

FoxTalk 2.0

Solutions for Microsoft® Visual FoxPro® Developers



Case Study

Does Your Application Understand You?

Dave Bernard

[Icons?](#)

[DOWNLOAD](#)

How cool would it be if you could just *tell* an application the data you needed, and it would actually go out and find it? In this case study, Dave Bernard explains a recent project that combined voice recognition technologies with Microsoft English Query, SQL Server, ASP, and Visual FoxPro.

In the course of my usual networking around town, I came into contact with a person who had a big problem. He had been contracted by a prestigious institute of higher learning to salvage a research project that had been foundering for months. After hearing that the project was a proof-of-concept for a unique art accessibility application, I signed on for the challenge, with little actual experience myself in some of the necessary technology areas, but feeling that it was a solvable set of problems. I was also aware that several of these technologies had recently matured.

And it looked like a whole lot of fun.

The scene: A local museum or gallery






Imagine this scenario: An art patron strolls into a gallery or museum. She's informed that this location supports a wireless network that will enhance her visit, so she pulls out her PDA and browses to the given Web site. As she works her way from gallery to gallery and from work to work, her PDA immediately senses the closest work, retrieves any relevant information, and

Continues on page 4

March 2005

Volume 17, Number 3

- 1 Case Study: Does Your Application Understand You?
Dave Bernard
- 3 Editorial: VFP 9 is Here!
What's Next?
David Stevenson
- 11 Hyperlink Your Reports
Doug Hennig
- 15 Tips from the VFP Team
The Microsoft Visual FoxPro Team
- 16 The Kit Box: Oh Parameters?
No, oParameters
Andy Kramek and Marcia Akins
- 20 March 2005 Downloads

			
Applies to VFP 9.0	Applies to VFP 8.0	Applies to VFP 7.0	Applies to VFP 6.0
	DOWNLOAD		
Applies to VFP 5.0	Accompanying files available online at www.pinnaclepublishing.com		

**FULL PAGE AD:
STONEFIELD**

VFP 9 is Here! What's Next?

David Stevenson

ALMOST two years after the release of VFP 8, Visual FoxPro 9.0 was released to manufacturing in December and is now available through retail channels as well as in MSDN subscriptions! As is usual in the Fox community, the announcement of VFP 9's release was met by a chorus of, "But what about the next version?"

Good grief. Get a grip, people. What comes next? Duh—get VFP 9 and use it!

Let's get a little perspective here. Microsoft's announcements so far have been ambiguous and intentionally vague about what will happen after VFP 9, what any future enhancements to the product would be called, and how future enhancements would be packaged. So what? They'll eventually make some decisions and tell us.

In my opinion, there are far too many people acting like Chicken Little, Henny Penny, Ducky Lucky, Goosey Poosey, and Turkey Lurkey (from the children's story). They spent so much time and energy worrying that the sky was falling that they were almost devoured by a sly Foxy Woxy who deceived them into thinking he could show them a shortcut to the king's palace.

I'm not saying that VFP developers should be blindly loyal to their favorite development tool to the exclusion of everything else. That's not reality for most of us. I *am* saying that we should take what has been provided and use it for productive work. Then, let Microsoft know what you think about it and make your case for what other features you'd like to see them provide, and why.

You'll find a wish list for VFP 10 on the Fox wiki, and you can add your comments and suggestions there and join in the discussion. Just go to <http://fox.wikis.com> and search for "VFP Version 10 Wish List."

Is VFP 9 evolutionary or revolutionary?

Some recent online discussions about VFP 9 have centered on the impact of the many new and enhanced features in this release. Some have argued that this release is simply a minor incremental, evolutionary step, while others view it as a revolutionary release. So, which is it?

It really depends on your perspective, which is dictated to a large degree by the type of development work you're doing, the specific needs of your projects, and your understanding of just what is in the new version.

For example, a developer who does mostly Web work might be unimpressed with the new additions to

BINDEVENT that allow you to hook into Windows events. A developer who uses Crystal Reports might not be impressed with the extensive reworking of the Report Writer. And, those who haven't even paid attention to VFP 9 yet might not realize just how much has changed.

Evolutionary is good

Backward compatibility is always a good thing, and VFP 9 is no exception. You can still run much of the old code and still take advantage of the new features. The VFP team has provided us with several SET commands for choosing which new features to use, such as SET REPORTBEHAVIOR, SET VARCHARMAPPING, and SET ENGINEBEHAVIOR.

You can move your projects ahead with minimum risk and add the features you like as you need them.

Many of the new features are evolutionary in the sense that they add additional features to existing classes or functions. XMLAdapter and CursorAdapter both got additional capabilities to help you fine-tune them to your specific circumstances, such as working with hierarchical XML and controlling when and how CursorAdapter cursors are refreshed.

Revolutionary is good

Some would say that the new "super SQL syntax" is just an evolutionary step, but it really is quite revolutionary when you look closely at performance gains and the ability to issue complex correlated sub-queries.

The new report system is also revolutionary, in my opinion, because of the power it gives us to write our own code in subclasses of ReportListener to augment the new output options. For example, did you know that you can change the default HTML output by substituting your own XSL style sheet for HTMLListener to use? It takes very few lines of code to do that, and you gain complete control over the output.

In fact, you can subclass XMLDisplayListener (just like HTMLListener does) and use XSL to create almost any type of output you want. Seems revolutionary to me.

Use it and talk about it

So, why not give VFP 9 a try today and join the online discussion about it? While we'll continue to provide coverage of previous versions in *FoxTalk 2.0*, you can count on us to provide cutting-edge information and insight into the new features. Stay with us and we'll keep you informed and challenged. ▲

Case Study...

Continued from page 1

tells her about it through headphones. She's prompted to ask questions regarding this particular work, so she speaks a plain-English question to the PDA, which immediately speaks the answer back to her. The system keeps track of how she interacted with the system and where she paused the longest during her visit, and it tunes her experience accordingly.

The goal of this project was to create a proof-of-concept to demonstrate that a stable, inexpensive system could be built to support this kind of application.

Adequate coverage of the voice recognition, text-to-speech, and RFID components is beyond the scope of this article. If you're interested in some of these aspects of the project, you may want to read up on Microsoft Speech Services (MSS), the Speech Application Language Tags protocol (SALT), and speech add-ins available for Microsoft Pocket Internet Explorer.

In this article, I'll focus on the core architectural pieces, involving Visual FoxPro, SQL Server, Microsoft English Query, and Web delivery components. It's a great example of using VFP to "glue" together disparate off-the-shelf technologies to greatly extend an application's capabilities.

Some background

Some may point out that we've had "natural language querying" for some time; after all, millions of us Google the Internet throughout the day for all sorts of information and assistance. When we do, we're presented with hundreds or thousands of "matches" that we cull through in order to satisfy our curiosity. While this capability is certainly very useful, it doesn't come close to approximating human-to-human interaction.

For instance, I may Google information regarding snow skiing techniques, but if I want to purchase an airline ticket to take me to a ski resort, I'd probably use a travel site (or a real person). Performing a search engine lookup on a particular topic is quite different from asking a reservations clerk, for example, "Which flights are available leaving Atlanta and going to Aspen next Wednesday night?" which would result in a discrete, finite answer.

For natural language understanding to occur, the machine must be able to understand the given question, relate it to a stored knowledge base, and then respond in a useful manner. Natural language processing (NLP) must understand grammar (how words connect and how their definitions relate to one another) and how all of that relates to the stored information.

In effect, you're defining a semantic model that describes your data schema. Because of this, you don't

have to know in advance the questions that will be asked; any question within the semantic model can be answered. The question is converted into a SQL query, which is then executed against the data store to return the result(s).

Once you've built something like this, the ramifications become clear. This structure can be laid on top of almost any well-defined data schema, enabling a non-sophisticated user (say, a CEO or CFO) to ask simple, direct questions ("What were our profit margins for the past eight quarters?") without having to make lots of requests of an overworked IT department. This capability can be used instead of building a large number of predefined reports and hoping that they meet user needs.

Think about that for a little while...

Planning the approach

For some time now, I've been building Web-based business systems that use the following general components:

- Browser on the client side (DHTML, JavaScript, CSS) serving up the presentation layer.
- A SQL Server 2000 database for the data store, hosted on a data server somewhere on the Internet.
- IIS 5.0 on Windows 2000 (or IIS 6.0 on Windows 2003) serving a small ASP page that interfaces with a heavy-lifting Visual FoxPro COM+ layer, hosted on a Web server somewhere on the Internet.

As you can see, this is a very component-oriented approach, which makes scaling the application in the future much easier. All of the pieces can reside on a single machine, or each can be deployed to different hardware.

All I needed to add to this architecture was a natural language processing engine that would create a SQL query to return the results I needed.

Microsoft English Query to the rescue!

One technology that I knew about, but had never tried, was Microsoft English Query (MSEQ), which comes with SQL Server and has been around since version 6.5. MSEQ is a component that allows users to query databases using plain English. As I got further into the project, it became apparent that Full-Text Indexing, another SQL Server tool, could be usefully married to MSEQ to handle blob columns (unstructured text, such as an artist's biography) that I couldn't otherwise query with MSEQ.

The work that MSEQ does is complex and sophisticated, but can be described very simply. Once you've defined and set up your data schema in SQL Server, you configure MSEQ to understand the relationships between your tables and columns, and the different ways to which they can be referred semantically.

In our art gallery example, "artists make artworks," "artists give presentations," and "artworks are housed in collections." Once you've set this up, MSEQ can take an English language request like "Which artists gave a

presentation in London?” and create a suitable SQL statement for the application to execute. (MSEQ doesn't actually execute the SQL call, so you'll have to do that separately, using the query it provides for you.)

