Adding IntelliSense to Your Applications

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com
Blog: http://doughennig.blogspot.com

Overview

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. Now, IntelliSense is supported in a runtime environment as well. This document discusses why this is a useful feature for many types of application and shows you how to create a customized IntelliSense environment specific for your application.

Introduction

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.

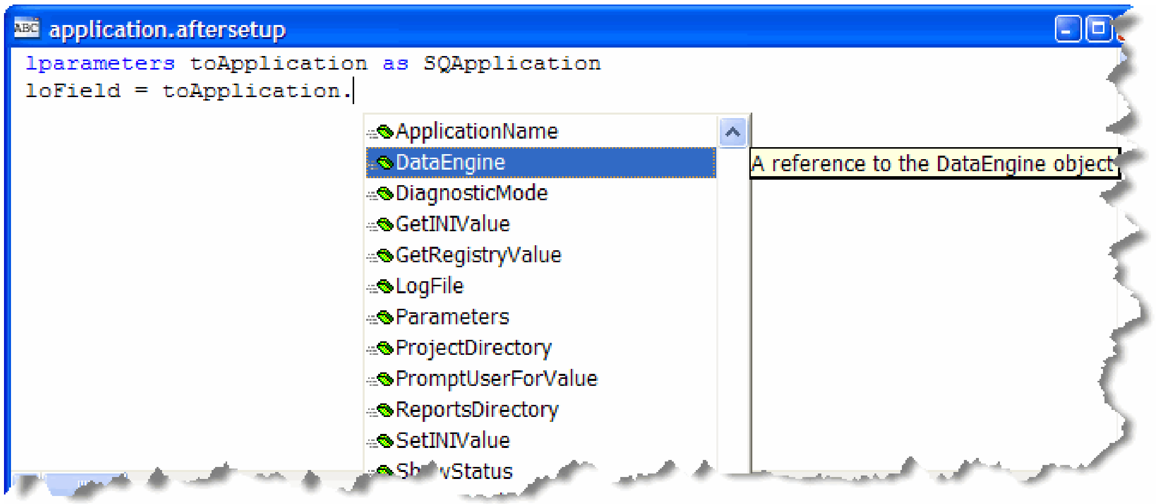

```
lparameters toApplication as SQApplication
loField = toApplication.
```

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

In her September 2004 article in FoxPro Advisor titled "Take Advantage of IntelliSense in VFP 9 Applications," 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.

In this document, I discuss two aspects to using IntelliSense within applications: implementing IntelliSense at runtime and creating customized IntelliSense for your application.

Setting Up IntelliSense for Runtime

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 the customization features I'll discuss later in this document. 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, 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.

One other thing I tried also didn't work: including the IntelliSense table, FOXCODE.DBF, in FOXCODE.PJX and/or the project for my application. 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, _FOXCODE, the path, and system menu if we're running in a development environment so I don't mess these settings up during testing.

```
* Set up IntelliSense.

local lcCodeSense, ;
  lcFoxCode, ;
  lcCaption, ;
  lcPath
if version(2) = 2
  lcCodeSense = _codesense
  lcFoxCode   = _foxcode
  lcCaption   = _screen.Caption
  lcPath      = set('PATH')
  set path to SOURCE
  push menu _msysmenu
endif version(2) = 2
_codesense = 'FoxCode.EXE'
_foxcode   = 'FoxCode.DBF'

* Do the normal application stuff here.

_screen.Caption = 'Sample Application'
do SampleMenu.mpr
read events

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

if version(2) = 2
  pop menu _msysmenu
  _codesense      = lcCodeSense
  _foxcode        = lcFoxCode
  _screen.Caption = lcCaption
  set path to &lcPath
endif version(2) = 2
```

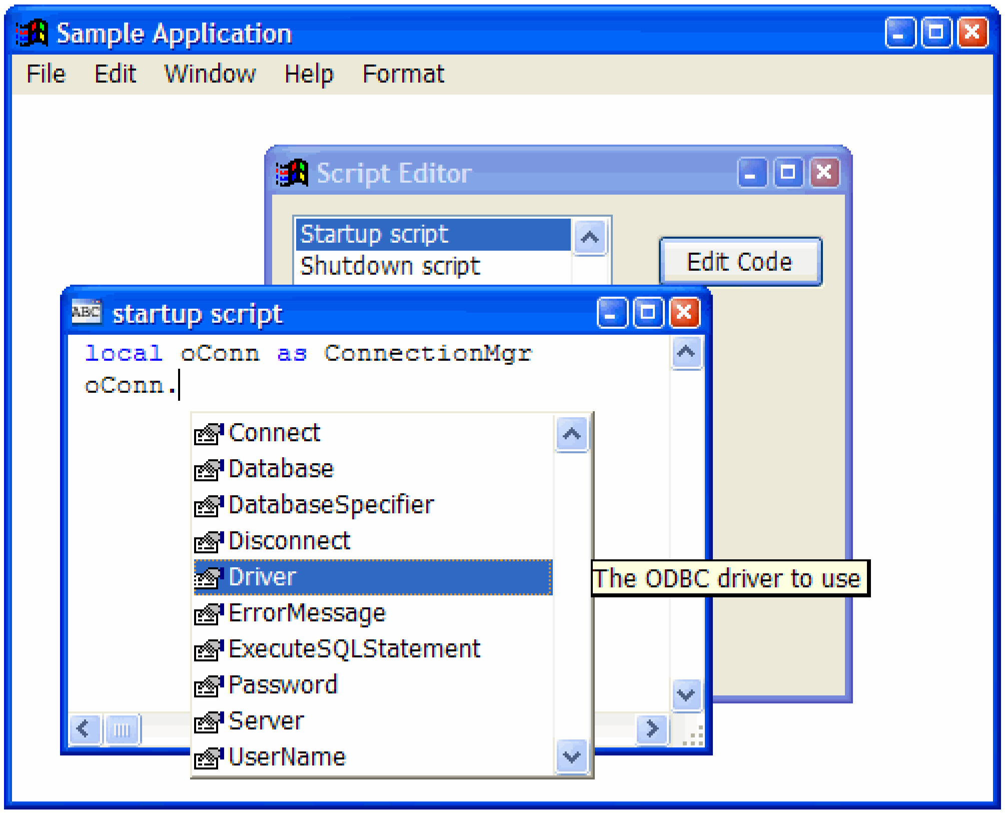
Using IntelliSense at Runtime

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, one of the forms 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.

