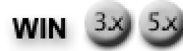
February 1997

Set Turbo On: How Visual FoxPro Memory Usage Affects Performance

Flavio Almeida and Walter Loughney 

The default installation of Visual FoxPro leaves something to be desired as far as efficient use of memory, which drastically affects performance. In some cases, FoxPro for Windows 2.6 can run certain queries many times faster than Visual FoxPro. How can this be? In this article, Flavio and Walter investigate the reasons behind this strange behavior and show you how to optimize your installation to get the best performance possible out of Visual FoxPro.

Over many months of working with VFP 3.0 and 5.0, we've come across a number of performance issues. Fox's performance seemed to vary wildly for no apparent reason, and thus warranted our further investigation. Upon doing so, we decided to share the results with the rest of the developer community. The startling results of our testing showed that Visual FoxPro would, in some cases, run queries many times slower than its older sibling, FoxPro for Windows 2.6. Here's what we found and what you can do about it. The good news is that, intrinsically, VFP is as fast as or faster than FPW2.6a in almost every area and isn't noticeably slower in any area. The bad news is that you're probably not getting that kind of performance.

Once we determined that we had to perform a series of tests to measure performance under various conditions, we had to figure out a clean way of testing so that we were always comparing apples to apples.

The first problem was determining whether these issues were due to some aspect of FoxPro or some peculiarity of the application or the system. Each of us has our own style of developing, our company or clients have different data structures and processing needs, the use of add-on tools and libraries may affect what we do, and of course different networks and hardware have a dramatic effect on performance. Choosing any one system or test as a base for measuring performance is sure to be invalid in some cases. With that in mind, we tested the performance of VFP on different systems and platforms, and its ability to perform a variety of tasks.

Before we show you how to speed up VFP, we need to set up a test so you can optimize it for your own system. Because most of what VFP does is table based (including forms, reports, menus, data, and so on), anything that speeds up table handling should speed up VFP. We want a table that's large enough to perform a test without being so large as to be impractical for testing. Using a table that's too small produces results in fractions of a second; these results are hard to interpret and make it difficult to gather meaningful data.

We chose to use the customer table from FoxPro for Windows 2.6, a table available to most FoxPro developers. Its default location is \FPW\TUTORIAL\CUSTOMER.DBF. The reason we

chose an FPW table is to provide an easy means of testing VFP against FPW.

This table has only 500 records, a number which isn't sufficient for testing, so we had to make it a little larger. You can run the following program just under VFP, but other tests we do later may change the table structure. So, if you wish to do your own comparisons, you should make a copy of the table and run this process for both FPW and VFP. If you want to use our results and just optimize your VFP system, you can run the following program under VFP alone:

```
* MAKETEST.PRG
* This makes a table for testing VFP performance.
* Change it as needed for your system.
* Set up your own test directory as needed.
SET DEFAULT TO \VFP5\TEST
USE \FPW\TUTORIAL\CUSTOMER
COPY STRUCTURE TO TESTDATA
USE TESTDATA
FOR I = 1 TO 600
  APPEND FROM  \FPW\TUTORIAL\CUSTOMER
ENDFOR
CLOSE ALL
```

This program produces a table with 300,000 records that's about 45M in size. You may be able to work with a smaller table, but your results could vary widely and lead to invalid conclusions. You need to have at least three times that much space available on your drive (150M minimum) to achieve accurate results. If you don't have that much space you can try the program using a smaller table, but your results will vary. Walter did extensive testing (hundreds of hours) using tables with 500,000 and 1 million records and ranging from 100M to 1G. The results of those tests may appear in a later article, but the result of this test dictates that they be rerun. (Yes, he wishes this was one of the first tests done -- not one of the last.)

Here's the test:

```
USE TESTDATA
SELECT * FROM TESTDATA WHERE STATE = "NC"
```

That's it! It's that simple. Just watch the status bar as the query progresses and note the length of time it took to find the 10,200 records from NC.

Before you jump the gun, it's important to understand that we're *not* testing query performance. We *know* that there's no index on this table. What we *are* testing is one of the differences between FPW and VFP.

Running this test on a Pentium 133 with 32M of RAM under Windows 95 produced the following results. Under FPW2.6a it took 19 seconds to run this test. Under VFP5.0 it took 115 seconds. That's not a misprint. VFP5.0 was six times slower than FPW2.6a. We've done this same test on several systems ranging from P100 to P166, 16M to 48M RAM, Win95 and

WinNT3.51. All tests were done on local drives, all with lots of spare room (at least 400M free). Using VFP3.0b and VFP5.0 the results were similar. Did we not say that VFP is as fast or faster than FPW? Yes, we did. Have faith and read on.

There's a lot more to analyzing test results than there is to doing the test in the first place. First, a test has to eliminate factors that aren't being tested. (That's why we didn't test across a network, for instance.) Second, a test has to be repeatable. It's at this point -- testing repeatability -- that the test results started to act strangely. We did this basic sequence:

```
USE TESTDATA
SELECT * FROM TESTDATA WHERE STATE = "NC"
* note time elapsed
CLOSE ALL
CLEAR ALL
* repeat the previous three commands 10 times
```

When we tested this 10 times in a row under FPW we got the same basic 19-second result with less than a one-second variation. But the results varied considerably under VFP. From an initial 115 seconds the results dropped to 19 seconds after 10 tests. Obviously, releasing and clearing all didn't do the same thing under VFP that it did under FPW. And this confirmed one of the strange phenomena reported during beta testing: The more you used VFP, the faster it got. But why? And how do we test results of changes if Fox speeds up by itself?

Well, the answer was pretty easy, if time consuming. After each test, we simply quit VFP, shut down Windows, cold-booted the system, and started over. That gave us a pretty consistent test setup. In analyzing why, our early theories covered a broad range of possibilities, ranging from thunking to 32-bit, multi-threading issues. But the more we worked on it, the more evident it became: FPW manages memory itself. It isn't a good Win95 or WinNT client, but it does a great job of managing memory. VFP, on the other hand, lets Windows manage memory *for* it. Unfortunately, this is required if you're going to be a good Win95 or WinNT client, especially if you come from the company that makes Windows. Unfortunately, Windows isn't as good at managing memory as is FPW. Here's what happens.

When VFP starts, it asks Windows how much memory it can have. As always, it wants all it can get. However, Windows uses some odd calculations to tell VFP how much there is, and it's never been right in more than 1,000 tests we've run. You want to consider three values: SYS(1001), SYS(3050,1), and SYS(3050,2).

SYS(1001) is probably familiar to a lot of FPW developers. In FPW, this reported the value of memory available to the FoxPro memory manager based on real memory. But its use has changed. In VFP, it returns the size of the virtual memory pool -- which is about four times the size of physical memory. (The default for Windows 95 is to adjust this by itself; under NT you normally specify it during installation.) On my 32M system, it returns 132M of available memory. (Note: This isn't based on *physical* memory but *available* memory.) Under FPW, SYS(1001) returns less than 2M for the same system.