MSEQ provides wizards to automate a lot of the process of creating the semantic models required for English Query. All that's required is a well-defined data schema, such as our example's SQL Server data schema, but it can also be used against any OLE DB data source, including OLAP data stores.

Once you've developed a model, the MSEQ authoring tool allows you to test it with sample English queries your users might pose. If a test query can't be handled, the Suggestion Wizard helps you to define the relationship manually, and the question will be properly handled in the future.

Once you've got everything ready, you can build a portable English Query Domain file that contains the compiled MSEQ model for use by the runtime engine. You could easily define many completely different MSEQ schemes based on different data schemas and have a single program switch among them at runtime.

So, the proposed architecture (see Figure 1) uses the following set of tools:

- Web/Data server running Microsoft Windows 2003
- Web server: IIS 6.0
- Database: Microsoft SQL Server 2000 SP4
- SQL Server English Query
- SQL Server Full-Text Indexing
- DHTML, JavaScript, VBScript (ASP), CSS
- Microsoft Visual FoxPro 8.0 SP1
- Microsoft Component Services
- Visual Studio

Visual FoxPro in the middle tier

For some time now, Microsoft has positioned VFP as an important tool for building “scalable multi-tier applications that integrate client/server computing and the Internet.” I've very much embraced this mantra over the past several years, and actually do very little work at all with the VFP GUI.

A Web-based application is much easier to deploy and maintain than a desktop-based rich client app because of the advantages you gain when centralizing the application. When I deploy an application or make changes, it all happens in one place and my customers get changes immediately. This saves me a lot of effort and allows me to provide much better customer service at a much lower cost.

The heart of this strategy is a VFP COM+ DLL (COM object). In my case, I've built my own framework class upon which I build custom Web-based applications. Typically, I use few VFP tables, preferring to rely on SQL Server's superior engine, tools, and storage capacities to handle application data needs. VFP is great up to the 2GB barrier, but many applications I deal with easily exceed that. In addition, the SQL syntax supported by SQL Server is much more powerful than VFP, although VFP 9.0 closes that gap considerably.

The best thing about using VFP in the middle tier is that you have the best available development tool to do all the heavy lifting you need, and it does it very quickly and efficiently. Plus, it's the tool you know, which to me is the best reason. Much of my consulting work requires me to get something back to a client very quickly, so I usually don't have six man-months to spend getting up to speed on the Visual Studio .NET IDE.

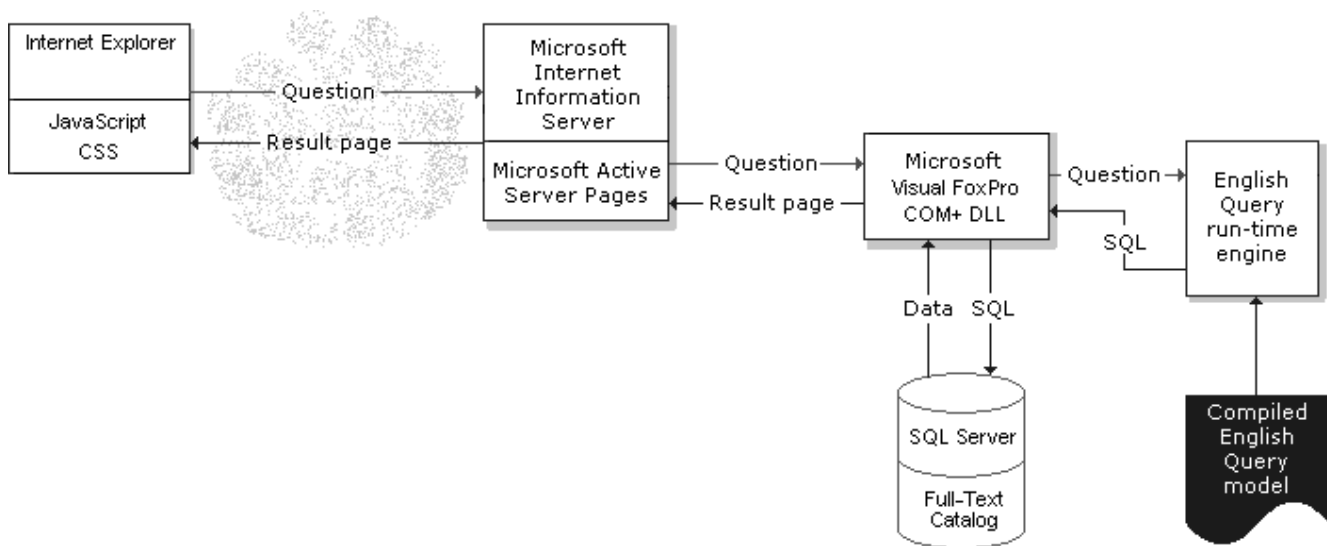


Figure 1. Scalable, multi-tier software application architecture.

Building a VFP COM DLL and interfacing with it is actually very simple, as is getting information from a Web page into a VFP COM object. In this application, the purpose of my VFP COM object is to 1) serve up custom Web pages, 2) interface with MSEQ, and 3) make SQL calls into the SQL Server database.

Using a VFP COM object in a Web application

First, why use VFP to build Web pages? The answer, for me, is speed and flexibility. A Web page is simply a long string of HTML markup; VFP is extremely adept at string handling. A few years ago, I built a VFP COM object that did XSLT transforms, which merge XML data with XSL markup to create a Web page. It was slow and remained slow even after considerable tuning and tweaking. I turned to an old standby, TEXTMERGE, and right out of the box I got a six-fold improvement over my best XSLT efforts! I've never looked back.

Combine the ability to quickly create a Web page with the ability to customize it any way you want because you have a full programming language available, and you have a compelling argument for building Web pages this way.

In order to leverage IIS (whose purpose in life is to serve up Web pages to the Internet or an intranet), I use a thin "classic" ASP layer to take requests from a browser (the user clicks Submit on a form, for example) and push it to the VFP COM object for processing. A criticism of "classic" ASP code is that it's slow because it's interpreted at runtime, which ASP.NET overcomes by being precompiled. Because I use so little ASP code, this issue has no real effect on the system.

Here's some "classic" ASP code that interfaces with a VFP COM object called MyVFPCOM. (I've removed error and security handling for clarity.)

```
<%
' Declare some variables
Dim loApp, lcURLAction, lcHTMLStr

' Get a reference to the VFP COM object
Set loApp = Server.CreateObject("MyVFPCOM.MyVFPCOM")

' Get the name of the method to call from a passed
' query string variable
lcURLAction = Request.QueryString("PRGMethod")

' Build the call to the VFP COM object reference and
' execute it, passing the full Request object to VFP;
' save the return value (an HTML page/string that the
' VFP method creates) to lcHTMLStr
lcHTMLStr = Eval("loApp." & lcURLAction _
    & "(Request)")

' Write the generated Web page back to the browser
Response.Write lcHTMLStr

' Clean up the object reference
Set loApp = Nothing

' End the connection between browser and Web server
Response.End
%>
```

This ASP code lives in a simple text file with an ASP

extension. Note that the Request object is passed in its entirety to the VFP method. The Request object (actually a COM object itself) contains various collections of objects that represent all of the Web page information available to the browser. This includes any form information, URL querystring variables, cookies, and a host of "internal" information about such things as the browser type, screen resolution, and so forth.

If you access the Web site with a URL like MyVFPCOM.asp?PRGMethod=GetAPage, then the ASP code would execute the GetAPage method in the VFP COM object, which in turn would create a response Web page in a string and return it to the ASP code, which in turn sends it back to the browser. The ASP code serves as a pass-through from the client browser to the VFP COM object and back.

It's plain to see that this architecture essentially exposes all of the non-GUI power of VFP to a Web browser! You've gotta love that if you're a VFP developer wanting to build Web applications. And remember: It's FoxPro, so it's fast, too!

Creating and using a VFP COM object

If you haven't done it before, you'll be happy to know that it's simpler than you think to create a VFP COM object. The most important aspects to remember are 1) on which class to base your work and 2) the OLEPUBLIC keyword. The lightweight, non-visual Session class, introduced in VFP 6 SP3, was designed specifically with multi-threaded COM objects in mind. It provides a private datasession without the overhead of a form.

The Session class also hides all intrinsic properties by default, so the type library isn't filled with irrelevant properties. The values for EXCLUSIVE, TALK, and SAFETY are off by default (beginning with VFP 7's version of the Session class), but you need to set other "SET" commands appropriately, since they're all scoped to the private datasession.

```
*****
DEFINE CLASS DemoCOM AS Session OLEPUBLIC
*****

*****
* Initialize the COM environment
*****
PROCEDURE INIT

    SET RESOURCE OFF
    SET DELETED ON

    * Prevent error messages and dialog boxes
    * from interrupting program execution
    SYS(2335, 0)

    * For SQL Server, prevent login dialog boxes
    SQLSetProp(0, "DispLogin", 3)
    SQLSetProp(0, "DispWarnings", .F.)

ENDPROC

*****
* Return some simple HTML
*****
```

PROCEDURE MakeSimpleWebPage

```
RETURN "<B>Here's a simple web page</B>"
```

ENDPROC

ENDDDEFINE

The Init method of the preceding class is very important because it's used for setting up the initial application runtime environment. Remember that a COM object can't have any user interface, so I added the calls to SYS(2335) and SQLSetProp. Init fires every time CREATEOBJECT is called, so keep the code small and efficient!