Of course, IntelliSense can be used for more than editing code; it's also useful for things like text expansion, similar to the AutoCorrect feature in Microsoft Word. For example, in a time billing application, descriptions entered for timecard entries often contain frequently-used text, such as "Worked on" or "Met with." It would be nice if the user could type an abbreviation and have it automatically expand to the desired text. This is easy to do with IntelliSense: create records with TYPE = "U," ABBREV = the abbreviation to use, and EXPANDED to the expanded text. Even better, you can have a list of values to choose from pop up by specifying "{cmdhandler}" in CMD and a list of the values to display (each on its own line) in DATA. **Figure 3** shows what happens when the user types "wo" and presses Space in the editor window for the Comment editbox in SampleForm.SCX.

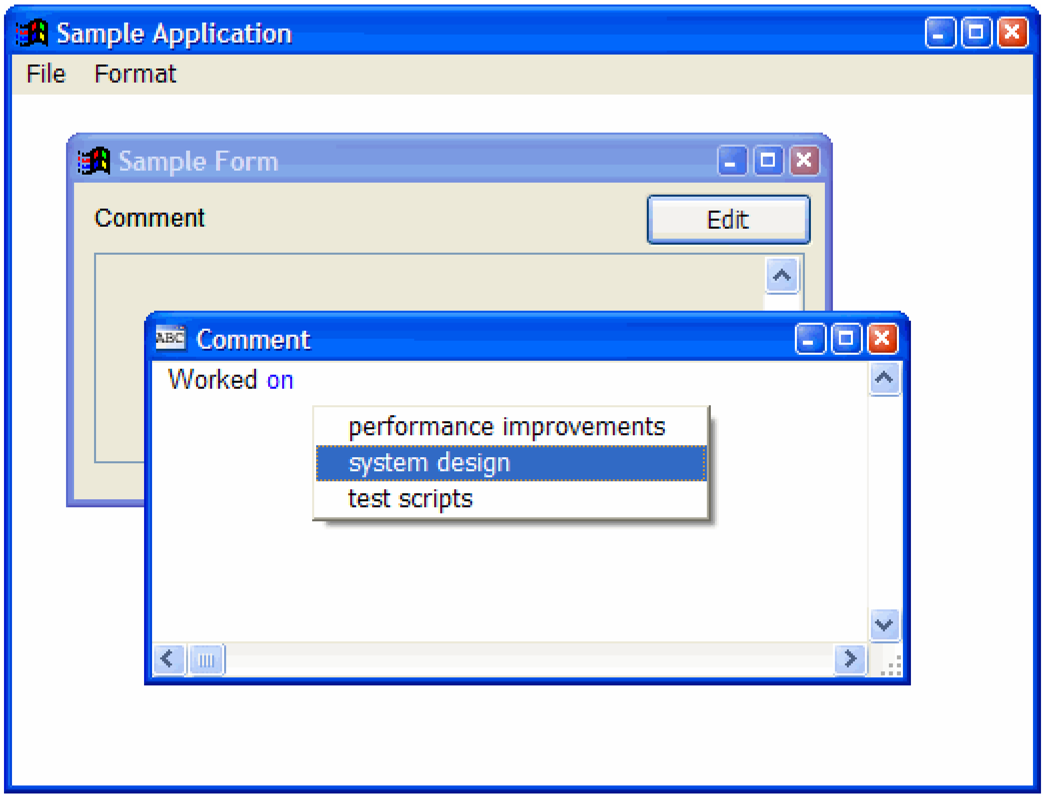
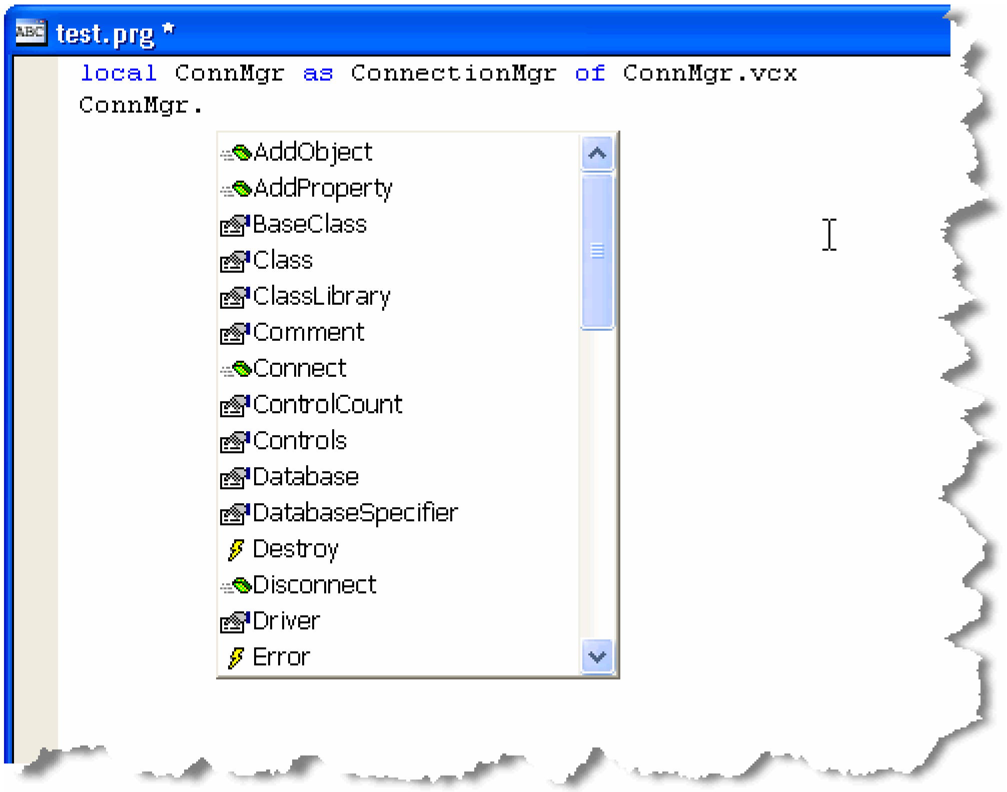


