IntelliSense at Runtime

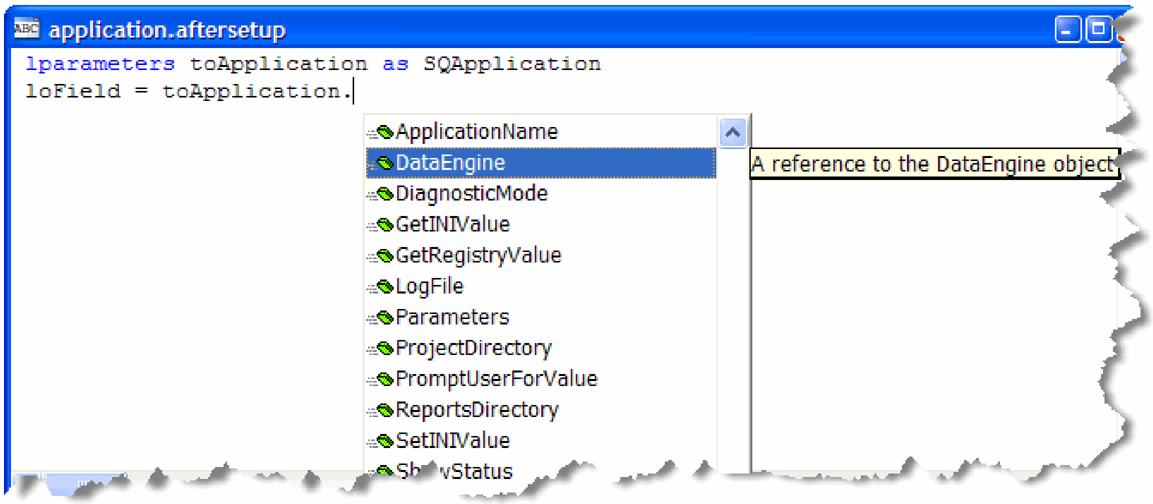
Doug Hennig

VFP 9 provides support for IntelliSense at runtime. This month, Doug Hennig examines why this is useful, discusses how to implement it, and extends Favorites for IntelliSense to support it.

As I mentioned in my August 2005 FoxTalk article, “Favorites for IntelliSense,” IntelliSense is easily the best feature ever added to Visual FoxPro. It provides a greater productivity boost to VFP developers than anything added before or since. However, until VFP 9, it was restricted to the development environment. Starting in VFP 9, IntelliSense is supported in a runtime environment as well. Before we look at how to do that, let’s discuss why.

Lately, I’ve been providing hooks into my applications to allow them to be customized. For example, in Stonefield Query, the user can specify code that should fire at lots of places: when the application starts up and shuts down, after the data dictionary has been loaded (so you can dynamically alter it if necessary), before the user logs in, after the data for a query has been retrieved but before the report is run, after a report has run, and so forth. The reason for doing this is flexibility; I can’t possibly come up with every configuration change every user could ever require, so I let them do it themselves. Obviously, this requires knowledge of the VFP language, but that isn’t too onerous for developers, IT staff, or power users, especially for simple changes.

In the VFP 8 version, a user could type the necessary code into editboxes provided in the appropriate code editing dialogs, but didn’t have the benefit of IntelliSense. In the VFP 9 version, not only is there IntelliSense on VFP commands and functions, there’s also IntelliSense on the Stonefield Query object model. **Figure 1** shows an example of what the VFP 9 version looks like when entering code. Notice that there’s both syntax coloring and member list IntelliSense in this example.



```
lparameters toApplication as SQApplication
loField = toApplication.
```

Figure 1. IntelliSense at runtime is useful if you allow users to script your applications.

In her session at the 2004 DevEssentials conference, Toni Feltman presented several other, less technical, uses for IntelliSense at runtime, such as providing something similar to Microsoft Office’s AutoCorrect feature and IntelliSense for business information such as medical diagnosis.

How IntelliSense works

IntelliSense works by running the program specified in the _CODESENSE system variable and using the table specified in the _FOXCODE system variable. By default, in a development environment, _CODESENSE points to FOXCODE.APP in the VFP program directory and _FOXCODE points to FOXCODE.DBF in the directory specified by HOME(7). Both of these are blank by default in a runtime environment. One of the tasks necessary to providing IntelliSense at runtime is setting these system variables to the appropriate values in your code and providing the IntelliSense application and table to your users.