While making use of virtual memory helps to task-switch among multiple applications, it has a lot of drawbacks when used with a development tool like VFP, or with a custom application running by itself in VFP. Fortunately, the VFP team gave us a new tool to overcome this, but they didn't really say much about when or how to use it. SYS(3050) is used to set buffer memory size. It has two settings, each of which can be passed a value. SYS(3050,1) returns the size of the foreground buffer, and SYS(3050,2) returns the size of the background buffer. If you add a value after the second parameter, you can specify the size you want:

```
* Set the foreground buffer to
* approximately 10M.
? SYS(3050,1,10000000)

* Set the background buffer to
* approximately 6M.
? SYS(3050,2,6000000)
```

This lets you adjust the amount of memory VFP will use if it runs in the foreground (while developing or running your app) or in the background (while you do something else in the foreground like surfing the Net while posting payroll).

This is a really significant function. When we do ?SYS(3050,1) on the 32M system we get a value 19398656. That's 19M-plus that VFP is going to use for foreground processing. The problem is that the 32M system doesn't have 19M-plus of physical RAM available at that time. Looking at the actual RAM with a tool such as Norton Utilities shows that it's several megabytes less. If you reboot the system and run the simple query listed previously, you can see and hear what's going on. Watch the thermometer as the query progresses. It starts out moving fairly quick and steady. Then, partway through, the query slows way down and crawls until completion. If you have a noisy drive, you'll hear a high amount of disk thrashing when the slowdown occurs. VFP thinks it's using physical RAM but it's actually using Virtual Memory. As an example, think of what happens when VFP thinks it has 19M of RAM but only has 16M. It fills the first 16M very quickly but the next 3M is filled much more slowly because it has to write it to the disk (while it's also reading data from the same disk). Then it repeats this process, becoming slower as it goes. You can avoid this problem by setting the value of SYS(3050,1) yourself.

Each system we tested this on had different optimal values. (The default set by VFP was never right.) To find the value that's best for your use, just change the setting and run the simple query. Of course you'll need to quit and start with a clean boot to be most accurate. Start with the default value obtained by ?SYS(3050,1) and decrease it by 1M each time. Our optimum setting was SYS(3050,1,15500000) on the 32M system and SYS(3050,1,7400000) on the 16M system. (VFP converts the amount into actual memory block sizes.) You'll note a point where the performance increase levels out; if you go low enough, it will go back up. You can fine-tune it by varying a few hundred K if desired.

There are, however, other factors that need to be considered. If you run other applications all the time (such as e-mail, Visual Source Safe, and so on) you may want to load them *before* you do

your tests. This way, you'll have a typical working environment because they'll use some of your RAM. If you can live with these apps being swapped to virtual memory, hog the memory with VFP and then start them. Windows will, of course, try to take the memory away from VFP, but when you bring it back to the foreground you'll get it back. You can increase the size of SYS(3050,2), which defaults to about 25 percent of available physical RAM, to prevent other apps from getting as much of VFP's memory when it isn't in the foreground -- or reduce it to as little as 256K to let other apps use more memory when they have focus.

So, what's the deal with having to reboot to test this? Well, 32-bit Windows adjusts memory usage as applications run. It adjusts swap (paging) file size, virtual memory, and cache as it thinks best. That's why VFP seems to speed up the more you use it. The problem with allowing Windows to adjust memory is twofold. First, you start out in low gear instead of in drive, and it takes awhile to get up to speed. Second, if you leave it up to Windows you have no control over when this works and doesn't work. The VFP team gave us the ability to optimize VFP for each system and workload (or at least as much as Windows will permit) with SYS(3050).

As noted, we aren't testing query performance, but the difference in memory usage between FPW and VFP. This improvement affects much more than this simple query. It affects all types of table access (SCX, MNX, VCX, DBF, and so on) as well as processing (index creation, report printing, and more).

What about performance comparisons among versions 2.6, 3.0, and 5.0? All three return different values. In some cases 2.6 is a second faster, sometimes 3.0 is, and sometimes 5.0 is. The variation is generally less than two seconds. You can see this for yourself by using this simple query and selecting different states. (For example, version 2.6 usually finds the 15,200 records in FL two seconds faster than version 5.0, while version 5.0 finds the 73,800 records in CA two seconds faster.) When we average all the queries on all the states, version 5.0 is the winner by a narrow margin. The point here isn't that version 5.0 is a lot faster but that it isn't any slower despite being a much more powerful programming environment. Maintaining performance was no small feat for the VFP team because they optimized version 5.0 to require less memory than version 3.0. Of course, VFP will still use all the memory you can give it, but it won't choke if it has a little less to work with.

We need to say something about the query itself. If these kinds of times were acceptable (developers of some products think this is good) we wouldn't need the power of FoxPro. Remember that queries are designed to work on indexed tables. This is essential for `Rushmore' to work. (If you don't have an index it essentially builds a temporary one to do the query.) If you INDEX ON State TAG State and SET ORDER TO 0 (indexes don't need to be active for `Rushmore') you'll see the real power of FoxPro. All three versions return the query results in less than one second. That's more than 20 times faster because of the presence of the index. If you need to do any regular queries you obviously want to have indexes; FPW and VFP are optimized to work this way. Nonetheless, understanding how Visual FoxPro uses memory -- and having the ability to tune its performance -- adds yet one more tool to your bag of tricks.

Flavio Almeida is a Microsoft Certified Product Specialist in Windows (WOSA I and II) and Visual FoxPro and is based in Raleigh, North Carolina. 919-781-4321, 104545.2274@compuserve.com.

Walter Loughney is the research manager for American TelNet, a Miami, Florida, telecommunications company. He has been developing with Fox since FoxBase 1.0, and was a beta tester for VFP 3.0 and 5.0. He consults and speaks on object-oriented technology and client/server development and is writing a book on object-oriented programming. Walter is a Microsoft Certified Professional and has been a member of the FoxTalk Editorial Advisory Board for two years. 72560.2123@compuserve.com.

Mission: Possible

Whil Hentzen

The calendar says 1997. Orwell's *1984* happened 13 years ago. Hong Kong is going to revert to China in a few months. And if the movies are at all accurate, we'll be walling off Manhattan and throwing all the criminals into it later this year. A big spaceship named Discovery will be heading toward Jupiter in a couple of years (2001). And Captain James T. Kirk is going to take command of his own starship in about 325 years.

Oh, yeah, one more thing. Virtually every computer in the world is going to crash and burn in about 33 months. (Yes, even those running Access 95.)