Figure 3*. Not only does IntelliSense allow text to be expanded, it can include a list of additional values.*

The Problem with IntelliSense

Although IntelliSense is the greatest thing since sliced bread, one thing that bugs me about it is that when used with a class, it displays all members of that class rather than the ones I really want to see.

For example, **Figure 4** shows the IntelliSense display for the ConnectionMgr class. Note that although there are only a few custom properties and methods we're interested in, IntelliSense displays everything. This requires more effort to select the exact member you want, especially if you're not very familiar with the class.



```
local ConnMgr as ConnectionMgr of ConnMgr.vcx
ConnMgr.
```

Figure 4*. Although IntelliSense allows you to choose a member name from a list, it displays more items than we usually need.*

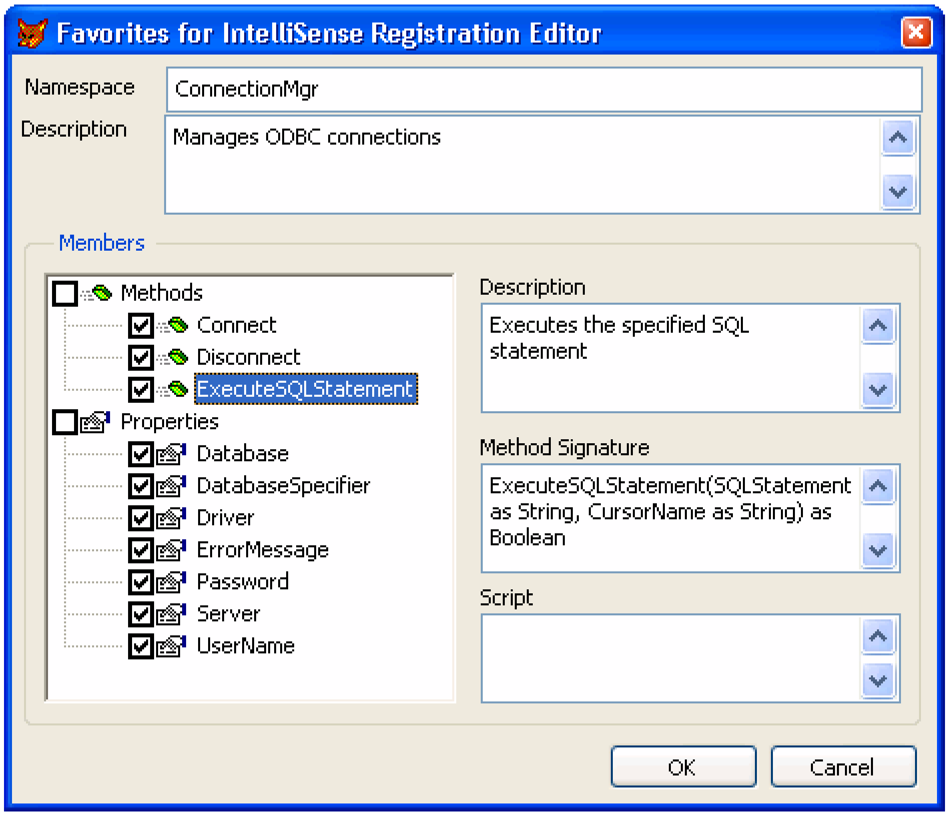
We had a similar problem with the Properties window until VFP 9. In that version, Microsoft added a Favorites tab that displays only those members you defined as belonging on that tab. (One way to do that is to right-click the member in the Properties window and choose Add to Favorites.) So now, rather than wading through dozens of members to find the one you want, you can pick among the few you've specified on the Favorites tab.

What I'd really like to see is the equivalent of the Favorites tab for IntelliSense. So, I decided to create it. The result is Favorites for IntelliSense (FFI).

Favorites for IntelliSense

Before we look under the hood, let's see FFI in action. The first thing we need to do is register a class with FFI. Open the ConnectionMgr class in ConnMgr.VCX (included in the source code accompanying this document), then type DO FORM FFIBUILDER in the Command window. You should see the dialog shown in **Figure 5**. This dialog allows you to specify the "namespace" for the class. The namespace is the name that appears in the IntelliSense list when you type LOCAL SomeVariable AS. It defaults to the name of the class, but you can specify something else if you wish. For example, for a class named cApplication, you may want the namespace to be Application instead.

The description is used as the tooltip for the class in the IntelliSense list if this class is used as a member of another class (for example, the User property of the sample cApplication class contains an instance of the cUser class, so that class' description is used for the User property). It defaults to the description for the class as specified in the Class Info function in the Class menu or by choosing the Edit Description function in the Project menu when the class is selected in the Project Manager.