When running as a COM object under IIS, the DLL is permanently locked in memory for caching purposes, greatly improving performance because the VFP runtime libraries are loaded only once and stay loaded in the IIS process after that. Unfortunately, when replacing a running COM object, several manual steps need to be taken to remove the existing instance and replace it with a new one. I'll talk more about that when we explore Component Services.

Create a project called SimpleCOM, save the preceding code to DemoCOM.PRG, and add it to the project. Then build a multi-threaded DLL. After the DLL is built, you can test it from within VFP:

```
oCOM = CREATEOBJECT("SimpleCOM.DemoCOM")
? oCOM.MakeSimpleWebPage
```

To use this from ASP, create a text file called COMDemo.asp and copy in the following:

```
<%
Dim loApp, lcHTMLStr
Set loApp = Server.CreateObject("SimpleCOM.DemoCOM")
lcHTMLStr = loApp.MakeSimpleWebPage
Response.Write lcHTMLStr
Set loApp = Nothing
Response.End
%>
```

Save the ASP file in your IIS wwwroot folder and make sure the folder has Execute permissions for the IUSR_MachineName account.

IIS security significantly affects COM object deployment. When an ASP page creates a reference to a COM object, that COM object inherits the IIS security context, which is usually the IUSR_MachineName account (or, in some cases, the IWAM_MachineName account). Make sure the IUSR_MachineName or Everyone account has Read and Execute rights on the COM object itself and full rights to any VFP data and runtime libraries that the COM object uses.

Note that the COM object requires the VFP runtime DLLs. On your development machine they're already installed, so nothing explicit needs to be done. If your production server doesn't have VFP installed, copy the VFP runtime DLLs into the same folder as the COM

object, allowing Component Services to find them when needed.

Before you can test this, the COM object must be installed under Component Services (the exact steps may vary slightly, depending on which version of Windows you have).

1. Start up Component Services from Administrative Tools.
2. Open up Component Services to expose COM+ Applications.
3. Click on COM+ Applications to highlight it.
4. Right-click on COM+ Applications and choose New | Application.
5. Click Next, and then Create an empty application.
6. Name it SimpleCOM and click Next, then Next again, then Finish.
7. In the left pane, right-click SimpleCOM and choose Properties.
8. Click on the Security tab, and then "Enforce access checks for this application."
9. Click OK.
10. In the left pane, expand SimpleCOM.
11. Click on Components to highlight it.
12. Right-click on Components and choose New | Component.
13. Click Next, then Install new component(s).
14. Navigate to the folder housing SimpleCOM.DLL, choose SimpleCOM.DLL, and click Open.
15. Click Next, then Finish.

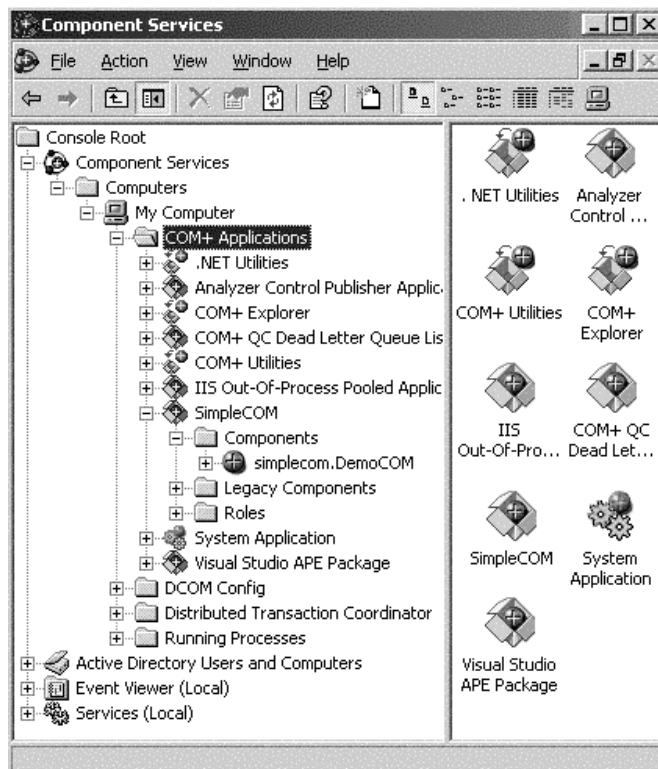


Figure 2. Component Services Manager.

To test the application, open your browser to <http://localhost/COMDemo.asp>. You should see the simple Web page generated by the MakeSimpleWebPage method of the VFP COM object.

More about Component Services

You can view COM as something that enables applications to be accessed as objects. COM+, better known as Component Services, was introduced with Windows 2000 and replaces, enhances, and simplifies the older Microsoft Transaction Server (MTS). COM+ applications are administered through the Component Services Manager, under the Administrative Tools group (see [Figure 2](#)).

COM+ uses role-based security, based on Windows 2000 users and groups. Roles are types of users to which specific Windows users are assigned. To allow our component to be accessed over the Internet, for example, we could create an Internet Users role to which we assign the IUSR_MachineName account. If anyone outside the allowed role attempts to access the component, the attempt is denied and an error is returned to the client.

Updating a VFP COM object

After making changes to a COM object, you can't recompile it if it's already running as an instance under Component Services. Here are the steps to take to compile and deploy your changes:

1. In Component Services, right-click on the COM application and choose Shut Down.
2. Recompile the DLL as before. Move it to the Web server if it's not there by default. The next time an application creates a reference to the COM object, it's automatically "started" by Component Services.

Note that the DLL self-registers on the machine on which it's compiled. Rebuilding the DLL automatically unregisters it first because the registration information is stored in the project file. However, if the project file is deleted and rebuilt, then the older Registry entry won't be removed during the next build.

The DLL build process also creates a Type Library file (TLB) and a Registry Key file (VBR) that must accompany the DLL. The Type Library contains the property and method information for the DLL interface. In short, it lists the entire object model of the COM object.

Testing and debugging

Building a COM object is a pretty simple task, but it's a very different matter to test it. The best bet is to test it as much as possible within VFP before doing system tests using ASP calls. In our particular example, we can't easily simulate the passing of the ASP Request object when testing in the VFP IDE; it can be done only in the actual (ASP) environment. And you can't step through a VFP

COM object in the VFP debugger, either. You can step through the PRG class, but not the compiled COM object.

Using VFP with MSEQ

Now it's time to take this simple architecture and extend it to handle natural language queries. Using our VFP COM object example, an additional method is needed that performs the following:

1. Retrieves a user-entered plain-English query from a Web page.
2. "Massages" the question before sending it to the MSEQ engine.
3. Sends the resulting SQL call to SQL Server and gets back a result set.
4. Formats the result set into a Web page.
5. Returns that Web page to the browser.

```

=====
* Handle English Query Request
=====
PROCEDURE EQResponse(loRequest AS Object) AS Variant

    LOCAL lcReturn, lcRest, lcInquiry, lcChar, i,,
        lcRespGrid, lnConn, loEQResult, lcSQL, loEQ

    WITH THIS
        * Get the question from the browser form variable
        lcInquiry = loRequest.Form("Question").Item()

        * Strip off trailing question mark, if any
        lcInquiry = STRTRAN(lcInquiry, "?", "")

        * Build the response web page
        lcRespGrid = ""

    IF NOT EMPTY(lcInquiry)
        loEQ = CREATEOBJECT("MSEQ.Session")

        IF VARTYPE(loEQ) != "O"
            RETURN "ERROR: MSEQ cannot be initialized"
        ENDIF

        loEQ.InitDomain("EQDemo.eqd")
        loEQResult = loEQ.ParseRequest(lcInquiry)

        loEQ = .NULL.
        RELEASE loEQ

    IF loEQResult.Type = 0
        * Create a connection to the SQL Server database
        lnConn=SQLStringConnect("DRIVER={SQL Server};"+
            "SERVER=MyServer;" +
            "UID=;PWD=;" +
            "DATABASE=MyDatabase")

        * Get an object reference to MSEQ's result
        * collection
        loEQCmd = loEQResult.Commands(0)

        * Retrieve the generated SQL call string
        lcSQL = loEQCmd.SQL

        * Execute that string against our
        * SQL Server connection
        lnTally = SQLEXP( lnConn, lcSQL)

        IF lnTally <= 0
            RETURN "EQ Response query failed"
        ENDIF

        IF RECCOUNT() > 0
            * If the resulting SQL call returned records,
            * format them into a simple Web page.
            SELECT DISTINCT Artist, ArtistID;
            FROM SQLResult;
            ORDER BY Artist;

```

```

        INTO CURSOR q

* Build a list of responses
SCAN
    lcRespGrid = lcRespGrid + ;
    IIF(EMPTY(lcRespGrid), [], [<br/>]) + ;
    ALLTRIM(Artist)
ENDSCAN

* Add a lead-in statement to the response and
* include a restatement of the question as it
* was understood by MSEQ.
lcRespGrid = ;
[I understood you to ask: ] + ;
loEQResult.Restatement + [; ] + ;
[I found ] + ;
ALLTRIM(STR(RECCOUNT())) + ;
[ match ] + ;
IIF(RECCOUNT() = 1, [], [es]) + ;
[, which ] + ;
IIF(RECCOUNT() = 1, [is], [are]) + ;
[ listed below.<br/>] + ;
lcRespGrid

USE IN q
ELSE
    lcRespGrid = "No matches found"
ENDIF

USE IN SQLResult
ELSE
    * Did an error occur?
    IF loEQResult.Type = 2
        lcRespGrid = loEQResult.Description
        lcRespGrid = STRTRAN(lcRespGrid, ;
            [Please capitalize proper names], [])
    ELSE
        * Is a clarification needed?
        IF loEQResult.Type = 3
            lcRespGrid = [Spelling clarification ]+ ;
                [needed for ] + lcInquiry
        ENDIF
    ENDIF
ENDIF
ENDIF
ENDIF
ENDWITH

* Send the results back to the browser
RETURN lcRespGrid

ENDPROC