But that's not all. A lot of applications out there are on their last legs. We don't have to worry about the gazillions of mainframe apps because, fortunately, experienced COBOL programmers are a dime a dozen and just dying for a chance to maintain some 1967 code. However, some of those old FoxBASE and Clipper (S'87) apps reliably toiling away in DOS don't work too well in DOS boxes and their users are getting tired of exiting Windows to run them. Others run okay under Windows -- they're just ugly. But they still don't play well with the other boys and girls in the GUI world. And let's face it: The business world has changed in 10 years. The pace has quickened -- and the business rules hard-coded in those apps are hopelessly out of date. The apps have been hacked and kludged to death, and using them is sometimes more trouble than going back to stone tablet and chisel. Do you want to hear a really scary statistic? There are more than $80 billion of live applications for which the source code can't be found. (You thought you were the only goofball who did that, right?) And over the last 12 months, a new delivery mechanism has arisen: The Internet and its associated technologies have been brought to the forefront as a new way to centralize data and applications but, at the same time, to provide inexpensive access to a larger community of users.

So these four factors -- Year 2000, character mode and DOS apps becoming outdated, business rules changing faster than the apps can be modified, and the Internet delivery model -- represent an incredible opportunity. The opportunity is accentuated because these factors are happening at the same time. Together they present a compelling reason for upgrading and replacing these applications.

Insurmountable opportunities

One of the most famous cartoon lines of all time is from Pogo: *We have met the enemy and he is*

us. Many people don't remember the next line: *We are faced with insurmountable opportunities.*

1997 will be the year that forward-thinking MIS managers look at the state of their systems, consider the factors that I've outlined, and start to (gasp!) plan for the future. We're expecting a number of calls from firms looking to begin system conversions.

Why should they start now? Well, some may be of the opinion that planning is a positive attribute. We'll let them live in that dream world, okay? But we need to face a couple of fear factors as well. First, the supply of developers is more plentiful now than it will be in another 18 or 24 months -- when *everyone* will need their systems rewritten. And not only will it be harder to find experienced, qualified developers, but they're going to need them even worse -- because as the supply and demand curve becomes tilted, profiteering will raise its ugly head. Yes, one side effect of this increased demand will be that these corporate developers will scratch their heads and ask themselves, "Why am I sitting here in this cubicle, making some paltry salary, and getting screamed at and dumped on because my managers didn't plan ahead? Why should I stay here and work till midnight for the next year to handle some emergency that wasn't my fault, when I could be out there as an independent and making a ton of money?" So these corporate folk may walk away, exactly at the time when their companies need them most.

Have I presented a convincing case that demand for our work will go through the roof?

What are we going to do about it? I know I've been wrestling with this situation, and many of you will do the same. The problem is that a lot of people have been rather disconcerted over the past year and a half or two.

Every time a radically new product is introduced (new product or new version -- heck, it's all the same thing), development takes a pause. Kind of like when the new model Corvette is introduced -- the purchasing of the old ones slows. Of course the old version doesn't disappear; some people like it better. (The Sting Ray is still the best looking `Vette of all time, right?) Some people want a deal and don't feel that the square taillights and different knobs on the air conditioner are worth $3,500. And still others just don't have a clue -- they aren't aware that Chevrolet introduces a new model each year, so they just buy whatever's on the lot. (They also pay sticker without asking <g>.)

But most people play the Wait And See game, at least for a while. And they drive Ol' Reliable -- the '69 Dodge Dart that's held up through three wars and six presidents -- until they decide . . .

VFP 5.0 -- the complete tool we've been waiting for

Well, we've kind of been waiting for a year or two for the new version of VFP. Version 3.0 just didn't feel quite right. Sure, it had a lot of really great features. OOP that was out of this world. Control over forms that we could only dream of. The basic rudiments of a data dictionary. True Windows. But there were still a few things that just annoyed me, and made dealing with the product less than the Zen-like experience I was waiting for. The Debugger, for example. Win32s. As long as this kludge was available, customers asked us to use a Ferrari to plow their back 40. We could, but we didn't really want to. And then there's, well, hey -- need I go further? Weren't these two reasons enough to stay in version 2.6 for a while?

Our shop sold four version 5.0 apps in the couple of weeks before DevCon in Phoenix, and we're getting a call a week from more people interested in FoxPro work. We've examined 5.0 pretty carefully, and while there are a couple improvements we'd like to see, we're really happy with it.

We've taken a good look at the competition. And to quote Alicia Silverstone: "*Develop in (fill in the blank with VF/Delphi/PowerBuilder/whatever)? As if!"* Life is too short!

However, development in a new version is a major commitment of resources. Another reason we declined to spend a lot of resources when version 3.0 was released was the tepid backing of the FoxPro marketing folk. We've seen where VFP 5.0 (and future releases) are being positioned. We can make a more educated guess about what 1998 and 2001 might bring, and feel comfortable that VFP is going to have a slice of that market. We feel confident that David Lazar and Robert Green will bring verve to the previously moribund image. And we see that customers are increasingly comfortable as well.

Don't want to be Pollyanna, to be sure. The various hordes in Redmond will continue to leave VFP off slides and some of their sales folk will say the stupidest things. Frankly, if corporate Earth were to wholesale adopt VFP in the next three months, we'd all be in big trouble. How would you like to have to turn down new work every four days? It would just kill you, wouldn't it?

Wait a minute! I just presented a picture where demand is going to shoot through the roof over the next 18 months! What are we going to do? Can we wait until VB 5.0? Delphi 3.0? PowerBuilder 5.0? Oracle99? These tools are just going to keep evolving. It's time to set some roots down and begin to develop apps that first, will last for a few years and second, can be extended as new technologies appear.

I think VFP 5.0 is the ideal platform for developing applications for the next three to four years. Here's why:

1. The technical merits of the tool are numerous. It's difficult to find anyone who has anything bad to say about VFP 5.0. Well, I'm sure we could find one loose cannon in the Pacific Northwest, but, hey, he's busy enough trying to buy a product that should have been buried in 1989.

2. Given what we've seen in public about VFP 6.0, as well as the current intentions of Microsoft and others in the industry, the use of VFP as one of the tools (not the only one, however) for the late 1990s is solid and robust.

3. There are no worthy competitors. Microsoft is continuing to pour resources into VFP, enhancing it and integrating it with the other development tools in the Visual Tools suite. ActiveX integration. Internet Server capability. Common tools.

4. Customers are responding considerably more favorably to VFP 5.0 than 3.0.

5. Last but not least, having people on the VFP marketing team who actually care about the product for a change is rather refreshing as well, wouldn't you say?

So what tool are we going to use? One that chokes on record sets of 100,000? One that comes from a company that hasn't made money in two years? One that requires a $3,000 outlay per developer just to click on an icon? One that requires $1,000 per user for the necessary operating system and back-end licenses? In many cases, Fox is an excellent choice. It's fast, it's inexpensive, we know how to use it, and it's going to grow with us.

The mission is possible.

What's in That Class Library?