Namespace: ConnectionMgr

Description: Manages ODBC connections

- ☐ Methods
  - ☑ Connect
  - ☑ Disconnect
  - ☑ ExecuteSQLStatement
- ☐ Properties
  - ☑ Database
  - ☑ DatabaseSpecifier
  - ☑ Driver
  - ☑ ErrorMessage
  - ☑ Password
  - ☑ Server
  - ☑ UserName

Description: Executes the specified SQL statement

Method Signature: ExecuteSQLStatement(SQLStatement as String, CursorName as String) as Boolean

Script:

Figure 5*. The Favorites for IntelliSense builder allows you to specify what appears in IntelliSense for the selected class.*

The TreeView shows public custom properties and methods for the class; if you want native members displayed as well, change the AMEMBERS() statement in the LoadTree method of the FFIBuilderForm class in FFIBuilderForm.VCX. The checkbox before the name indicates whether the member is included in IntelliSense or not; by default, all custom members are included.

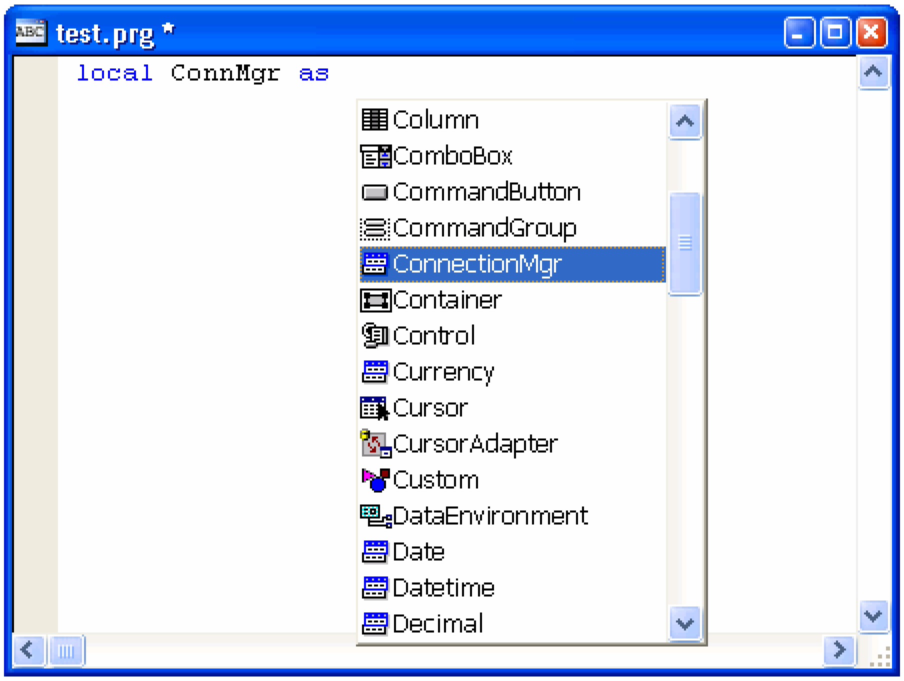
The description is used as the tooltip for the member in the IntelliSense list and defaults to the description you entered for the member when you created it. The method signature is displayed as a tooltip for a method when you type an open parenthesis or a comma in the parameter list for the method. This tooltip shows you what parameters can be passed to the method. The signature defaults to the method name and the contents of any LPARAMETERS statement in the method, but you can edit it to display anything you wish, including the data type of the return value.

The script editbox allows you to enter code that's executed when you select this member and type an open parenthesis, a comma in the parameter list, or an equals sign (to assign a value to a property). We'll see an example of this later.

Make any changes you wish in this dialog, then click OK. This form adds records for the class and each of the registered members to an FFI table. It also adds two records to your IntelliSense table if necessary: one for the class and one for an IntelliSense script for FFI. We'll look at this in more detail later, as well.

To see how Favorites for IntelliSense works with this class, create a PRG and type LOCAL ConnMgr AS. When you press Space after AS, you should see the IntelliSense list of types shown in **Figure 6**. When you select ConnectionMgr and press Enter, you'll see the following code inserted into the PRG (where Path is the path for the VCX):

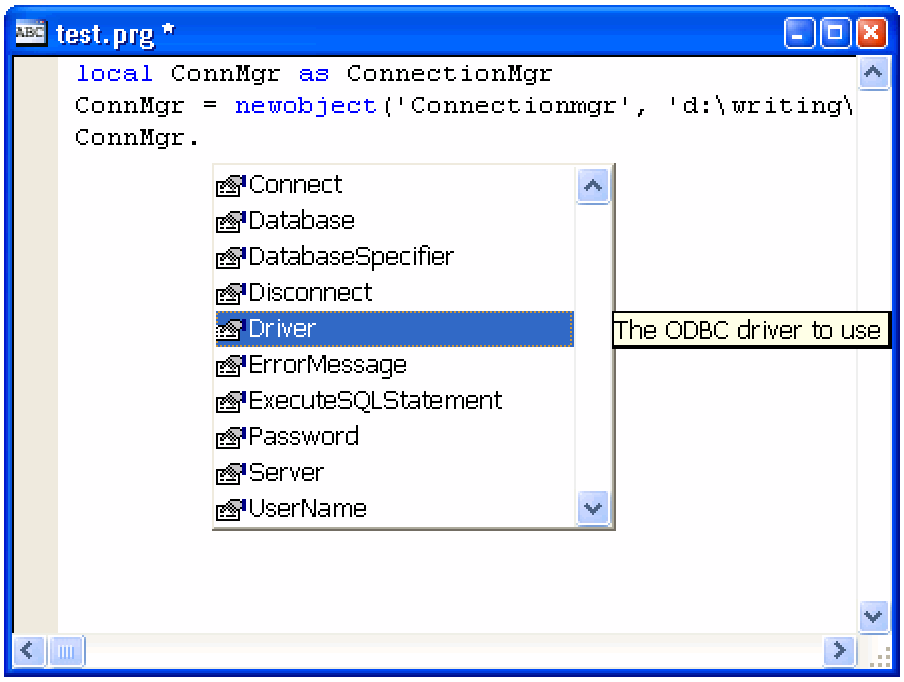
```
local ConnMgr as ConnectionMgr
ConnMgr = newobject('Connectionmgr', 'Path\connmgr.vcx')
```



```
local ConnMgr as
```