```

Breaking down the code

The EQResponse method encapsulates the creation of a Web page that contains the result of a SQL call that MSEQ generated from a plain-English query. As noted before, the entire Request object is passed to the EQResponse method from the browser so that browser objects can be accessed. In this case, we're only interested in the textbox on the Web page in which the user entered the plain-English query; it would be specified in HTML like the following:

```
<input type="text" name="Question" size="40" value=""/>
```

Assign the text of the entered query to lcInquiry:

```
lcInquiry = loRequest.Form("Question").Item()
```

Next, get a reference to the MSEQ engine:

```
loEQ = CREATEOBJECT("MSEQ.Session")
```

We have to tell the engine which compiled MSEQ definition to use:

```
loEQ.InitDomain("C:\MyDemo\EQDemo.eqd")
```

Now, pass the plain-English query to MSEQ and store the returned object:

```
loEQResult = loEQ.ParseRequest(lcInquiry)
```

Working with the result object, check the status using the Type property. A status of 0 means that MSEQ returned a Command object, which, for us, means a SQL statement string. A status of 2 means that an error occurred; the error message is stored in the Description property. A status of 3 means that MSEQ wants the user to clarify the query, usually due to a spelling error of some kind.

For Command responses, we go ahead and make a connection to our SQL Server database and execute the MSEQ-generated SQL statement against it:

```

lnConn = SQLStringConnect("DRIVER={SQL Server};" + ;
    "SERVER=MyServer;" + ;
    "UID=;PWD=;" + ;
    "DATABASE=MyDatabase")
loEQCmd = loEQResult.Commands(0)
lcsQL = loEQCmd.SQL

lnTally = SQLEXP( lnConn, lcsQL)

```

Lastly, we format a result page and send that back to the browser.

Running the code

Create a simple Web page to capture the user's query:

```

<form action="SubmitQuestion.asp" method="POST">
  Enter a plain-English query:
  <input type="text" name="Question"
    size="40" value=""/>
  <input type="Submit" name="btnSubmit"
    value="Submit"/>
</form>

```

Here's the ASP code (call it SubmitQuestion.asp) that will be executed when the user hits the Submit button, assuming that the EQResponse method has been added to DemoCOM.PRG:

```

<%
  Dim loApp, lcHTMLStr
  Set loApp = Server.CreateObject("SimpleCOM.DemoCOM")
  lcHTMLStr = loApp.EQResponse(Request)
  Response.Write lcHTMLStr
  Set loApp = Nothing
  Response.End
%>

```

Database design guidelines

MSEQ-based applications work best and are easiest to create with databases that are as normalized as possible and that maximize the use of primary and foreign keys to fully define inter-table relationships keys (see [Figure 3](#) for the schema used in this example).

Planning the English Query model

Before we can make use of the English Query engine, we

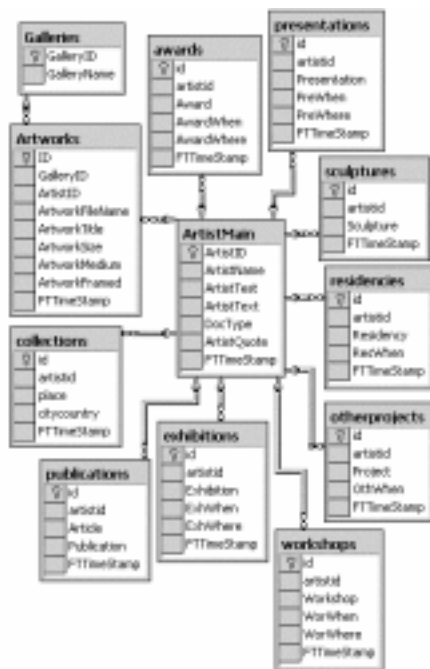
need to define the semantic model for the data store. Here's an outline of the major steps required to develop a very effective English Query model:

1. Think about the questions that users are likely to ask. In my case, their questions could be about artists, their works, their exhibitions, and so forth. Make sure that the semantic model connects these entities together well.
2. Create a model using the English Query Project Wizard, which uses the data schema to automatically create entities and relationships based on the tables, columns, primary keys, foreign keys, and joins.
3. Refine the model to accommodate questions that the wizard didn't handle. You do this by adding necessary entities and relationships that the wizard missed. There's a great deal of flexibility in defining relationships between entities, and it can get quite complex. Proceed slowly and make maximum use of the Suggestion Wizard to test possible questions and tune the model.
4. Build the EQD file for deployment.

Creating the English Query model

1. Go to Start | Programs | Microsoft SQL Server | English Query | Microsoft English Query. This loads Visual Studio and puts you in the New Project dialog under English Query Projects. Make sure the SQL Project Wizard is highlighted and set the name and location as you wish, then click Open.
2. In the Data Link Properties dialog, define a connection to your database.
3. When the wizard presents a list of all the tables found, select everything and click OK.

Figure 3. Database schema.



4. The next dialog presents the standard set of entities and relationships that the wizard gleaned from the schema. You can take it as is or uncheck the items you don't want. Click OK.
5. At this point, the basic model has been built and you can now make changes, test, and tune the model before building the final version.

MSEQ stores the model definition in two files: an English Query Module (with an EQM extension) and an English Query Project (with an EQP extension). Both are XML files and can be edited by any text editor. MSEQ uses these files to build the portable English Query Domain (EQD) file that the COM object uses when interfacing with the English Query engine.

Testing and tuning the English Query model

English Query comes with a very helpful test capability that I've found to be indispensable in making sure that the questions users are likely to ask are supported by the model. For example, I can test the question "Who made a presentation in London?" (see Figure 4) and see detailed responses from the English Query engine. These include the SQL statement that will return the desired answer from the database and the actual database results.

If a question is posed that the engine can't decipher, you can invoke the Suggestion Wizard, which will help you tune the model. Remember that what MSEQ can understand depends strictly on the relationships that exist in the model.

In English Query, a relationship associates one or more entities with each other and defines how they're related. This can be accomplished via simple statements about entities (for example, "painters create artworks") that involve phrasings. Phrasing types include name, adjective, subset, preposition, verb, and trait phrasings.

Continues on page 20

Artist	ArtistID	Quote
BONGI BENGU	114	

Figure 4. Microsoft English Query model test.

Hyperlink Your Reports

Doug Hennig



DOWNLOAD

Last month, Doug Hennig discussed the new ReportListener class in VFP 9 and how it can be used to control report output in ways that previously weren't possible. This month, he looks at how to add live hyperlinks to output generated from reports, allowing some action to be performed when they're clicked.

WOULDN'T it be cool if you could tell VFP to add a hyperlink to a field in a report? Then the user could click on the hyperlink to navigate to some related information. For example, a report showing customers and their Web sites or e-mail addresses would have live links; clicking on a Web site link would navigate the browser to that URL.

Even more interesting would be the ability to navigate somewhere else within your application. For example, clicking on a company name in a report could bring up the customer data entry form with that company as the selected record.

Because the report preview window that ships with VFP doesn't support live, clickable objects in a report, the easiest way to implement this is using HTML, which natively supports hyperlinks.

Hyperlinking reports

VFP comes with a report listener that outputs HTML (the HTMLListener class built into ReportOutput.APP and also included in _ReportListener.VCX in the FFC folder), but I was sure it would require a lot of work to get it to support hyperlinks. However, I was pleasantly surprised to discover how little effort was required.

First, a little background. HTMLListener is a subclass of XMLDisplayListener, which is a subclass of XMLListener, which is a subclass of _ReportListener, the class I discussed last month and recommended you normally use as the parent class for your own listeners. When you use HTMLListener, either directly by instantiating it and using it as the listener for a report or by specifying OBJECT TYPE 5 in the REPORT command, it actually generates XML for the report (this is performed by its parent classes), then applies an XSL transform to the XML to generate the HTML. The XSLT to use is defined in the GetDefaultUserXSLTAsString method.

The default XSLT used by HTMLListener is very complex, and not being much of an XSL expert, I thought it might be an overwhelming task to figure out what to

change to add support for hyperlinks. However, as I started poking through GetDefaultUserXSLTAsString, I discovered the following:

```
<xsl:when test="string-length(@href) > 0">
  <A href="{@href}">
    <xsl:call-template name="replaceText"/>
  </A>
</xsl:when>
```

This XSL adds an anchor tag to the HTML if there's an HREF attribute on the current element in the XML. This is cool—it means HTMLListener already supports hyperlinks! However, searching for "HREF" turned up no hits in XMLDisplayListener or XMLListener, so how can you add that attribute to an element, especially dynamically?

After poking around some more, I found that the attributes of a particular element were set in the GetRawFormattingInfo method of XMLListener. So, I subclassed HTMLListener and added the desired behavior to this method.