Chaim Caron

(1)

Object Orientation greatly increases productivity -- but alas, there's no free lunch: functionality and its corresponding procedural code can be difficult to locate. Have you ever needed to know what functionality is in a class library, and where it's located? Chaim Caron has run into this problem and created a tool -- the Class Reporter -- that will help answer these questions.

Object Oriented Programming (OOP) promises much to both the developer and the client: vastly improved productivity and reduced maintenance expense. OOP gets much of its strength from the twin attributes of inheritance and subclassing -- powerful techniques that eliminate the redundant code inherent in "copy and paste" techniques. Because each child class inherits the functionality of all its ancestors, subclassing permits developers to create a hierarchical tree of classes and place each function in exactly the right node of the class tree. Like data in a normalized database, each piece of functionality belongs in exactly one location.

OOP: the promise and the peril

If the *promises* are increased productivity and reduced maintenance, the *perils* are learning the functionality of an undocumented or under-documented class library and finding specific code or functionality in a class or class library. Becoming familiar with a new class library can be difficult because of the way in which code is distributed among the methods within a class and among the class's parents and children. We currently have to read the method code for every method in a class hierarchy, similar to the sleuthing required with a complex set of nested procedure files.

Finding code that 'you know is in there, somewhere' is just as difficult. For example, I once had to modify a certain behavior in a form class. I had written the class library myself but hadn't documented it in great detail. I knew that the behavior that needed to be changed was in the Activate method, but I couldn't remember which class held that functionality. In the "old days," I had to open some or all of the classes in the class hierarchy using the project manager and inspect the Activate method code in each class to find the right spot. Because I expected this situation to come up again and again, I wrote a tool, the Class Reporter (CR), that makes short

work of both tasks.

What CR does

The Class Reporter is a utility written in VFP that displays selected information about one or all methods in a selected class and for all classes in the class hierarchy to which the selected class belongs. The developer can specify any class in a class library as a starting point, and CR will work its way up to the top of the hierarchy as needed.

CR makes developers' lives easier by creating a pair of reports that show method code organized by class and by method, thus permitting the developer to see at a glance what functionality exists and where the functionality resides in the class hierarchy. The developer can also create self-documenting method code with CR.

In this article, "higher up" the class hierarchy means closer to the base class or first level parent. "Lower down" the class hierarchy means farther from the base class or closer to the last child level.

Methods in each class report

The first report (see Figure 1) is organized around the class -- it shows all methods in a class. For each class in the specified class hierarchy, this report shows:

- The class and class library name.
- The names of all methods that first appear in the class (that is, this method isn't overridden or modified by code for the same method lower down the class hierarchy).
- The names of all methods that recur in this class (that is, this method overrides or modifies code for the same method in some class lower down the class hierarchy).

This report is for convenience and contains information already available from the class browser.

Classes for each method report

The second report (see Figure 2) is organized around the method -- it shows all classes for which this method has code.

The report header shows the class hierarchy from the top-level class down to the specified class. The class and class library are shown for each level. Then, for each method, information is shown for each class for which this method contains any code. The user can determine which information about the method will be shown: one method or all methods; comments only or all source code.

Option selection (main)

The user can specify the starting class and class library using the Main option form (see Figure 3). Instead of typing in the name of the class library, the user clicks on the button with the ellipses (to the right of the ClassLib field) and the standard VFP locator dialog box is called.

Once the ClassLib is selected, the user can select the class within the class library.

Next, the user can select either or both of the reports. The reports can be sent to the screen or the printer.

Option selection (additional)

The Additional Options form (see Figure 4) permits the user to customize the operation of the Class Reporter. A description of additional options follows.

Method Report -- What To Show: Oftentimes I want to look at only selected comments. At other times, I want to see all comments or even all source code. With the What To Show option, I can request only the information I need from each selected method. The choices are:

- All Source Code
- All Comments
- Selected Comments

If you select "Selected Comments," you must also enter a Selected Comment Prefix. CR will only report comments that begin with the text string you enter in this field. This is a powerful option, allowing you to use naming conventions for various types of comments so that they can be pulled out selectively. For example, if you began method description comments with "*$" (rather than simply "*"), CR would pull out just those descriptions if you specify "*$ \" as the Selected Comment Prefix.

Messages to yourself (or other developers working on the same project) such as "Still to do" or "Future Enhancement" could be marked with a different string, such as "*?". Using the string "*?\" as the Selected Comment Prefix would generate a "to do" or "future enhancement" list for you.

The Selected Comment Prefix itself contains a delimiter at its head and tail. For example, if the Selected Comment Prefix is "*$ \" (the default), then CR would only report on comments that start with "*$". You may select any character for a delimiter as long as you use the same character for the leading and trailing delimiter, so "9*$ 9" yields the same result as "*$ \".

Method Report -- Method Selection: Show "All Methods" or "One Method." If you select "One Method," specify the method name and both reports will show only the selected method.

Both Reports -- Specified Class Only: If you select this option, the reports will show the specified class only and won't walk up the class hierarchy.

Both Reports -- Tool Tips: This option permits you to turn off Tool Tips in the Option Selection Form. This option doesn't affect the reports.

Both Reports -- Erase Temporary Cursors: If you want to retain the temporary cursors created by CR, deselect this option.

Creating self-documenting class libraries

Learning class libraries (and passing this knowledge to others) requires complete and accurate documentation. In the real world, developers sometimes find it difficult to keep documentation accurate when the system changes.

Fortunately, there's a technique for keeping documentation in sync with the system it describes: Simply store documentation as comments in code (all code should be well documented anyway) and keep the comments up to date. All you need is a mechanism to print the comments, and Class Reporter provides this capability.

To create self-documenting class libraries, simply do the following:

1. Select a prefix for comments that you want to appear in the documentation (such as "*$").

2. When you create a comment that you want to appear in the documentation, use the prefix selected in step 1. For example:

```
*$ This is an important comment.
```

3. Keep the comments in your class libraries up to date.

4. To create a documentation report, run CR using the Selected Comment Prefix technique as described earlier (in the section titled Method Report -- What To Show.)

I'd like to thank Yair Alan Griver of Flash Creative Management Inc. for the idea for self-documenting code.

Other notes

The Class Reporter can accept an SCX-based form as the specified class.

CR currently can't show all the classes in an entire hierarchy tree. It can show only the classes in one branch, starting at one specified class and working its way up to the top. CR doesn't report on custom properties.

Running the program

To run Class Reporter, type "DO CR" in the command window. The app attempts to auto-detect your screen resolution and selects the appropriate form (the app contains two versions of the main form: one for 640x480 resolution and one for higher resolutions). If the auto detect doesn't work properly, you can override it by starting the program as follows:

```
DO CR WITH "HiRes" -or- DO CR WITH "LoRes"
```

Warning: If you specify "HiRes" or "LoRes," the system variable _FoxGraph will be modified.