Although I could ship FOXCODE.APP with my application, I really wanted a customized IntelliSense application that supports Favorites for IntelliSense (FFI) as I discussed in my August article, so I looked into what it would take to create one. Looking at the source code for FOXCODE.APP in the Tools\XSource\VFPSource\FoxCode directory of the VFP program directory (created by unzipping XSOURCE.ZIP in Tools\XSource), I noticed that FOXCODE.PRG is the main program for FOXCODE.APP. My first thought was to simply include FOXCODE.PRG in my application and set _CODESENSE to "FOXCODE.PRG." Unfortunately, IntelliSense didn't work when I did that, probably because VFP couldn't look inside the running EXE to find the specified program. So, I decided to create a custom FOXCODE.EXE instead.

To create this EXE, I started by copying FOXCODE.PRG and FOXCODE.H from Tools\XSource\VFPSource\FoxCode. (Because I needed to make some changes to the PRG, I copied it and made changes to the copy rather than simply adding the program to my project.) Next, I commented out the DO FORM calls in the main code and the GetInterface method in FOXCODE.PRG because otherwise these calls would cause the project manager to pull a number of files into the project that we won't need at runtime. I then created a project called FOXCODE.PJX and added the customized FOXCODE.PRG, FFI.VCX (the FFI class library; see my August article for details), the FFI.DBF that contains the FFI records for my application's objects, a CONFIG.FPW file that simply has RESOURCE=OFF (this prevents FOXUSER.DBF from being created if the user accidentally runs FOXCODE.EXE), and FOXCODE2.DBF from Tools\XSource\VFPSource\FoxCode (this is used by code in FOXCODE.PRG). Finally, I built FOXCODE.EXE from this project.

A couple of other things I tried also didn't work:

- I included the image files FFI uses for member lists, PROPTY.BMP and METHOD.BMP, in FOXCODE.PJX but the member lists had no images. I found that these image files either had to be included in my main application's project (SAMPLE.PJX in this case) or provided as separate files.
- I tried including the IntelliSense table, FOXCODE.DBF, in FOXCODE.PJX and/or SAMPLE.PJX. In either case, IntelliSense gave an error that the IntelliSense table couldn't be found. So, you must ship FOXCODE.DBF and FPT as separate files. You can rename them to something else; simply change the _FOXCODE system variable to specify the new name. Also, you don't have to provide the copy of the IntelliSense table you use in the development environment; you can make a copy of it and edit the contents as you see fit, such as removing IntelliSense records for certain commands or functions that don't make sense in a runtime environment or adding business-related records for shortcuts in memo fields.

Now that I have a customized IntelliSense application and table, all I need to do to tell my application about it is set _CODESENSE and _FOXCODE to the appropriate values. Here's the code from STARTUP.PRG, the main program for SAMPLE.PJX. Notice that it saves and restores _CODESENSE and _FOXCODE if we're running in a development environment so I don't mess up my usual IntelliSense settings during testing.

```
* Set up IntelliSense.

local lcCodeSense, ;
  lcFoxCode
if version(2) = 2
  lcCodeSense = _codesense
  lcFoxCode   = _foxcode
endif version(2) = 2
_codesense = 'FoxCode.EXE'
_foxcode   = 'FoxCode.DBF'

* Do the normal application stuff here (this is a
* demo, so we'll just run the Script Editor form).

_screen.Caption = 'Sample Application'
```

```
do form ScriptEditor

* If we're in development mode, restore the
* IntelliSense settings before exiting.

if version(2) = 2
  _codesense = lcCodeSense
  _foxcode   = lcFoxCode
endif version(2) = 2
```

Updated Favorites for IntelliSense

I had to do a few tweaks to FFI.VCX to handle IntelliSense at runtime:

- I moved FFIBuilderForm from FFI.VCX to a new class library named FFIBUILDER.VCX, since we don't need the overhead of that visual class in a runtime EXE.
- I changed FFIFoxCode.DisplayMembers to not use a path for the image files in a runtime environment, since they'll be built into the EXE.
- This is a simplication change rather than one required for IntelliSense at runtime. Rather than requiring a different script record for each namespace in the IntelliSense table, I created a single generic HandleFFI script record and made all namespace records (the ones with TYPE = "T" and ABBREV set to the namespace for the object we want IntelliSense for) use HandleFFI as their script record. The code for HandleFFI now looks for FFI.VCX in FOXCODE.EXE and no longer passes a hard-coded class name or library to FFIFoxCode.Main. Here's the updated script code:

```
lparameters toFoxCode
local loFoxCodeLoader, ;
  luReturn
if file(_codesense)
  set procedure to (_codesense) additive
  loFoxCodeLoader = createobject('FoxCodeLoader')
  luReturn        = loFoxCodeLoader.Start(toFoxCode)
  loFoxCodeLoader = .NULL.
  if atc(_codesense, set('PROCEDURE')) > 0
    release procedure (_codesense)
  endif atc(_codesense, set('PROCEDURE')) > 0
else
  luReturn = ''
endif file(_codesense)
return luReturn

define class FoxCodeLoader as FoxCodeScript
  cProxyClass    = 'FFIFoxCode'
  cProxyClasslib = 'FFI.vcx'
  cProxyEXE      = 'Path\FoxCode.EXE'

  procedure Main
    local loFoxCode, ;
      luReturn
    loFoxCode = newobject(This.cProxyClass, ;
      This.cProxyClasslib, This.cProxyEXE)
    if vartype(loFoxCode) = 'O'
      luReturn = loFoxCode.Main(This.oFoxCode)
    else
      luReturn = ''
    endif vartype(loFoxCode) = 'O'
    return luReturn
  endproc
enddefine
```

- As a result of the preceding change, FFIFoxCode.Main now only accepts toFoxCode as a parameter; the earlier version also accepted the namespace, class, and library of the desired class. Since we can get the namespace from the Data member of the toFoxCode object and the class and library from the appropriate record in FFI.DBF, there's no need to pass those parameters. Here's the code for this method:

```
lparameters toFoxCode
local lcNameSpace, ;
  loData, ;
  lcReturn, ;
  lcTrigger
with toFoxCode

* Get the namespace and an object from the FFI table
* for that namespace.

  lcNameSpace = .Data
  loData      = This.GetFFIMember(.UserTyped, ;
    lcNameSpace)
  lcReturn    = ''

* If we're on the LOCAL statement, handle that by
* returning text we want inserted.

  if atc(lcNameSpace, .MenuItem) > 0
    lcReturn = This.HandleLOCAL(toFoxCode, ;
      lcNameSpace, trim(loData.Class), ;
      trim(loData.Library))
  else
* Rest of the code unchanged
```

I made a couple of other changes to FFIBuilderForm thanks to suggestions by reader Michael Hawksworth. First, I adjusted the LoadTree method so in addition to handling any member data for the object being worked on by the builder form, it also picks up any global member data. The second change was in GetObjectReference, which now properly handles objects based on Form.