The following code, taken from HyperlinkListener.PRG, provides a listener that generates a hyperlink on an object in a report if that object's User memo contains the directive "*:URL =" followed by the expression to use as the URL.

```
define class HyperlinkListener as HTMLListener ;
of home() + 'ffc\_ReportListener.vcx'
QuietMode = .T.
  && default QuietMode to suppress feedback
dimension aRecords[1]
  && an array of information for each record in FRX

* Before we run the report, go through the FRX and
* store information about any field with our expected
* directive in its USER memo into the aRecords array.

function BeforeReport
dodefault()
with This
  .SetFRXDataSession()
dimension .aRecords[reccount()]
scan for atc('*:URL', USER) > 0
  .aRecords[recno()] = ;
    alltrim(strextract(USER, '*:URL =', ;
      chr(13), 1, 3))
endscan for atc('*:URL', USER) > 0
.ResetDataSession()
endwith
endfunc

* If the current field has a directive, add the URL
* to the attributes for the node.

function GetRawFormattingInfo(tnLeft, tnTop, ;
tnWidth, tnHeight, tnObjectContinuationType)
```

```

local lcInfo, ;
  lnURL
with This
  lcInfo = dodefault(tnLeft, tnTop, tnWidth, ;
    tnHeight, tnObjectContinuationType)
  lcURL = .aRecords[recno('FRX')]
  if not empty(lcURL)
    .SetCurrentDataSession()
    lcInfo = lcInfo + ' href="" + ;
      textmerge(lcURL) + ''
    .ResetDataSession()
  endif not empty(lcURL)
endwith
return lcInfo
endfunc
enddefine

```

```

loListener = newobject('HyperlinkListener', ;
  'HyperlinkListener.prg')
loListener.TargetFileName = fullpath('Links.html')
report form Links object loListener
loShell = newobject('_ShellExecute', ;
  home() + 'ffc\_\_Environ.vcx')
loShell.ShellExecute(loListener.TargetFileName)

```

Example 2: Drilldown reports

HyperlinkReports.SCX is a more complex example. As you can see in Figure 2, it presents a list of customer information. However, this HTML is displayed in a Web

The BeforeReport event fires just before the report runs. It uses the SetFRXDataSession method to select the data session the FRX cursor is in, and then scans through the FRX and puts the URL expression for any object that has the directive into an array. It calls ResetDataSession at the end to restore the data session the listener is in.

The GetRawFormattingInfo method uses DODEFAULT() to perform the usual behavior, which generates the attributes for an XML element as a string. It then checks the appropriate array element (the data session for the FRX cursor was selected by code in XMLListener before this code executes) to see whether the current object in the report has the directive, and if so, adds an HREF attribute to the XML element. It calls SetCurrentDataSession to select the data session used by the report's data and uses TEXTMERGE() on the URL expression because the expression will likely contain something specific for each record, such as <<CustomerID>>. Finally, it performs some essential housekeeping by calling .ResetDataSession() to leave the data session as we found it.

That's it! Let's look at some examples of how we can use this listener.

Example 1: Live links to URLs

Links.FRX is a simple example that shows how this listener works. It reports on the Links table, which has a list of company names and their Web sites. The website field in the report has "*:URL = http://<<trim(website)>>" in its User memo. Links.PRG runs this report, using HyperlinkListener as the report listener, and uses the _ShellExecute class in the FFC to display the HTML file in your default browser. Figure 1 shows the results.

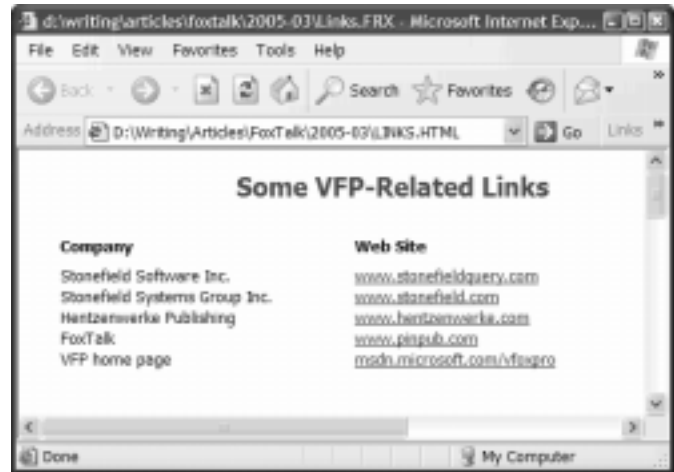


Figure 1. The HTML generated from the Links report has live hyperlinks.



Figure 2. HyperlinkReports.SCX shows a customer report with each customer hyperlinked to the matching orders.



Figure 3. Clicking on a customer's link displays a report of the orders for that customer.

Browser ActiveX control embedded in a VFP form rather than a browser window. When you click on a company name, VFP runs a report of orders for that customer and displays it in the form, as shown in **Figure 3**. The orders report also has a hyperlink that returns the display to the customer list. So, this form provides drilldown reports.

The Init method of the form uses the HyperlinkListener class to generate a hyperlinked HTML file from the HyperlinkCustomers report, and then calls the ShowReport method to display it in the Web Browser control. It also maintains a collection of HTML files generated by the form so they can be deleted when the form is closed.

```
with This

* Create a collection of the HTML files we'll create
* so we can nuke them all when we close.

.oFiles = createobject('Collection')

* Create the customers report.

.oListener = newobject('HyperlinkListener', ;
'HyperlinkListener.prg')
.oListener.TargetFileName = ;
fullpath('HyperlinkCustomers.html')
report form HyperlinkCustomers object .oListener
.oFiles.Add(.oListener.TargetFileName)

* Display it.

.ShowReport()
endwith
```

The ShowReport method simply tells the Web Browser control to load the current HTML file:

```
local lcFile
lcFile = This.oListener.TargetFileName
This.oBrowser.Navigate2(lcFile)
```

Rather than generating order reports for every customer and hyperlinking to them, I decided to generate the reports on demand when a customer name is clicked. To do that, I needed to intercept the hyperlink click. Fortunately, that's easy to do: Simply put code into the BeforeNavigate2 event of the Web Browser control.

To tell BeforeNavigate2 that this isn't a normal hyperlink, I used a convention of "vfps://," which stands for "VFP script," rather than "http://." The code in BeforeNavigate2 looks for this string in the URL and, if found, executes the code in the rest of the URL rather than navigating to it. For example, the User memo of the CompanyName field in HyperlinkCustomers.FRX has the following:

```
*:URL = vfps://Thisform.ShowOrdersForCustomer('<<CustomerID>>')
```

The HyperlinkListener report listener will convert this to an anchor tag such as for the customer with a CustomerID of ALFKI. When you click on this

hyperlink in the Web Browser control, BeforeNavigate2 fires and the code in that event strips off the "vfps://" part and executes the rest. It also sets the Cancel parameter, passed by reference, to .T. to indicate that the normal navigation shouldn't take place (similar to using NODEFAULT in a VFP method). Here's the code for BeforeNavigate2:

```
LPARAMETERS pdisp, url, flags, targetframeName, ;
postdata, headers, cancel
local lcMethod
if url = 'vfps://'
lcMethod = substr(url, 8)
lcMethod = left(lcMethod, len(lcMethod) - 1)
&& strip trailing /
&lcMethod
cancel = .T.
endif url = 'vfps://'
```

The ShowOrdersForCustomer method, executed when you click on a company name, runs the HyperlinkOrders report for the specified customer, displays it in the Web Browser control, and adds the file name to the collection of files to be deleted when the form is closed.

```
lparameters tcCustomerID
with This
.oListener.TargetFileName = ;
fullpath(tcCustomerID + '.html')
report form HyperlinkOrders object .oListener ;
for Orders.CustomerID = tcCustomerID
.ShowReport()
.oFiles.Add(.oListener.TargetFileName)
endwith
```

The CustomerName field in the HyperlinkOrders report has "*:URL = vfps://Thisform.ShowCustomers()" in its User memo, so clicking on this hyperlink in the report redisplay the customer list.

Example 3: Launching a VFP form

CustomerReport.SCX is similar to HyperlinkReports.SCX, but is a little simpler. It also hosts a Web Browser control that displays the HTML from a report, EditCustomers.FRX, which looks the same as the previous example. However, clicking on a customer name in this form displays a maintenance form for the selected customer.

EditCustomers.FRX is a clone of the HyperlinkCustomers report used in the previous example, but has "*:URL = vfps://Thisform.EditCustomer('<<CustomerID>>')" in the User memo of the CompanyName field instead. The form's EditCustomer method, called from BeforeNavigate2 when a customer name is clicked, launches the Customers form, passing it the CustomerID for the selected customer. The Customers form is a simple maintenance form for the Customers table, with controls bound to each field and Save and Cancel buttons.

NavPaneListener

MVP Fabio Vazquez has created another kind of listener

that has hyperlinks, albeit for a completely different purpose. His NavPaneListener, available for download from <http://ReportListener.com>, provides an HTML report previewer with a table of contents for the report. As you can see in **Figure 4**, a thumbnail image of each page is shown at the left and the current page is shown at the right. Clicking on a thumbnail navigates to the appropriate page.

Like HyperlinkListener, NavPaneListener is quite simple. Its OutputPage event, called as each page is to be output, simply generates a GIF file for the page by calling itself again with the appropriate parameters. tnDeviceType is initially -1, meaning no output, because the listener's ListenerType property is set to 2. In that case, OutputPage calls itself, passing the name and path for a GIF to generate (the cPath property defaults to the current directory) and a device type value that indicates a GIF file. On the second call, in addition to the normal behavior (generating the specified GIF file), OutputPage passes the name and path to the AddPage method of a collaborating object stored in the oNavigator property. (Note: I translated some of Fabio's code into English to make it more readable.)

```

procedure OutputPage(tnPageNo, teDevice, ;
tnDeviceType)
  local lnDeviceType
  with This
  do case
    case tnDeviceType = -1  && None
      lnDeviceType = 103  && GIF
      .OutputPage(tnPageNo, .cPath + 'Page' + ;
        transform(tnPageNo) + '.gif', lnDeviceType)
    case tnDeviceType = 103
      .oNavigator.AddPage(teDevice)
  endcase
endwith
endproc

```

After the report is done, the navigator object creates a couple of HTML documents—one that defines a frameset with the table of contents in the left frame and the contents to display in the right frame, and one that contains the table of contents as thumbnails of the GIF files hyperlinked to display the full-size GIF file in the content frame. The navigator object then automates Internet Explorer to display the frameset document.