Figure 6. *Any class registered with Favorites for IntelliSense is displayed as a type in IntelliSense.*

Now type ConnMgr followed by a period. As you can see in **Figure 7**, IntelliSense displays only registered members of the class (hence, "Favorites for IntelliSense"). If you select a method such as ExecuteSQLStatement, when you type the opening parenthesis, you'll see the method signature as the IntelliSense tooltip, making it much easier to see what parameters to pass to the method.



```
local ConnMgr as ConnectionMgr
ConnMgr = newobject('Connectionmgr', 'd:\writing\
ConnMgr.
```

Figure 7. *IntelliSense displays only the registered members for the specified class. Compare this to Figure 4.*

How FFI Works

The secret behind FFI lies in two things: how IntelliSense deals with things defined as “types” in the IntelliSense table, and IntelliSense scripts. Types are normally use for data types (such as Integer or Character) and base classes (such as Checkbox and Form). However, you can define other things as types, either by manually adding records with TYPE set to “T” or by using the IntelliSense Manager in the Tools menu.

Other kinds of type records are usually custom classes or COM objects, allowing you to get IntelliSense on them. That’s what we’ll use type records for as well, but we’ll customize how IntelliSense works by using a script and a custom IntelliSense-handling class.

If you look in your IntelliSense table (USE (_FOXCODE) AGAIN and BROWSE) after using the FFIBUILDER form to register a class, you’ll see two new records at the end of the table. One is the type record for the class; it doesn’t contain much information other than the namespace you defined in the builder form and the name of a script to use for IntelliSense for the class; that’s specified as “{HandleFFI}” in the CMD column. The other is a script record, with TYPE set to “S” and ABBREV containing “HandleFFI,” the name specified in the CMD column of the T record.

The script record has the following code in its DATA memo (in this code, Path1 is replaced with the path for FFI.VCX and Path2 is replaced with the path for FoxCode.EXE):

```
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 = 'Path1\FFI.vcx'
  cProxyEXE      = 'Path2\FoxCode.EXE'

  procedure Main
    local loFoxCode, ;
      luReturn
    if version(2) = 2
      loFoxCode = newobject(This.cProxyClass, This.cProxyClasslib)
    else
      loFoxCode = newobject(This.cProxyClass, ;
        juststem(This.cProxyClasslib), This.cProxyEXE)
    endif version(2) = 2
    if vartype(loFoxCode) = 'O'
      luReturn = loFoxCode.Main(This.oFoxCode)
    else
      luReturn = ''
    endif vartype(loFoxCode) = 'O'
    return luReturn
  endproc
enddefine
```

This code defines a subclass of the FoxCodeScript class defined in the IntelliSense application specified by the _CODESENSE system variable. This subclass overrides the Main method (which is called by IntelliSense) to instantiate the FFIFoxCode class in FFI.VCX and call its Main method. The call passes a reference to the IntelliSense data object, which contains information about what the user typed and other IntelliSense settings. Note FFIFoxCode is instantiated either within FoxCode.EXE or the standalone VCX, depending on whether we’re in a runtime environment or not; we don’t want to use FoxCode.EXE in a development environment because it has a table called FFI.DBF, which I'll discuss shortly, built into it, and we don't want that copy of the table in that case because it might contain outdated records.

The HandleFFI script is used for all classes registered with FFI, so there's only a single record for that script in the table.

As a result of this script, FFIFoxCode.Main is called for all IntelliSense tasks for any FFI-registered namespace, such as when you select the namespace from the IntelliSense list displayed when you type LOCAL SomeVariable AS or when you type one of the "trigger" characters—such as a period, an opening parenthesis, or an equals sign—in a statement containing the name of the variable the class is instantiated into.

FFIFoxCode

The FFIFoxCode class does all of the custom IntelliSense work for FFI, so let's examine this class in detail.

The Init method does just two things:

- Turns on debugging in system components (without this, you can't easily debug problems in the code)
- Opens FFI.DBF (the table containing registration information for a class and its members) by calling OpenFFITable. If the table can't be opened, Init displays an error message and returns .F. so the class isn't instantiated.

```
* Turn debugging on.

sys(2030, 1)

* Open the FFI (Favorites for IntelliSense) table.

local llReturn
llReturn = This.OpenFFITable()
if not llReturn
  messagebox('Could not open the Favorites for IntelliSense table.', 64, 'IntelliSense Handler')
endif not llReturn
return llReturn
```

The Main method, called from the IntelliSense script, handles all of the IntelliSense tasks for FFI. As we saw earlier, the script passes a FoxCode object to Main. The Data property of that object contains the content of the Data memo of the appropriate record in the FoxCode table, which happens to be the record for the registered class, in which case Data contains the namespace of the class. Main calls the GetFFIMember method to determine which member of the class you typed (it could also be the class itself) and return a SCATTER NAME object from the appropriate record in the FFI table.

If the namespace is found in the MenuItem property of the FoxCode object (MenuItem contains the prompt of the item chosen by the user if IntelliSense was triggered by selecting an item from an IntelliSense menu), we must be on the LOCAL SomeVariable AS statement, so Main calls the HandleLOCAL method to deal with it. Otherwise, Main does one of several things. If the MenuItem property of the FoxCode contains a value, the user chose an item from the list for a property assignment, so it calls the HandlePropertyAssignment method to insert the chosen value into the code (we'll see how that's used later). If the character which triggered IntelliSense is a period, we need to display a list of the registered members of the class, so Main calls DisplayMembers to do the work. If the trigger character is an opening parenthesis, an equals sign, or a comma and there's code in the SCRIPT memo of the FFI record, that code is executed. This script code could, for example, specify a list of enumerated values that IntelliSense should display for a property value or a parameter of a method (we'll see an example of that later).

