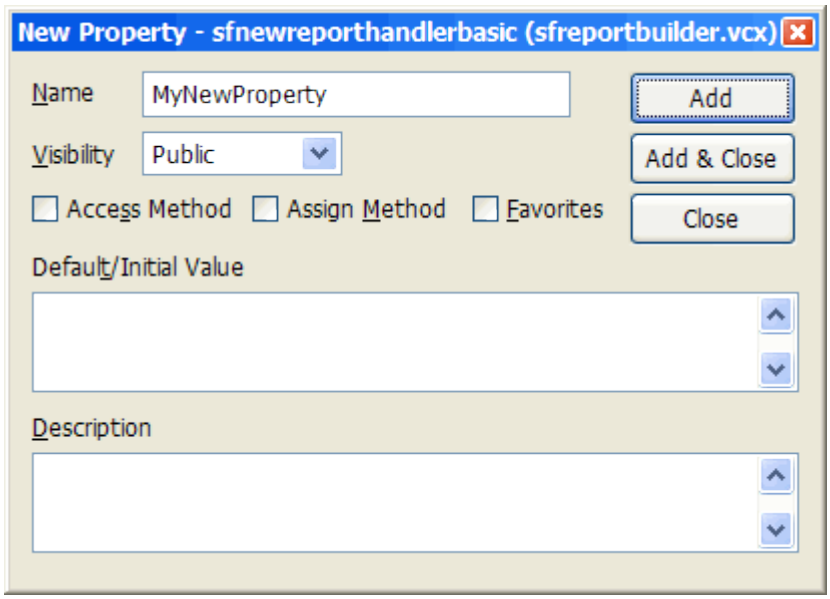

Figure 1. This replacement for the New Property and New Method dialogs has a lot more capabilities than the native dialogs.

We won't look at the code that makes up NewPropertyDialog.APP. It's not complicated code; at its heart, it calls the AddProperty and WriteMethod methods of the object being edited in the Class or Form Designer to add a new property or method. Both of these methods accept two new parameters in VFP 9: the visibility (1 for public, 2 for protected, or 3 for hidden) and description for the new property or method.

To use the replacement dialogs, simply DO NewPropertyDialog.APP to register it in the IntelliSense table. It adds the MENUHIT record discussed earlier and two records, one for "New Property" and one for "New Method", that have the same code in DATA to use the NewPropertyDialog class as the replacement dialog. In this code, "<path>" represents the path to NewPropertyDialog.APP on your system.

```
lparameters toParameter
local llReturn, ;
  llMethod, ;
  llClass
try
  llMethod = toParameter.MenuItem  = 'New Method'
  llClass  = toParameter.UserTyped = 'Class'
  release _oNewProperty
  public _oNewProperty
  _oNewProperty = newobject('NewPropertyDialog', ;
    'NewProperty.vcx', ;
    '<path>NewPropertyDialog.app', llMethod, llClass)
  _oNewProperty.Show()
  llReturn = .T.
catch
endtry
return llReturn
```

Now, when you select either New Property or New Method from the Form or Class menus, you'll get the new dialog rather than the native ones.

MENUCONTEXT

In addition to hooking into selections from the VFP system menu, you can also trap system shortcut menus, such as the one displayed when you right-click in the Properties window. This is done using the same mechanism as MENUHIT, except the ABBREV field in the IntelliSense table contains "MENUCONTEXT" rather than "MENUHIT". As with MENUHIT, the code for the MENUCONTEXT record should accept an object parameter. In this case, there are three properties of interest: Item, an array of the prompts displayed in the shortcut menu, ValueType, which has the same purpose as it does for MENUHIT, and MenuItem, the ID for the shortcut menu. Some of the values for MenuItem are "24446" for the shortcut menu for the Command window, "24447" for the Project Manager's menu, and "24456" for the Properties window's menu. How did I discover these values? I created a MENUCONTEXT record with code that simply displayed the value of toParameter.MenuItem and then I right-clicked in various places. As with MENUHIT, set ValueType to "V" or "L" and return .T. to prevent the native behavior (the display of the shortcut menu).

MENUCONTEXT isn't nearly as easy to use as MENUHIT for a variety of reasons. First, changing the contents of the Items array doesn't change the display of the menu. For example, this code doesn't seem to have any effect on the shortcut menu at all:

```
lparameters toParameter
toParameter.Items[1] = 'My Bar'
```

Similarly, the following code doesn't result in a new bar in the menu:

```
lparameters toParameter
lnItems = alen(toParameter.Items) + 1
dimension toParameter.Items[lnItems]
toParameter.Items[lnItems] = 'My Bar'
```

Second, there isn't a way to change what happens when a bar is selected. For example, you may want the Properties bar of the shortcut menu for the Command window to display your dialog rather than the native one. The problem is that none of the properties of the parameter object specify what to do when a menu item is chosen; that's built into the menu itself.