Summary

Notice that none of the code in any of these examples is complicated, nor is there much of it. As a result, it takes only a few moments to implement reports with live

hyperlinks, drilldown reports, reports that launch some VFP form or other action, or reports with navigation panes. This truly shows the power of report listeners!

Next month, we'll look at a similar topic—report previews that perform some action when clicked—but using an entirely different technique that will give us abilities such as text search and bookmarks. ▲

DOWNLOAD 503HENNIG.ZIP at www.pinnaclepublishing.com

Doug Hennig is a partner with Stonefield Systems Group Inc. He's 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's a 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's a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. www.stonefield.com, www.stonefieldquery.com, dhennig@stonefield.com.

Know a clever shortcut?
 Have an idea for an article for *FoxTalk 2.0*?
 Visit www.pinnaclepublishing.com and click on
 "Write For Us" to submit your ideas.

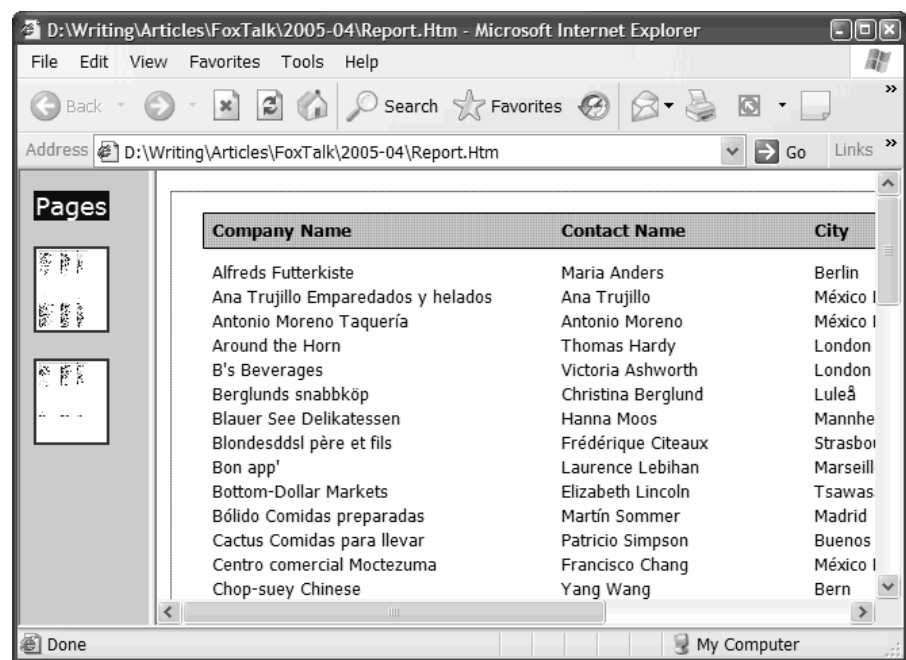


Figure 4. Fabio Vazquez's NavPaneListener creates an HTML report previewer with a table of contents.

Tips from the VFP Team

The Microsoft Visual FoxPro Team



This month's "Tips from the VFP Team" column continues the focus on new features and speed improvements in VFP 9's SQL engine. VFP 9 was released to manufacturing in December 2004 and is available now for download by MSDN subscribers. It should be available through retail outlets by the time you read this article.

Update records faster with VFP 9's Correlated Update

The following code demonstrates the tremendous speed improvements of the new Correlated Update capability, when compared to the procedural style of coding necessary in previous versions. *Note:* All of the code for all three samples is available in the accompanying download file.

```
* Demonstrate correlated UPDATE (UPDATE ... FROM ..)
* UPDATE unitprice of products from mfg_msrp with 10%
* off MSRP
```

```
CLOSE DATABASES ALL
SET MULTLOCKS ON
OPEN DATABASE Northwind
CLEAR
```

```
? "Procedural REPLACE FOR"
lnStart = SECONDS()
USE Products IN 0
CURSORSETPROP("Buffering",5,"Products")
USE mfg_msrp IN 0
SELECT mfg_msrp
SCAN
  SELECT Products
  REPLACE ALL unitprice WITH mfg_msrp.msrp*.90 ;
  FOR Products.ProductID =
```

```
mfg_msrp.ProductID
  SELECT mfg_msrp
ENDSCAN
? SECONDS() - m.lnStart
```

```
SELECT Products
=TABLEREVERT(.T.)
CLOSE TABLES ALL
```

```
? "Correlated UPDATE"
```

```
lnStart = SECONDS()
USE Products IN 0
CURSORSETPROP("Buffering",5,"Products")
UPDATE products ;
  SET unitprice = mfg_msrp.msrp*.90 ;
  FROM mfg_msrp ;
  WHERE mfg_msrp.productID = products.productID;
  AND mfg_msrp.discontinu = .F.
```

```
? SECONDS() - m.lnStart
SELECT Products
=TABLEREVERT(.T.)
CLOSE DATABASES ALL
```

Use sub-selects in the SQL FROM clause with VFP 9's Derived Table feature

The following code sample demonstrates VFP 9's new SQL Derived Table feature:

```
* Derived table (Sub-select as part of FROM)
* Returns customer ID and product count for all custs
* who have purchased at least 40% of the product line.
CLOSE DATABASES ALL
OPEN DATABASE Northwind
LOCAL lnPct
lnPct = .40
SELECT ;
  C.customerid, ;
  C.companyname, ;
  P.p_count AS Product_Count;
FROM Customers C, ;
  (SELECT c2.customerid, ;
  COUNT(DISTINCT D.productID) AS p_count ;
  FROM Customers C2 ;
  INNER JOIN Orders O ;
  ON C2.customerid = O.customerid ;
  INNER JOIN OrderDetails D ;
  ON O.orderid = D.orderid ;
  GROUP BY c2.customerid) AS P ;
WHERE C.customerID = p.customerID ;
  AND P.p_count >= ;
  (SELECT (COUNT(*)*m.lnPct) FROM Products ;
  WHERE NOT EMPTY(ProductName)) ;
ORDER BY p.p_count DESC

CLOSE DATABASES ALL
```

Use sub-selects in the SQL field list with VFP 9's Projection feature

The following code sample demonstrates VFP 9's new SQL Projection feature:

```
* Projection (Sub-select as part of field list and
* complex expression)
* Sales total by customer with PCT of All cust sales
CLOSE DATABASES ALL
OPEN DATABASE Northwind
SELECT ;
  C.customerID, ;
  C.companyname, ;
  SUM(D.quantity*D.unitprice) AS CustTotal, ;
  (SUM(D.quantity*D.unitprice) / ;
  (SELECT SUM((quantity*unitprice)-discount) ;
  FROM OrderDetails D2) ;
  )*100 AS PctTotal ;
FROM Customers C ;
INNER JOIN Orders O ;
  ON C.customerID = O.customerID ;
INNER JOIN OrderDetails D ;
  ON O.orderid = D.orderid ;
GROUP BY C.customerID, C.companyname, O.orderID ;
ORDER BY pctTotal DESC

CLOSE DATABASES ALL
```



503TEAMTIPS.ZIP at www.pinnaclepublishing.com

Line too long, how should it break?

Oh Parameters? No, oParameters

Andy Kramek and Marcia Akins

Icons?

DOWNLOAD

As developers, we all rely heavily on passing parameters between objects to manipulate behavior or report results. However, when we need to pass more than a few parameters, the process rapidly becomes unwieldy because, in Visual FoxPro, parameters must be passed in a specific sequence. Also, we can return only a single value from a function or method, so passing back multiple values gets tricky. In this month's column, Andy Kramek and Marcia Akins discuss how using a parameter object can simplify your life.

Andy: Doesn't this just drive you crazy?

Marcia: What?

Andy: Trying to remember the correct sequence for passing parameters to functions. I've recently been working on an old FoxPro 2.6 system that uses lots of (very cryptic) function calls to handle routine tasks, and trying to remember the correct parameter sequence is a real problem. For example, in order to display the dialog box in [Figure 1](#), the necessary function call is:

```
ln_resp=f0getdl("Continue anyway",1,1,0,.F.,0,"",;
"2","Yes","No","","','','Errors Found' )
```

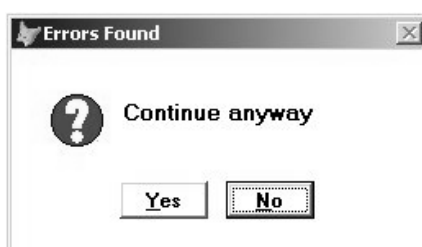
and the actual function contains nearly 500 lines of code.

Marcia: First, that was an old application. Second, the size of the function has nothing to do with anything, and third, you don't [have write](#) functions to show a simple dialog anymore. Just use the `MessageBox()` function.

Andy: Yes, I know that. But it got me thinking about parameters in general. How many parameters should you reasonably expect to use? In fact, how many *could* you use?

Marcia: Well, the VFP system capacities table in the Help file lists 27 as the maximum number of parameters that may be passed in Versions 6.0 and 7.0 but, interestingly,

Figure 1. A simple Yes/No dialog.



only 26 for Versions 8.0 and 9.0.