Finally, if the TIP memo of the FFI record is filled it, Main uses it as the tooltip for IntelliSense. This is usually used to display the signature of a method (for example, "Login(UserName as String, Password as String) as Boolean").

```
* This is main routine that gets called from the IntelliSense script for an FFI
* class.

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

* Get the namespace and an object from the FFI table for that namespace. Also,
* get the character that triggered IntelliSense.

  lcNameSpace = .Data
  loData      = This.GetFFIMember(.UserTyped, lcNameSpace)
  lcReturn    = ''
  lcTrigger   = right(.FullLine, 1)
  do case

* The user chose an item from the list for a property assignment.

    case vartype(loData) <> 'O' and not empty(.MenuItem)
      lcReturn = This.HandlePropertyAssignment(toFoxCode)

* We can't find the member in the FFI table, so do nothing.

    case vartype(loData) <> 'O'

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

    case atc(lcNameSpace, .MenuItem) > 0
      lcReturn = This.HandleLOCAL(toFoxCode, lcNameSpace, ;
        trim(loData.Class), trim(loData.Library))

* If we were triggered by a ".", display a list of members.

    case lcTrigger = '.'
      This.DisplayMembers(toFoxCode, loData)

* If we were triggered by a "(" (to start a method parameter list), an "="
* (for a property), or "," (to enter a new parameter) and we have a script,
* execute it.

    case inlist(lcTrigger, '=', '(', ',') and not empty(loData.Script)
      lcReturn = execscript(loData.Script, toFoxCode, loData)

* If we were triggered by a "(" or "," , display the tooltip for the method
* (normally the method signature).

    case inlist(lcTrigger, '(', ',') and not empty(loData.Tip)
      .ValueTip  = loData.Tip
      .ValueType = 'T'
  endcase
endwith
return lcReturn
```

We won't look at the code for HandleLOCAL here; feel free to examine this method yourself. Main calls it when you type LOCAL SomeVariable AS and choose one of the registered namespaces from the type list. All it does is generate a NEWOBJECT() statement for the class so you don't have to. The only complication for this method is determining the case to use for NEWOBJECT().

(I could have hard-coded this as "newobject" since I always use lower-case for FoxPro keywords, but decided to be nice to other developers who may have different ideas about casing. This is solved by looking in the IntelliSense table for a record with TYPE = "F" (for "function") and ABBREV = "NEWO" (the abbreviation for NEWOBJECT) and using the value in the CASE field.)

GetFFIMember, called from Main, looks for the class or member you typed in the FFI table. It uses the UserTyped property of the FoxCode object (passed as the first parameter), which contains the text you typed pertaining to the namespace. For example, when you type "llStatus = ConnMgr.ExecuteSQLStatement(", UserTyped contains "ConnMgr.ExecuteSQLStatement".

GetFFIMember starts by finding the record for the specified class in the FFI table. If a period is included in UserTyped, it then looks for the appropriate member record. If it finds the correct record, it returns a SCATTER NAME object from that record.

```
* Determine which member of the namespace the user typed and return a SCATTER
* NAME object from the appropriate record in the FFI table.

lparameters tcUserTyped, ;
  tcNameSpace
local loReturn, ;
  lcUserTyped, ;
  llFound, ;
  lnPos, ;
  lcMember, ;
  lnSelect

* Grab what the user typed. If it ends with an opening parenthesis, strip that
* off.

loReturn    = .NULL.
lcUserTyped = alltrim(tcUserTyped)
if right(lcUserTyped, 1) = '('
  lcUserTyped = substr(lcUserTyped, len(lcUserTyped) - 1)
endif right(lcUserTyped, 1) = '('

* Find the record for the class in the FFI table. If there's a period in the
* typed text, try to find a record for the member.

if seek(upper(padr(tcNameSpace, len(__FFI.CLASS))), '__FFI', 'MEMBER')
  llFound = .T.
  lnPos   = at('.', lcUserTyped)
  if lnPos > 0
    lcMember = alltrim(__FFI.MEMBER) + substr(lcUserTyped, lnPos)
    llFound  = seek(upper(padr(lcMember, len(__FFI.MEMBER))), '__FFI', ;
      'MEMBER')
  endif lnPos > 0

* If we found the desired record, create a SCATTER NAME object for it.

  if llFound
    lnSelect = select()
    select __FFI
    scatter memo name loReturn
    select (lnSelect)
  endif llFound
endif seek(upper(padr(tcNameSpace ...
return loReturn
```