Using IntelliSense at runtime

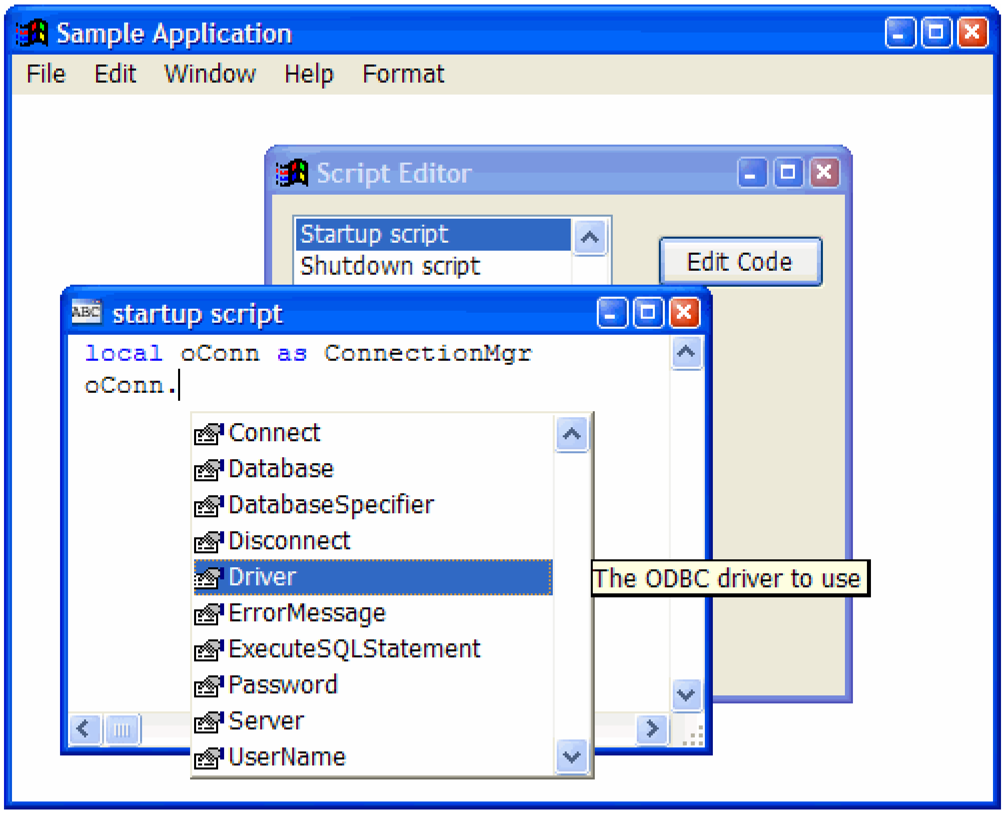
Now that we've got it set up, how do we use IntelliSense at runtime? Like design time, there are two places IntelliSense is supported at runtime: in a code (PRG) window and in a memo field. It'd be really nice if it was supported in an editbox because of the control we have over that object, but no such luck. Note that getting IntelliSense to work with a memo field is a little tricky: you need to ensure that word wrap is turned off and syntax coloring is turned on, which means providing a custom FOXUSER resource file with these settings in place. Because of this, my preference is to use a PRG. Even if what the user types ultimately ends up in a memo field, you can still copy the contents of the memo field to a file, use a PRG window to modify that file, and then write the file contents back to the memo field. That's what the sample application does.

The Click method of the Edit Code button in ScriptEditor.SCX, the main form in the sample application, has the following code:

```
local lcPath, ;
  loForm

* Write the current contents of CODE to a file.

lcPath = addbs(sys(2023)) + trim(NAME)
strtofile(CODE, lcPath)

* Create a form with the desired characteristics for
* the PRG window.

loForm = createobject('Form')
with loForm
  .Caption  = trim(NAME)
  .Width    = _screen.Width  - 50
  .Height   = _screen.Height - 50
  .FontName = 'Courier New'
  .FontSize = 10
endwith
```

```
* Edit the code in a PRG window, then put the results
* back into CODE.

modify command (lcPath) window (loForm.Name)
replace CODE with filetostr(lcPath)
erase (lcPath)
```

This code writes the current contents of the record to a PRG in the user's temporary files directory, creates a form with the characteristics we want for the editing window, edits the file, and then puts the contents of the PRG back into the table and deletes the PRG file. **Figure 2** shows how IntelliSense looks when you run the sample application, choose a script, and click on the button.



```
local oConn as ConnectionMgr
oConn.
```

Figure 2. IntelliSense works at runtime as well as design time.

Notice that other than certain things like font, window size, and placement, you don't have much control over the window the PRG appears in. One undesirable behavior is that clicking outside the code window causes it to be closed automatically (unless you use NOWAIT, which then introduces other complications). Also, if your application is in a top-level form, there are other issues. If you don't use the IN WINDOW clause for the MODIFY COMMAND statement and specify the name of your top-level form, and _SCREEN isn't visible, which it normally isn't when you use a top-level form, the code window won't appear. Using IN WINDOW makes the code window a child of the top-level form, so it can't be moved outside or sized larger than the form. So, we just have to live with the lack of control we have over these windows, I'm afraid.

I also ran into what looks like a couple of bugs in IntelliSense while testing this:

- The list of types that normally appears when you type LOCAL SomeVariable AS doesn't show up at runtime.
- If you have a object that's a member of another object (such as the User member of the cApplication object in this month's samples, which contains a reference to a cUser object), the

IntelliSense that appears for that member is for the parent object. For example, if you type LOCAL loUser as Application.User (the cApplication class is registered with the Application namespace), when you type “loUser” and a period, the member list IntelliSense displays is for the cApplication class, not the cUser class.

Summary

IntelliSense is one of those things we VFP developers can’t live without once we started using it. Now, you can give your users the same benefits in your applications. Obviously, this technique isn’t for everyone, but if it fits in your application, I guarantee your users will love you for it.

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, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the “What’s New in Visual FoxPro” series 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 long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com