Andy: And even that is apparently wrong. The following code works perfectly well in Versions 6.0, 7.0, 8.0, and 9.0. In fact, it even compiles without error in FoxPro 2.6, although you get an “insufficient stack space” error when you try to run it. (Notice that even there, it's not a “too many parameters” error.)

```
DO ParmTest WITH 'Test1','Test2','Test3','Test4',;
'Test5','Test6','Test7','Test8','Test9','Test10',;
'Test11','Test12','Test13','Test14','Test15','Test16',;
'Test17','Test18','Test19','Test20','Test21','Test22',;
'Test23','Test24','Test25','Test26','Test27','Test28',;
'Test29','Test30','Test31','Test32','Test33','Test34',;
'Test35','Test36','Test37','Test38','Test39','Test40',;
'Test41','Test42','Test43','Test44','Test45','Test46',;
'Test47','Test48','Test49','Test50','Test51','Test52',;
'Test53','Test54','Test55','Test56','Test57'
FUNCTION ParmTest
LPARAMETERS p01,p02,p03,p04,p05,p06,p07,p08,p09,p10,;
p11,p12,p13,p14,p15,p16,p17,p18,p19,p20,p21,p22,p23,;
p24,p25,p26,p27,p28,p29,p30,p31,p32,p33,p34,p35,p36,;
p37,p38,p39,p40,p41,p42,p43,p44,p45,p46,p47,p48,p49,;
p50,p51,p52,p53,p54,p55,p56,p57
*** How many parameters?
FOR lnCnt = 1 TO PCOUNT()
  lcVal = 'p'+ PADL( lnCnt,2,'0')
  ? &lcVal
NEXT
RETURN
```

Marcia: That's odd! Why did you stop at 57 parameters? Is that the real limit?

Andy: No, I just got bored typing them in. The same code works in forms too, so it appears that, despite what the Help file says, we have the potential to pass more parameters than we could ever reasonably use. But, that doesn't answer the question, “How many should you use?”

Marcia: Personally, I've always preferred to use named parameters.

Andy: But VFP doesn't support that.

Marcia: Not directly. However, if you use a parameter object, you really do get named parameters. By adopting this approach we answer the “how many parameters” question, too.

Andy: How so?

Marcia: Because we can encapsulate any number of

Should this be [have to write](#)?

parameters in the object, and so we only ever need to pass one!

Andy: Ah, yes! I can see another benefit here, too. Since we only need one object, we could use it in either direction.

Marcia: What do you mean by that?

Andy: Well, since we can only pass one result back from a function or method, we could make that one return value an object and use it to return multiple results.

Marcia: Oh yes, indeed we can. In fact, returning a “results” object is the easiest way to get around the limitation of one result per function or method. But let’s concentrate for the moment on passing parameters into a method or function. This is, after all, the most common scenario. First, we need to create an object to which we can add our parameter names as properties. Once we have that, we can simply assign values to those properties.

Andy: The “empty” base class that was exposed in Version 8.0 is the obvious choice for creating parameter objects; it has no properties or methods of its own and therefore is extremely lightweight.

Marcia: That would be my first choice, too. Of course, to actually add the properties, we have to use the `AddProperty()` function because there’s no `AddProperty()` method (the empty base class really is empty!). So the basic code we need looks like this:

```
oParams = CREATEOBJECT( 'empty' )
IF VARTYPE( oParams ) = "O"
  ADDPROPERTY( oParams, <property name>, ;
    <property value >
  *** Repeat ADDPROPERTY() calls as needed ***
ENDIF
```

Andy: Of course, there’s an implicit assumption that the names of the parameters that are expected are known when you’re creating the parameter object.

Marcia: It’s more than an assumption—it’s required if you’re going to use named parameters. However, in that scenario the names of the parameters are part of the published interface anyway, so it’s a non-issue.

Andy: I wouldn’t say that. After all, when using sequential parameters it doesn’t matter what names are used in the source code, because the only thing that counts is the order in which passed values are received. The names are set locally and there’s no direct connection between the calling code and the called. However, I suppose we can always use `PEMSTATUS()` to find out if an expected property is actually there:

```
IF PEMSTATUS( oParams, 'cMyExpectedParam', 5 )
```

Marcia: Yes, of course. You could also use more generic code to determine what parameters have actually been passed in. For instance, you can use the `AMEMBERS()` function to populate an array with the names of all the properties on the object, like this:

```
lnProps = AMEMBERS( laPropertyNames, oParams, 0 )
FOR lnCnt = 1 TO lnProps
  *** Get the parameter name
  lcVarName = laPropertyNames[ lnCnt ]
  *** Declare as local variable with the same name
  LOCAL &lcVarname
  &lcVarname = EVAL( "oParams." + lcVarName )
NEXT
```

After all, since the empty base class has no native properties, the only things that will be in this array are the passed in parameters.

Andy: That’s very neat. But it assumes that we always want to transfer parameters to local variables. Often (and especially when passing parameters to the `Init()` of a form), we really want to make them properties. If we adopted a naming convention for the parameters, we could handle different types. For example, parameters named “p_xxxx” could be created as properties on the receiving object, while “v_xxxx” would be handled as variables. We could modify the code like this:

```
lnProps = AMEMBERS( laPropertyNames, oParams, 0 )
FOR lnCnt = 1 TO lnProps
  *** Get the parameter name
  lcVarName = laPropertyNames[ lnCnt ]
  luPropVal = EVAL( "oParams." + lcVarName )
  *** Do we want a variable, or a property
  IF LOWER( LEFT( lcVarname, 2 )) == 'p_'
    *** This is a property
    This.AddProperty( SUBSTR( lcVarname, 3 ), luPropVal )
  ELSE
    *** Declare as local variable with the same name
    LOCAL &lcVarname
    &lcVarname = luPropVal
  ENDIF
NEXT
```

Marcia: You know what? Now that I see it, we could make this even simpler. Why don’t we use an object based on the *collection* base class instead of *empty*?

Andy: How is that going to simplify things?

Marcia: The *collection* base class already has methods to add and retrieve items, and a count property to tell us how many items it has. We can use the name of the parameter as the key, and the value as the item. Now, the code to create a parameter object is:

```
oParams = CREATEOBJECT( 'collection' )
IF VARTYPE( oParams ) = "O"
  oParams.Add( <property value >, <property name> )
  *** Repeat ADD() calls as needed ***
ENDIF
```

Andy: Looks pretty much the same to me.

Marcia: I agree that there's no real difference on the creation side of it, but unpacking the object is simpler. Look:

```
FOR lnCnt = 1 TO oParams.Count
  lcVarName = oParams.GetKey( lnCnt )
  luPropVal = oParams.Item( lnCnt )
  *** Do we want a variable, or a property
  IF LOWER( LEFT( lcVarName, 2 ) ) == 'p_'
    *** This is a property
    This.AddProperty( SUBSTR( lcVarName, 3 ), luPropVal )
  ELSE
    *** Declare as local variable with the same name
    LOCAL &lcVarName
    &lcVarName = luPropVal
  ENDIF
NEXT
```

Andy: Ah, I see. You avoid the need to use AMEMBERS() and you don't need to use EVAL() to get the passed value. However, there's one problem. What happens if you want to pass an array as a parameter? You can't add an array directly to a collection. You'd have to convert the array into a collection and then add that as an item.

Marcia: I hadn't thought of that. How would you even add an array as a property to the empty object?

Andy: Just use the ACOPY() function:

```
DIMENSION laFruit[3]
laFruit[1] = 'Apples'
laFruit[2] = 'Oranges'
laFruit[3] = 'Grapes'
oParams = CREATEOBJECT( 'empty' )
AddProperty( oParams, 'a_fruit[1]' )
ACOPY( laFruit, oParams.a_fruit )
```

Marcia: Okay, I didn't know you could do that. I see you're defining the array property with an "a_" prefix. How are you going to unpack it? To a variable, or to a property?

Andy: I'm not sure. I suppose we could extend the convention to include the prefix "a_" for array variables and "z_" for array properties. However, we could also leave the decision to the actual implementation.

Marcia: Hang on there. I was assuming that this code would be truly generic. In other words, we would have an unPackParameters() method that we could call whenever we need to handle a parameter object. You're talking as though the code will be embedded directly into the method that receives the object.

Andy: Well, yes. After all, if you're going to unpack the values to local variables, there wouldn't be much point in doing it in a subordinate method. The calling method could never get the values.

Marcia: I must have taken my stupid pills by mistake this morning! I hadn't thought of that. When viewed in that light, there are actually two different scenarios:

- The first is when we simply want to use named parameters. In this case, there's no need to use generic code because the receiving method knows the names of the parameters and can just read them off the object and deal with them appropriately.
- The second is when an unknown number of parameters, or parameters whose names might vary at runtime, are involved (for example, column names from one of a number of possible tables). In this case, we use the generic code to unpack the parameter object and always create properties for whatever is found. Then it's up to the actual methods to utilize those properties as needed.

Andy: That makes sense. It also avoids the reliance on a naming convention to decide how to handle things differently. However, the generic unpacking code will have to include a check to see whether a specific property already exists, but that's a very small overhead given the flexibility that it buys us.

Marcia: We'd still need to differentiate between arrays and other properties. Fortunately, VFP 9.0 has enhanced the TYPE() function to return a value of "A" when the reference being tested is an array, and "C" if it's a collection. All you need to do is pass the second parameter as 1, like this:

```
lcVarName = laPropertyNames[ lnCnt ]
lcType = TYPE( lcVarName, 1 )
DO CASE
  CASE lcType = 'A'
    *** its an array
  CASE lcType = 'C'
    *** It's a collection
  OTHERWISE
    *** Simple property
ENDCASE
```

Andy: That's very good, because I was just beginning to wonder how we should handle the case when a collection was passed in a parameter object. Though, on reflection, the handling doesn't change—it's still a single property. But it's good to know that we can test whether a property is a collection. So can we put all the code together?