Programming notes

CR extracts data from the .VCX and .SCX files. PEMSTATUS wasn't used. Method source code was extracted from memo fields character by character. For details, take a look at ColMeths.PRG, available in the accompanying Download files (02CARON3.EXE for version 3.0b, 02CARON5.EXE for version 5.0).

CR is a small utility and was therefore implemented using a few shortcuts which I don't recommend for industrial-strength system development. In particular, the form contains objects that are subclassed directly from VFP base classes. Also, much of the procedural code is in PRGs rather than methods. Because the entire system consists of only one main form, I felt this shortcut was reasonable.

An .APP file and all source code, in both 3.0b and 5.0 versions, is available in the accompanying Download files (02CARON3.EXE for version 3.0b, 02CARON5.EXE for version 5.0). I encourage and welcome your comments and suggestions. I'll fix bugs but can't commit to a particular time table. I encourage you to take bug fixing into your own hands, if necessary. I present this article and utility in the hopes of making it easier for developers to work with OOP methodology in the real world of system development, debugging, and enhancement.

Chaim Caron has been programming with FoxPro since 1990. He is the president of Access Computer Systems Inc., a consulting firm in New York City that provides custom FoxPro development services and productivity training. He was the chapter leader of the New York City FoxPro user group for three years and has been a speaker at user group meetings throughout the area. 212-627-8507, 72520.2272@compuserve.com.

PEMs for Your Base Classes

Doug Hennig

This month's column examines some ideas to put in your subclasses of Visual FoxPro base classes. Some of these ideas are straightforward, but others may be less obvious. We'll look at some custom properties and methods you can add to provide more automatic functionality to your applications.

You've probably heard a thousand times by now that using the Visual FoxPro base classes in your applications is bad, and that you should create your own subclass of each VFP base class. I won't bore you with the reasons why in this article. Instead, we'll look at the subclasses I've created for my use. In this article, I refer to the subclasses I've created as "my base classes" because I always use them in my applications as the starting point for any subclasses. When I

want to refer to a class built into VFP (such as a ComboBox), I'll refer to it as a "VFP base class."

While you could create a set of your own base classes by simply subclassing VFP base classes without making any changes, I find that I usually end up changing or adding something to any VFP base class, so why don't we construct our base classes so that they include the behavior we want directly? We'll look at the changes I've made to the properties, events, and methods (PEMs) of VFP's classes. Some of these changes are straightforward, such as setting the BackStyle property of a Label subclass to Transparent so the background of the container shows through. Others may be less obvious, such as having the Error method of an object look up its containership hierarchy to find the first parent with error handling code.

Common PEMs

All classes in the CONTROLS.VCX class library (included in the accompanying Download file) have some common properties and methods added or filled in. In addition, some classes that have common behavior (such as those that allow the user to change their values) have the same methods added or filled in. These common PEM changes are described here.

About

This custom method is added to every class as a way to document the class. I've seen other developers add "Copyright" or "Documentation" methods to serve the same purpose. Whatever the name of the method, it generally consists of comments that describe the features of the class. Some folks enter the comment text between TEXT and ENDTEXT statements (so you don't have to comment out each line); I just used comment lines. In the About method for each class, I included the following information:

- The name of the class.
- The class this class is based on.
- The purpose of the class.
- The author, copyright, and last revision date.
- The name of the INCLUDE file used by the class (if any).
- Changes made to properties of the parent class.
- Changes made to methods of the parent class.
- Custom public properties added to this class.
- Custom protected properties added to this class.
- Custom public methods added to this class.
- Custom protected methods added to this class.

Here's the About method from the SFCheckBox class as an example:

```
*======================================================
* Class:         SFCheckBox
* Based On:      CheckBox
* Purpose:       Base class for all CheckBox objects
* Author:        Doug Hennig
* Copyright:     (c) 1996 Stonefield Systems Group Inc.
* Last revision: 11/02/96
* Include file:  none
*
* Changes in "Based On" class properties:
*AutoSize:     .T.
*BackStyle:     0 (Transparent)
*Value:         .F. since check boxes usually are
*               used for logical values
*
* Changes in "Based On" class methods:
*Error:         calls the parent Error method so
*               error handling goes up the
*               containership hierarchy
*InterActiveChange:  call AnyChange()
*ProgrammaticChange: call AnyChange()
*
* Custom public properties added:
*Builder: holds the name of a custom builder
*
* Custom protected properties added:
*None
*
* Custom public methods added:
*About:         for documentation purposes
*AnyChange:     called by InteractiveChange() and
*               ProgrammaticChange()
*Release:       releases the object
*
* Custom protected methods added:
*None
*======================================================
```

Error

In the June and July 1996 issues of *FoxTalk* (see "Error Handling in Visual FoxPro" and "An Error Handling Class for VFP," respectively), I discussed an error handling strategy in which all objects handle any errors they can and pass any other errors to the Error method of the form they reside in so common error handling services can be centralized. I've since refined this scheme; objects now call the Error method of their container rather than jumping directly up to the form (actually, calls go up the class hierarchy first before going up the containership hierarchy). This allows additional levels of localized error handling. However, this scheme has one problem: Controls sitting on base class Page, Column, or other containers with no Error method code essentially have no error trapping because they call an empty method! The solution is to travel

up the containership hierarchy until I find a parent that has code in its Error method. If I can't find such a parent, then I display a generic error message (this isn't likely, because I base all forms on the SFForm class, which does have Error method code). Here's the code used in all objects except SFForm and SFToolbar:

```
LPARAMETERS nError, cMethod, nLine
local oParent
oParent = iif(pemstatus(Thisform, ;
  'FindErrorHandler', 5), ;
  Thisform.FindErrorHandler(This), .NULL.)
if isnull(oParent)
  = messagebox('Error #' + ltrim(str(nError)) + ;
    ' occurred in line ' + ltrim(str(nLine)) + ;
    ' of ' + cMethod, 0, _VFP.Caption)
else
  oParent.Error(nError, This.Name + '.' + cMethod, ;
    nLine)
endif isnull(oParent)
```

This code checks to see if the form the control is sitting on has a FindErrorHandler method, and if so, calls it to locate the first parent of the control with code in its Error method (you'll see the code for this method in a moment). If such a parent is found, its Error method is called with the same parameters that this Error method received, except the name of the object is added to cMethod so your error handling services can know in which object the error originated.