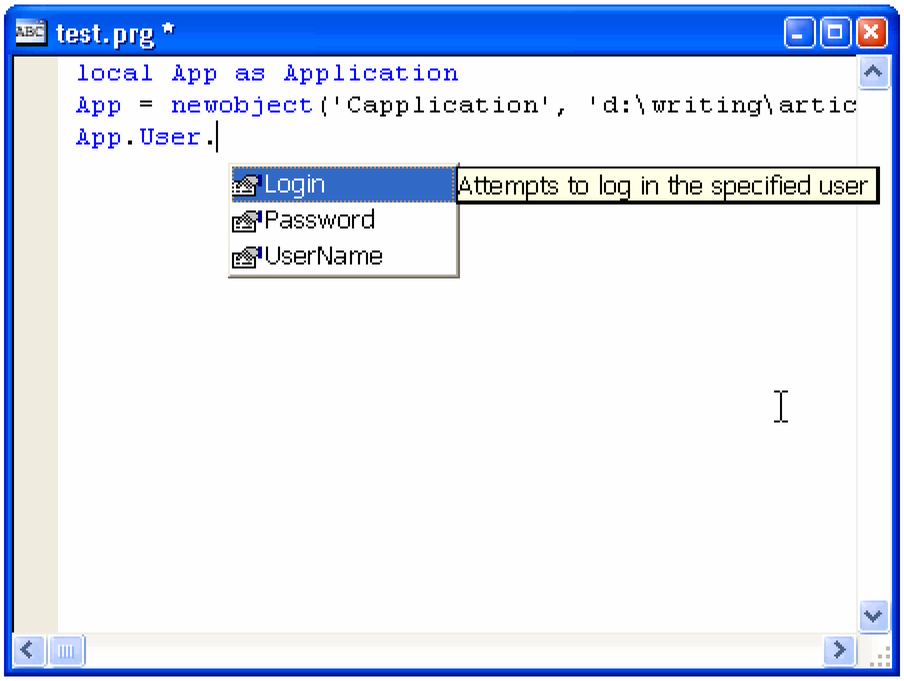
DisplayMembers is called from Main to tell IntelliSense to display a list of registered members for the class when you type a period in the command line. DisplayMembers calls GetMembers to retrieve a collection of members for the specified class (we won't look at that method here). It then fills the Items array of the FoxCode object with the names and descriptions of the members and sets the object's ValueType property to "L," which tells IntelliSense to display a listbox with the contents of the Items array.

This code shows one slight design flaw in IntelliSense: the FoxCode object has a single Icon property which contains the name of the image file to display in the listbox. What is actually needed is an additional column in the Items array, since in this case, we want to display different images for properties and methods. Unfortunately, we get only a single image displayed for all items.

```
* Builds a list of members for IntelliSense to display.

lparameters toFoxCode, ;
  toData
local loMembers, ;
  lcPath, ;
  lnI, ;
  loMember
with toFoxCode

* Get a collection of members for the current class.

  loMembers = This.GetMembers(alltrim(toData.Member))
  if loMembers.Count > 0

* Add each member to the Items array of the FoxCode object.

    dimension .Items[loMembers.Count, 2]
    lcPath = iif(file('propty.bmp'), '', home() + 'FFC\Graphics\')
    for lnI = 1 to loMembers.Count
      loMember       = loMembers.Item(lnI)
      .Items[lnI, 1] = loMember.Name
      .Items[lnI, 2] = loMember.Description
      if loMember.Type = 'P'
        .Icon = lcPath + 'propty.bmp'
      else
        .Icon = lcPath + 'method.bmp'
      endif loMember.Type = 'P'
    next loMember

* Set the FoxCode object's ValueType property to "L", meaning display a listbox
* containing the items defined in the Items array.

    .ValueType = 'L'
  endif loMembers.Count > 0
endwith
```

Some Advanced Uses

There are some interesting things you can do with FFI to make it even easier to use. One is that classes can contain a hierarchy of objects, and by registering the classes contained as members of a class with FFI, you can get clean IntelliSense on those members. For example, the cApplication class instantiates an instance of the cUser class into its User property at runtime. Normally, you wouldn't get IntelliSense on the User property because it doesn't contain an object at design time. However, if you open the cUser class and specify "Application.User" as the namespace, you get IntelliSense on the User member of the Application namespace, as you can see in **Figure 8**.



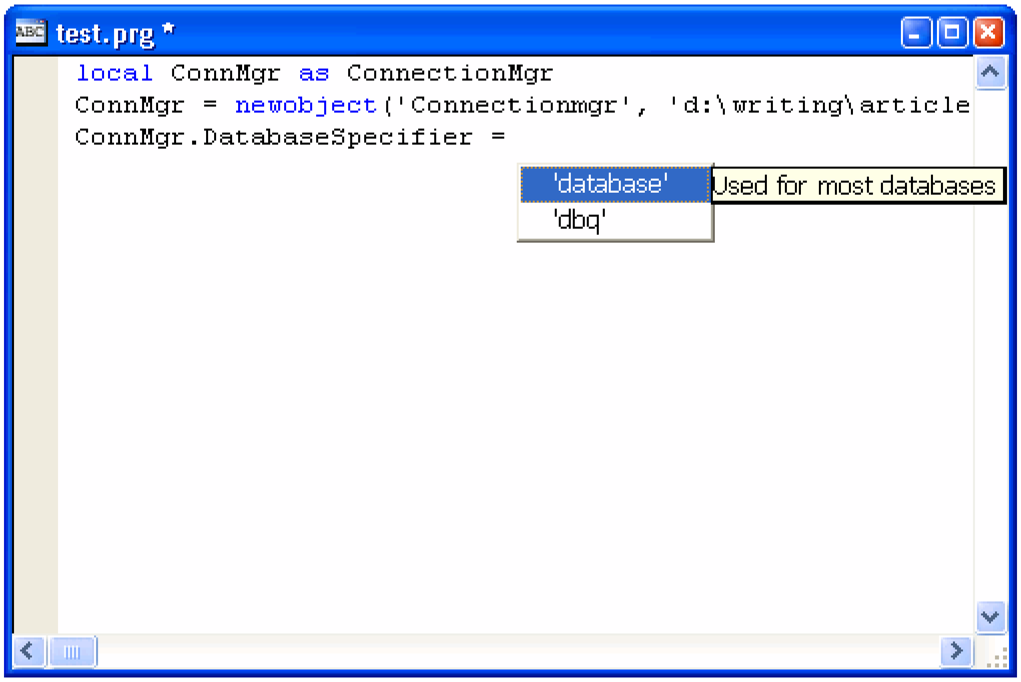
```
local App as Application
App = newobject('Capplication', 'd:\writing\artic
App.User.
```

***Figure 8**. You can even get IntelliSense on objects instantiated at runtime if they're registered in a sub-namespace of the main namespace.*

Another useful thing is to script how IntelliSense deals with a member by entering code in the Script editbox of the FFI builder form. For example, the DatabaseSpecifier property of ConnectionMgr indicates how the database should be specified in an ODBC connection string. For some databases, such as Access and dBase, you specify the database using “dbq=database path”, while for others, you specify “database=database name.”