Marcia: Yes, I think so. I suggest that this be implemented as a global function so that we don't need to add the method to individual base classes.

Andy: Or, you could just add it to an application object class.

Marcia: Either way would work, but in any case, you still have to pass two object references to the function—

one that refers to the object that's going to receive the parameters, and one that refers to the parameter object itself. The version of the code as a standalone function is included in the download for this column, but here's the complete code:

```

LPARAMETERS toSource, toParams
LOCAL ARRAY laPropertyNames[1]
LOCAL lnProps, lnCnt, lcVarName, lcType

*** We require TWO objects here!
IF VARTYPE( toSource ) # "O"
  ASSERT .F. MESSAGE ;
  "Must pass an object reference to UNPACKPARAMOBJ"
ENDIF
IF VARTYPE( toParams ) # "O"
  ASSERT .F. MESSAGE ;
  "Must pass a parameter object to UNPACKPARAMOBJ"
ENDIF

*** First get the names of all parameters on the object
lnProps = AMEMBERS( laPropertyNames, toParams, 0 )

*** Now process the array
FOR lnCnt = 1 TO lnProps
  *** Get the parameter name
  lcVarName = laPropertyNames[ lnCnt ]

  *** What type of parameter have we got here?
  lcType = TYPE( "toParams." + lcVarName, 1 )

  *** See if the property exists
  IF NOT PEMSTATUS( toSource, lcVarName, 5 )
    *** Create the property, based on the type
    IF lcType = 'A'
      *** its an array
      ADDPROPERTY( toSource, lcVarName + "[1]" )
    ELSE
      *** It's either a collection or simple property
      ADDPROPERTY( toSource, lcVarName )
    ENDIF
  ENDIF

  *** Now populate the property

```

```

IF lcType = 'A'
  *** This is the array
  ACOPY( toParams.&lcVarName, toSource.&lcVarName )
ELSE
  *** A simple property or collection
  toSource.&lcVarName = toParams.&lcVarName
ENDIF
NEXT
*** No Return value other than the usual .T.
RETURN

```

Andy: Not much! Just one question: Why are we using macro substitution instead of EVAL() to populate the properties?

Marcia: Because in the ACOPY() function, EVAL() doesn't work (you can't use a name expression when referencing an array because you get only the first element). So, we have to use macro substitution—it's just one of those situations where there's no alternative. ▲

 [503KITBOX.ZIP at www.pinnaclepublishing.com](http://www.pinnaclepublishing.com)

Andy Kramek is a long-time FoxPro developer, FoxPro MVP, independent consultant, and joint owner of Tightline Computers Inc., based in Akron, OH. A veteran conference speaker, he has published widely and can be found online in the CompuServe forums (<http://go.compuserve.com/msdevapps>), Foxite (www.foxite.com), and the Virtual FoxPro Users Group (www.vfug.org). andykr@tightlinecomputers.com.

Marcia Akins is a FoxPro MVP, independent consultant, and joint owner of Tightline Computers Inc., based in Akron, OH. A veteran conference speaker, she has published widely and is well known for her contributions to Tek-Tips (www.tek-tips.com) and the Universal Thread (www.Universalthread.com). marcia@tightlinecomputers.com.

Don't miss another issue! Subscribe now and save!

Subscribe to *FoxTalk 2.0* today and receive a special one-year introductory rate:
Just \$129* for 12 issues (that's \$30 off the regular rate)

NAME _____

COMPANY _____

ADDRESS _____

CITY _____ STATE/PROVINCE _____ ZIP/POSTAL CODE _____

COUNTRY IF OTHER THAN U.S. _____

E-MAIL _____

PHONE (IN CASE WE HAVE A QUESTION ABOUT YOUR ORDER) _____

- Check enclosed (payable to Pinnacle Publishing)
- Purchase order (in U.S. and Canada only); mail or fax copy
- Bill me later
- Credit card: VISA MasterCard American Express

CARD NUMBER _____ EXP. DATE _____

SIGNATURE (REQUIRED FOR CARD ORDERS) _____

Detach and return to:
Pinnacle Publishing ▲ 316 N. Michigan Ave. ▲ Chicago, IL 60601
Or fax to 312-960-4106

* Outside the U.S. add \$30. Orders payable in U.S. funds drawn on a U.S. or Canadian bank.

INS5

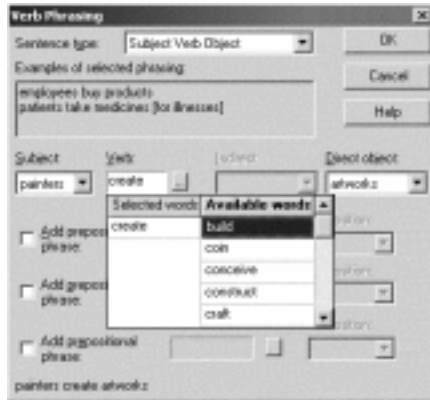
Pinnacle, A Division of Lawrence Ragan Communications, Inc. ▲ 800-493-4867 x.4209 or 312-960-4100 ▲ Fax 312-960-4106

Case Study...

Continued from page 10

For example, “painters create artworks” is an example of a verb phrasing (see Figure 5). Using different phrasings is a good way to expand a model. A full explanation of phrasing capabilities is beyond the scope

Figure 5. A verb phrasing.



of this article, but there’s a mind-boggling array of complex relationships that can be built into a model.

After you’re satisfied with the performance of the model, build it into a compiled English Query application (EQD file).

And beyond...

I’ve literally scratched only the very surface of what can be done with English Query. I hope that I’ve piqued your interest sufficiently for you to investigate it further. ▲

DOWNLOAD

[503BERNARD.ZIP at www.pinnaclepublishing.com](http://www.pinnaclepublishing.com)

Dave Bernard, co-founder of The Intellection Group, Inc., has been a software developer, manager, and executive for more than 25 years. He has worked in FoxPro continuously from version 1.02 through VFP 9, is vice president of the Atlanta FoxPro Users Group, and holds MCSD and MCDBA certifications from Microsoft. He currently specializes in Internet-based extranet development using VFP and SQL Server and has recently been immersed in voice recognition, natural language, and RFID projects. dbernard@intellectiongroup.com.

March 2005 Downloads

- **503BERNARD.ZIP**—Source code to accompany Dave Bernard’s article, “Case Study: Does Your Application Understand You?”
- **503HENNIG.ZIP**—Source code to accompany Doug Hennig’s article, “Hyperlink Your Reports.”
- **503TEAMTIPS.ZIP**—Source code to accompany this month’s “Tips from the VFP Team” column.
- **503KITBOX.ZIP**—Source code to accompany Andy Kramek and Marcia Akins’ article, “The Kit Box: Oh Parameters? No, oParameters.”

For access to current and archive content and source code, log in at www.pinnaclepublishing.com.

Editor: David Stevenson (david@topstrategies.com)
CEO & Publisher: Mark Ragan
Group Publisher: Michael King
Executive Editor: Farion Grove

Questions?

Customer Service:
Phone: 800-493-4867 x.4209 or 312-960-4100
Fax: 312-960-4106
Email: PinPub@Ragan.com

Advertising: RogerS@Ragan.com

Editorial: FarionG@Ragan.com

Pinnacle Web Site: www.pinnaclepublishing.com

Subscription rates

United States: One year (12 issues): \$159; two years (24 issues): \$278
Other:* One year: \$189; two years: \$338

Single issue rate:

\$20 (\$25 outside the United States)*

* Funds must be in U.S. currency.

FoxTalk 2.0 (ISSN 1042-6302)
is published monthly (12 times per year) by:

Pinnacle Publishing
A Division of Lawrence Ragan Communications, Inc.
316 N. Michigan Ave., Suite 300
Chicago, IL 60601

POSTMASTER: Send address changes to Lawrence Ragan Communications, Inc., 316 N. Michigan Ave., Suite 300, Chicago, IL 60601.

Copyright © 2005 by Lawrence Ragan Communications, Inc. All rights reserved. No part of this periodical may be used or reproduced in any fashion whatsoever (except in the case of brief quotations embodied in critical articles and reviews) without the prior written consent of Lawrence Ragan Communications, Inc. Printed in the United States of America.

Brand and product names are trademarks or registered trademarks of their respective holders. Microsoft is a registered trademark of Microsoft Corporation. The Fox Head logo, FoxBASE+, FoxPro, and Visual FoxPro are registered trademarks of Microsoft Corporation. *FoxTalk 2.0* is an independent publication not affiliated with Microsoft Corporation. Microsoft Corporation is not responsible in any way for the editorial policy or other contents of the publication.

This publication is intended as a general guide. It covers a highly technical and complex subject and should not be used for making decisions concerning specific products or applications. This publication is sold as is, without warranty of any kind, either express or implied, respecting the contents of this publication, including but not limited to implied warranties for the publication, performance, quality, merchantability, or fitness for any particular purpose. Lawrence Ragan Communications, Inc., shall not be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this publication. Articles published in *FoxTalk 2.0* reflect the views of their authors; they may or may not reflect the view of Lawrence Ragan Communications, Inc. Inclusion of advertising inserts does not constitute an endorsement by Lawrence Ragan Communications, Inc., or *FoxTalk 2.0*.