The SFForm and SFToolbar classes, because they're the "top-level" containers (I never use form sets), have a different Error method than do other objects. (It's beyond the scope of this article to discuss an error handling object; see my column in the July 1996 issue for a simple example of such a class.) This Error method checks to see if an oError object has been added to the form, and if so, calls its ErrorHandler method to provide global error handling services. If not, an error message is displayed:

```
LPARAMETERS nError, cMethod, nLine
local laError[1]
aerror(laError)
if type('This.oError') = 'O' and ;
  not isnull(This.oError)
  This.oError.ErrorHandler(nError, ;
    This.Name + '.' + cMethod, nLine)
else
  if messagebox('Error #' + ltrim(str(nError)) + ;
    ' (' + laError[2] + ')' + chr(13) + ;
    'occurred in line ' + ltrim(str(nLine)) + ;
    ' of ' + cMethod, 17, _VFP.Caption) = 2
    cancel
  endif messagebox ...
endif type('This.oError') = 'O' ...
```

As I just mentioned, the Error method of all objects calls the FindErrorHandler method of the form. This method accepts as a parameter a reference to the object that called it. It starts with the parent of that object and travels up the containership hierarchy until it finds a parent with code in its Error method. This prevents the problem of error handling stopping on base class Page, Column, or other containers because they have no code in the Error method. Here's the code for that method:

```
lparameters oObject
local oParent
oParent = oObject.Parent
do while type('oParent') = 'O' and ;
  not isnull(oParent)
  do case
    case pemstatus(oParent, 'Error', 0)
      exit
    case type('oParent.Parent') = 'O'
      oParent = oParent.Parent
    otherwise
      oParent = .NULL.
  endcase
enddo while type('oParent') = 'O' ...
return oParent
```

Builder

Ken Levy created a tool for VFP 3.0 called BuilderX that was included with the *Visual FoxPro 3 Codebook* by Yair Alan Griver. This tool acts as a wrapper for VFP's native builder application (BUILDER.APP). Its main purpose is to allow you to define a specific builder for each class, stored in a custom Builder property of the class. When you invoke the builder for an object created from the class, BuilderX first checks if the object has a Builder property and if it contains anything. If so, it instantiates an object of the class specified in that property (you can also specify the class library in which your builder class is located by entering the VCX name and class name separated by a comma). If not, it calls BUILDER.APP to carry on as it normally would. This allows you to create customized builders for any class (perhaps using Levy's BuilderB tool, which is the source of another article) and specify that builder in the Builder property of the class itself. This idea is so cool that Microsoft added this behavior to the native BUILDER.APP in VFP 5.0 so you don't need BuilderX.

I've added a custom Builder property to all my base classes so I can take advantage of this capability. This property contains a blank value for all classes, so it's really there as an "abstract" property. I use it for specialized subclasses of my base classes, although you could also use it for simple builders for even base classes.

Release

I add a custom Release method to any class that doesn't already have one. It contains just one line of code:

```
release This
```

This permits you to use a common way to release any object: <object>.Release().

AnyChange, InteractiveChange, ProgrammaticChange

Classes that allow the user to change their values (such as TextBox, CheckBox, EditBox, and so on) have InteractiveChange and ProgrammaticChange methods. Usually if you put any code into one of these methods you end up having the other method call it so the same behavior takes place regardless how the object's value was changed. To make this work automatically, I added a custom AnyChange method to those classes and added the following to their InteractiveChange and ProgrammaticChange methods:

```
This.AnyChange()
```

Thus, any code needed when any change is made to the value of the control should be entered into the control's AnyChange method.

SelectOnEntry

Microsoft added this new property to several controls in VFP 5.0. If you set it to .T., the text in the control will automatically be selected when the control receives focus. Because this is the default behavior I want, I set it to .T. in all classes with this property.

Valid, Validation

The Valid method of controls that have it (such as ComboBox and TextBox) has two behaviors that annoy me:

- It's fired even when the user clicks on a "cancel changes" button (unless this button is in a toolbar), which means that the user has to enter a valid value into a control before he or she can cancel the changes made to a record.
- If I put code into the Valid method of a class (for example, to overcome the first problem), when I need custom Valid code in a subclass or an object instantiated from the class, I usually have to do something like this:

```
  dodefault()     && execute the normal behavior
* custom validation code here
nodefault      && don't re-execute the normal behavior
```

To overcome the first problem, I added a custom lCancel property to my CommandButton base class, which allows me to indicate that a button is a "cancel changes" button (the Cancel property of a button indicates that the button is selected by pressing Escape, but I may not want that

behavior for my "cancel changes" button). The Valid method -- of all classes that have one -- then checks to see if a "cancel changes" button was pressed before performing any validation code. It does this by using the SYS(1270) function to get an object reference to the control the user just clicked on, and checking to see if that object has an lCancel property (and if so, whether it's .T. or not).

To solve the second problem, I added a new Validation method to these same classes and made the Valid method call it. This way, I leave the Valid method of an object alone and instead put my specific validation code into the Validation method.

Here's the code for the Valid method:

```
local oObject
oObject = sys(1270)
if lastkey() = 27 or (type('oObject') = 'O' and ;
  type('oObject.lCancel') = 'L' and oObject.lCancel)
  return .T.
endif lastkey() = 27 ...
return This.Validation()
```

My base classes

Here's the description of the PEM changes in each of my base classes. These classes can be found in the CONTROLS.VCX file in the accompanying Download file. All classes are named according to the VFP base class they're based on, plus an SF (Stonefield) prefix. For example, SFCheckBox is based on the CheckBox base class.

SFCheckBox

- *AutoSize*: set to .T. so I don't have to manually size the control to fit the caption.
- *BackStyle*: set to 0 -- Transparent so the background color of the container shows through.
- *Value*: set to .F. because I normally use a CheckBox for logical values.
- *About*, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, and *Release*: described earlier.

SFComboBox

- *aItems[1]*: I added this custom property so if desired, the ComboBox can be self-contained; its source of display items is a property of itself.
- *BoundTo*: I set this new VFP 5.0 property to .T. so the ComboBox will properly work with numeric data sources (see the VFP 5.0 documentation for more information on this property).
- *GetActiveProperties*, *Init*, and *lActive*: see SFSpinner for information on these changes.

Init also initializes the This.aItems array to an empty string.

- *ItemTips*: set to .T. so this new VFP 5.0 feature is turned on.
- *RowSource* and *RowSourceType*: set to This.aItems and 5 -- Array, respectively, because I most frequently tie ComboBoxes to arrays. You can, of course, easily override these properties as needed.
- *About*, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFCommandButton

- *lCancel*: .T. if this button is used as a "cancel" button (this allows the Valid method of a control to avoid validation if the user clicked a "cancel changes" button as described earlier).
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFCommandGroup

- *BackStyle*: set to 0 -- Transparent so the background color of the container shows through.
- *ButtonCount*: 0 so buttons can be added from the SFCommandButton class if desired.
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFContainer

- *BackStyle* and *BorderWidth*: set to 0 -- Transparent and 0, respectively, so the container itself has no visible appearance.
- *SetEnabled*: While disabling a container causes the member objects of the container to be disabled, they don't appear to be disabled. I added this custom method to set the Enabled property of the object and all member objects to the specified value so all objects appear to be enabled or disabled appropriately:

```
  lparameters tlEnabled
This.SetAll('Enabled', tlEnabled)
This.Enabled = tlEnabled
```

- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFControl

- *BackStyle* and *BorderWidth*: set to 0 -- Transparent and 0, respectively, so the control itself has no visible appearance.