Rather than expecting the developer to know that the valid choices for DatabaseSpecifier are “database” and “dbq” (and to type them without any mistakes), you can tell IntelliSense to display a list of the valid values (as shown in **Figure 9**) in a similar manner to the way the list of members is specified: by filling in the Items array of the FoxCode object and setting its ValueType property to “L.” The script code for DatabaseSpecifier does exactly that.

```
lparameters toFoxCode, ;
  toData
dimension toFoxCode.Items[2, 2]
toFoxCode.Items[1, 1] = 'database'
toFoxCode.Items[1, 2] = 'Used for most databases'
toFoxCode.Items[2, 1] = 'dbq'
toFoxCode.Items[2, 2] = 'Used for Access and dBase'
toFoxCode.ItemScript  = 'HandleFFI'
toFoxCode.ValueType   = 'L'
return ''
```



```
local ConnMgr as ConnectionMgr
ConnMgr = newobject('Connectionmgr', 'd:\writing\article
ConnMgr.DatabaseSpecifier =
```

***Figure 9**. Using a script, you can display a list of custom values for property assignments..*

Notice all scripts are passed a reference to the FoxCode object and the SCATTER NAME object for the FFI record. Also note this code sets ItemScript to “HandleFFI;” that tells IntelliSense to call the HandleFFI script, which in turns call FFIFoxCode.Main, after the user chooses a value. As we saw earlier, Main calls HandlePropertyAssignment when the FoxCode object’s MenuItem property contains a value, which is the case when the user chooses a value from the list we’ve provided. That method adds quotes around the value if it isn’t numeric; without that, the value wouldn’t work correctly. Finally, note the return value of the script is sent to IntelliSense, so unless you want that value inserted into the code, return an empty string and set ValueType to something other than “V.”

Here’s a script that displays a context-sensitive tooltip for some of the parameters of a method and a values list for one of them:

```
lparameters toFoxCode, ;
  toData
local lcMember, ;
  lnPos, ;
  lcParameters, ;
  lnParameters
lcMember     = alltrim(substr(toData.Member, rat('.', toData.Member) + 1))
lnPos        = atc(lcMember, toFoxCode.FullLine)
lcParameters = substr(toFoxCode.FullLine, lnPos + len(lcMember))
lnParameters = occurs(',', lcParameters) + 1
do case
  case lnParameters = 1
    toFoxCode.ValueTip  = 'First parameter tooltip'
    toFoxCode.ValueType = 'T'
  case lnParameters = 2
    toFoxCode.ValueTip  = 'Second parameter tooltip'
    toFoxCode.ValueType = 'T'
  case lnParameters = 3
    dimension toFoxCode.Items[2, 2]
    toFoxCode.Items[1, 1] = '1'
    toFoxCode.Items[1, 2] = 'Description for value 1'
    toFoxCode.Items[2, 1] = '2'
    toFoxCode.Items[2, 2] = 'Description for value 2'
    toFoxCode.ValueType = 'L'
endcase
```

Updating IntelliSense Application

To support FFI, add FFI.VCX and FFI.DBF to FOXCODE.PJX. Also, 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.

Issues

I ran into what looks like a few bugs or design flaws in IntelliSense while testing this:

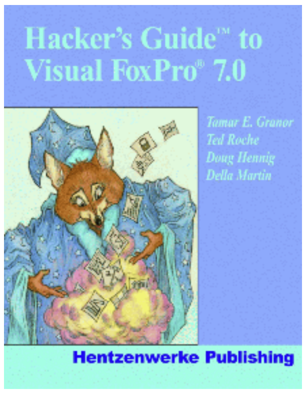
- The list of types that normally appears when you type LOCAL SomeVariable AS doesn't show up at runtime.
- Parentheses completion doesn't work at runtime.
- If you have an object that's a member of another object (such as the User member of the cApplication object in this month's samples, which contains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

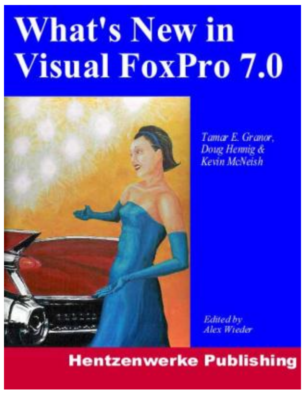
Summary

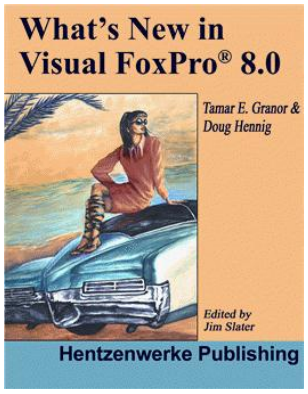
IntelliSense is one of those things we VFP developers can't live without once we started using it. Now, you can give your users the same benefits in your applications. Obviously, this technique isn't for everyone, but if it fits in your application, I guarantee your users will love you for it. Thanks to Randy Brown, former Microsoft Program Manager for Visual FoxPro, for pointing me on the path to figuring out how to make IntelliSense display only the members you want.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro. Doug is co-author of the "What's New in Visual FoxPro" series (the latest being "What's New in Nine") and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). Doug formerly wrote the monthly "Reusable Tools" column in FoxTalk. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is one of the administrators for the VFPX VFP community extensions Web site (http://www.codeplex.com/Wiki/View/aspx?ProjectName=VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996.




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Edited by Alex Wieder
Hentzenwerke Publishing


What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing

stonefield Database TOOLKIT

stonefield Query

stonefield SYSTEMS GROUP INC