So, it appears the only way we can change the menu is to use our own shortcut menu and then prevent the native behavior. However, there's a complication with that approach: since you have to replace the entire menu with your own, how do you tell VFP to use the native behavior when some of your menu items are chosen? As of this writing, there doesn't appear to be any way to do that, so I suspect MENUCONTEXT will be used only sparingly.

Even with these issues, let's try an example anyway. In my May 2000 FoxTalk article "Zip it, Zip it Good", I presented a utility that packs all of the table-based files in a project (such as VCX, SCX, and FRX files) and another utility that zips all of the files referenced in a project into one ZIP file. In that article, these utilities were called from a toolbar displayed by a project hook for the project, but that relied on using a project hook. Let's put them in the shortcut menu for any project.

MENUCONTEXT has the same potential conflict issue as MENUHIT, so we'll use the same solution: a "standard" record that simply delegates to another record for a particular shortcut menu. In fact, the code for this standard record is exactly the same as it is for the MENUHIT record (run StandardMenuContext.PRG to create the MENUCONTEXT record). However, instead of putting the bar prompt into the ABBREV field for handler records, we'll put the menu ID. (You may want to put the menu purpose, such as "Command window", into the EXPANDED field, to make it obvious what the record is for.)

After creating the MENUCONTEXT record, I created a record with TYPE = "M", ABBREV = "24447" (the ID for the Project Manager's shortcut menu), and the following code in DATA:

```
lparameters toParameter
local loMenu
loMenu = newobject('_ShortcutMenu', ;
```

```
  home() + 'ffc\_menu.vcx')
loMenu.AddMenuBar('\<Pack Project', ;
  'do PACKPROJ with _vfp.ActiveProject.Name')
loMenu.AddMenuBar('\<Zip Project', ;
  'do ZIPPROJ with _vfp.ActiveProject.Name')
loMenu.AddMenuBar('Project \<Info...', ;
  'keyboard "{CTRL+J}" plain')
loMenu.ShowMenu()
```

This code uses the FFC (FoxPro Foundation Classes) _ShortcutMenu class to provide the shortcut menu. I discussed this class in my February 1999 FoxTalk article, "A Last Look at the FFC".

Run ProjectMenu.PRG to create this record in the IntelliSense table.

When you right-click in the Project Manager, our custom menu shows three bars (see **Figure 2**): Pack Project, which calls PACKPROJ.PRG with the name of the current project file, Zip Project, which calls ZIPPROJ.PRG with the name of the current project file, and Project Info, which uses the KEYBOARD command to invoke the Project Info dialog from the Project menu. (The latter function shows that you can reproduce the functionality of a bar in the native shortcut menu as long as there's a system menu function it calls.)

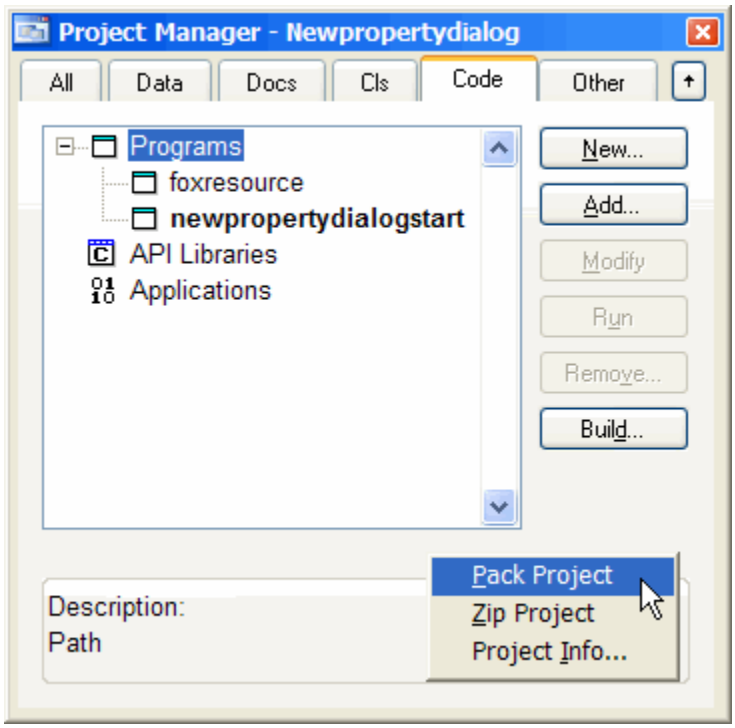

Figure 2. This custom shortcut menu for the Project Manager provides functions to pack or zip a project's file or display the Project Info dialog.

Summary

The new MENUHIT and MENUCONTEXT features allow us to hook into the VFP IDE more tightly than even before. In addition to the uses I presented in this article, I can see replacements for the Edit Property/Method dialog and the New, Import, and Export functions in the File menu. I'm sure you can think of IDE functions you'd like to see implemented differently in VFP 9; please let me know what ideas you have for this cool new feature.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and author of the MemberData Editor that ships with VFP 9. He is co-author of "What's New in Visual FoxPro 8.0," "What's New in Visual FoxPro 7.0," and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997

and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com