- *SetEnabled*: see SFContainer.
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFCustom

- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFEditBox

- *IntegralHeight*: set to .T. so the box always displays the last line of text properly.
- *About*, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFForm

- *Destroy*: hides the form so it appears to go away faster:

```
  This.Hide()
```

- *Init*: Because the AutoCenter and BorderStyle properties are used at design time as well as runtime, and they can be a hassle when used at design time, I added new lAutoCenter and nBorderStyle properties to contain the desired runtime AutoCenter and BorderStyle values, and made the Init form do the following:

```
  with This
.AutoCenter  = .lAutoCenter
.BorderStyle = .nBorderStyle
endwith
```

- *lAutoCenter* and *nBorderStyle*: new custom properties as described earlier. They default to .T. and 2 (double border), respectively.
- *Load*: set some environmental things the way we want:

```
  set talk off
set safety off
set deleted on
set fullpath on
set exact off
set unique off
```

- *oError*: a reference to an ErrorMgr object.

- *ShowTips*: set to .T. so tool tips appear.
- *About*, *Builder*, *FindErrorHandler*, and *Error*: described earlier.

SFGrid

- *AllowHeaderSizing*, *AllowRowSizing*, and *SplitBar*: all set to .F. because I don't want that behavior by default.
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFImage

- *BackStyle*: set to 0 -- Transparent so the background color of the container shows through.
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFLabel

- *AutoSize*: set to .T. so I don't have to manually size the control to fit the caption.
- *BackStyle*: set to 0 -- Transparent so the background color of the container shows through
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFLine

- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFListBox

- *aItems[1]*: I added this custom property so if desired, the ListBox could be self-contained; its source of display items is a property of itself.
- *IntegralHeight*: set to .T. so the box always displays the last line of text properly.
- *ItemTips*: set to .T. so this new VFP 5.0 feature is turned on.
- *RowSource* and *RowSourceType*: set to This.aItems and 5 -- Array, respectively, because I most frequently tie ListBoxes to arrays. You can, of course, easily override these properties as needed.
- *About*, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, and *Release*: described earlier.

SFOLEBoundControl

- *About*, *Builder*, *Error*, and *Release*: described earlier

SFOptionButton

- *AutoSize*: set to .T. so I don't have to manually size the control to fit the caption.
- *BackStyle*: set to 0 -- Transparent so the background color of the container shows through.
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFOptionGroup

- *BackStyle*: set to 0 -- Transparent so the background color of the container shows through.
- *ButtonCount*: 0 so buttons can be added from the SFOptionButton class if desired.
- *About*, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, and *Release*: described earlier.

SFPageFrame

- *PageCount*: 0 so pages can be added or removed more easily.
- *TabStyle*: I set this new VFP 5.0 property to 1 -- Nonjustified because I want this appearance by default.
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFSeparator

- *About* and *Release*: described earlier.

SFShape

- *BackStyle*: set to 0 -- Transparent so the background color of the container shows through.
- *SpecialEffect*: set to 0 -- 3D.
- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFSpinner

- *GetActiveProperties*: If the custom property lActive is .T., this method sets the InputMask and Format properties of the control to the values stored in the database for the control's ControlSource. This allows these properties to be updated dynamically whenever the database properties change. Here's the code for this method:

```
  local lcControl, ;
  lnDot, ;
  lcAlias, ;
  lcFormat, ;
  lcInput
with This
  lcControl = .ControlSource
  lnDot     = at('.', lcControl)
  if .lActive and lnDot > 0 and not empty(dbc())
    lcAlias = left(lcControl, lnDot - 1)
    if indbc(lcAlias, 'Table') or ;
      indbc(lcAlias, 'View')
      lcFormat    = dbgetprop(lcControl, 'Field', ;
        'Format')
      lcInput     = dbgetprop(lcControl, 'Field', ;
        'InputMask')
      .Format     = iif(empty(lcFormat), .Format, ;
        lcFormat)
      .InputMask = iif(empty(lcInput),  .InputMask, ;
        lcInput)
    endif indbc(lcAlias, 'Table') ...
  endif .lActive ...
endwith
```

- *Init*: If the custom property lActive is .T., get the active properties for this control by calling the GetActiveProperties method:

```
  with This
  if .lActive
    .GetActiveProperties()
  endif .lActive
endwith
```

- *lActive*: This custom property, which is initially set to .F., indicates whether the Init method calls the custom GetActiveProperties method.

About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFTextBox

- *GetActiveProperties*, *Init*, and *lActive*: see SFSpinner.
- *About*, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFTimer

- *About*, *Builder*, *Error*, and *Release*: described earlier.

SFToolbar

- *Destroy*: hides the toolbar so it appears to go away faster:

```
This.Hide()
```

- *oError*: a reference to an ErrorMgr object.
- *About*, *Builder*, *FindErrorHandler*, *Error*, and *Release*: described earlier.

Conclusion

Some of the ideas for the PEM changes I discussed in this article came from other folks, especially the FoxGang on CompuServe, who every day unselfishly share a wealth of great ideas. I don't recall (or even know) who originated some of these ideas, but if you recognize your idea in this column, thank you!

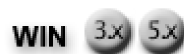
Next month, I'll cover some special subclasses of these base classes, including an EditBox that automatically expands keywords to complete text (similar to the AutoCorrect function in WinWord) and a ComboBox that supports a feature similar to Quicken's "quick fill" function.

DOWNLOAD

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of The Visual FoxPro Data Dictionary in Pinnacle Publishing's The Pros Talk Visual FoxPro series. Doug has spoken at user groups and regional conferences all over North America. 75156.2326@compuserve.com.

Add MDI Window Management to Your Application

Mike Lewis



Users of MDI applications expect to have a Window menu that contains options for cascading, tiling, closing, and selecting the open windows. Visual FoxPro 3.0 supplies some of those functions, but in a half-hearted way -- and you have to work to incorporate them into your applications. In this article, Mike Lewis explains how to give your users the MDI window management they expect.

Visual FoxPro applications are inherently Multiple Document Interfaced (MDI) in nature, which means that users can open several forms at the same time and jump from one to the other at will. MDI can greatly enhance your application by allowing users to view customer and product

records at the same time as they enter orders, for example. But once you permit more than, say, three forms to be open simultaneously, you need to provide some form of window management. The usual way of doing that is through a Window menu.

To see what this involves, take a look at the Window menu in any mainstream MDI application, such as Microsoft Word, Excel, PowerPoint, or VFP itself (see Figure 1). Although the exact composition of the Window menu varies, it usually provides at least the following items:

- **Cascade**. This resizes the open windows and arranges them in an overlapping stack so they fit comfortably in the parent window.
- **Arrange All**. Also known as Tile, this arranges the windows so they're all visible without overlapping. Some applications offer separate options for horizontal and vertical tiling.
- **Close All**. This does just what its name suggests.
- **A list of the open windows**. When the user selects an item from the list, the application transfers its focus to the corresponding window.

VFP's own Window menu offers the second and fourth of these options, along with some less common ones of its own (Hide, Clear, and Cycle, plus direct jumps to the Command Window and View Window). However, it lacks Cascade and Close All commands.

To equip your application with a useful Window menu, you need to do three things. First, bring VFP's built-in Window menu into the application. Next, prune it to get rid of the FoxPro-specific items. Finally, extend it so that it supports Cascade and Close All.

Incorporate the standard menu

The simplest way to import the VFP Window menu (or any other standard VFP menu) into your application is to use the Quick Menu feature in the menu designer. Here's what you do:

1. Create a new menu in the menu designer by selecting Menus in the Project Manager and choosing New.

2. Select Quick Menu from the Menu menu. The menu designer will populate its window with the prompts, bar numbers, and sub-menus of the VFP system menus.

3. Carefully delete the menu items that you don't want. In the Window menu, delete everything except Arrange All.

4. Add your own menu items -- those which are relevant to your application.

You'll end up with a complete set of menus for your application, complete with a rudimentary Window menu.

If you've already created your application's menus, you can add a Window menu retroactively. In the menu designer, add a new prompt at the top level and label it Window (it's conventional to place it just before the Help menu). In the Prompt Options dialog box, click Pad Name and type _MSM_WINDO. Then create a sub-menu for this menu. In the sub-menu, insert a prompt for

Arrange All with the bar number _MWI_ARRAN.

When you run the application, the Window menu will initially contain just the Arrange All prompt. As you open forms within the application, their names will be added to the foot of the Window menu.

You can now execute the Arrange All command. When you do, you'll see the forms neatly tiled on the desktop. You can transfer focus to a form by clicking on its name at the foot of the menu. This functionality comes free with the Window menu, so you don't need to write any code to get the benefits.

Unfortunately, the same isn't true for the Cascade and Close All options. These functions aren't provided within VFP, so you need to cut your own code to make them happen.

The Cascade function

Before we look at the code, let's consider what the Cascade function has to do. Microsoft's interface guidelines for Windows don't stipulate exactly how this function should behave; they merely state that cascaded windows should be "offset slightly to the right and below." But if you look at other applications you'll observe fairly consistent behavior.

Most applications handle cascading in the following way (see Figure 2):

1. Resize all windows so that each occupies three-quarters of the parent window.

2. Move the first window to the top left corner of the parent.

3. Place each subsequent window on top of the previous one, but offset down and to the right by an amount equal to the height of the title bar.

4. If, as a result of step 3, a given window doesn't fit within the boundaries of the parent, move it to the top left corner of the parent.

5. Repeat steps 3 and 4 until all windows have been handled.

The following listing shows how this scheme might be implemented:

```
* Cascade routine. Cascades all the open forms.
PROCEDURE Cascade
local lnHorizOff, lnVertOff, lnNewWidth, ;
lnNewHeight, lnNewLeft, lnNewTop, lnFmIdx
lnHorizOff = Sysmetric(9)    && horizontal offset
lnVertOff = Sysmetric(9)     && vertical offset
lnNewWidth = 0.75 * _SCREEN.Width    && new width
lnNewHeight = 0.75 * _SCREEN.Height && new height
lnNewLeft = 0
lnNewTop = 0
for lnFmIdx = 1 to _SCREEN.FormCount

 WITH _SCREEN.Forms(lnFmIdx)

  IF Type(".BorderStyle")<>"U" AND .BorderStyle=3
   * Form has a resizable border, so adjust its
   * dimensions, subject to min/max height, width.
   .Width=IIF(.MinWidth = -1, ;
      lnNewWidth,Max(lnNewWidth,.MinWidth))
   .Width=IIF(.MaxWidth = -1, ;
      lnNewWidth,Min(lnNewWidth,.MaxWidth))
   .Height=IIF(.MinHeight = -1, ;
      lnNewHeight,Max(lnNewHeight,.MinHeight))
   .Height=IIF(.MaxHeight = -1, ;
      lnNewHeight, Min(lnnewHeight,.MaxHeight))
  ENDIF

  * Move the form to its new position.
  IF (lnNewTop + .Height < _SCREEN.Height) ;
   AND (lnNewLeft + .Width < _SCREEN.Width)
    * The form will not overflow the main window,
    * so go ahead and move it.
    .Move(lnNewLeft,lnNewTop)

    * Calculate the position of the next form.
    lnNewLeft = lnNewLeft + lnHorizOff
    lnNewTop = lnNewTop + lnVertOff

  ELSE
    * Form does overflow, so start stack again
    * at top left corner of the main window.
    .Move(0,0)
    lnNewLeft = lnHorizOff
    lnNewTop = lnVertOff
  ENDIF

 ENDWITH

ENDFOR
ENDPROC
```

This code is generic, in that it doesn't know the names of the forms that it's working with.

Instead, it relies on two properties of the _SCREEN system variable.

If you haven't yet become acquainted with _SCREEN, I urge you to do so. Essentially, it's a device that allows the main VFP window to be manipulated as if it were an object. With about 70 properties and 21 methods, it's an immensely useful piece of equipment. You can use it to alter the main window's title, to assign an icon to the main window, and a great deal more.

In this case, you're using two properties of _SCREEN: FormCount is a count of the forms that are open on the screen, and Forms is an array of the individual forms. You'll use these properties to iterate through the open forms, changing their sizes and positions as you do so.

There are, however, a couple of things to watch out for. First, if a form has a border style other than 3, it can't be resized. Before you can test the BorderStyle property, you must verify that this property actually exists. That's because the Forms array can also hold floating toolbars, and these don't have a border style. Use the Type() function to verify that the property exists.

Even if the form is resizable, it might be subject to minimum and maximum sizes, as stipulated by its MinWidth, MinHeight, MaxWidth, and MaxHeight properties (a setting of -1 for any of these properties means that there's no limit). The resizing is handled by the four assignment statements which follow the test of the border style.

With resizing out of the way, you can now move the form to its new position. Remember, the offset is equal to the height of the title bar. To determine this figure, call Sysmetric() with a parameter of 9.

The Close All function

The Close All routine is much simpler than Cascade:

```
* Close All routine
LOCAL lnFmIdx
FOR lnFmIdx = _SCREEN.FormCount to 1 STEP -1
  IF _SCREEN.Forms(lnFmIdx).BaseClass = "Form"
    _SCREEN.Forms(lnFmIdx).Release
  ENDIF
ENDFOR
ENDPROC
```

Again, the code works by iterating through the Forms array of _SCREEN, but this time it steps through the forms backwards. That's because, as each form is closed, each of the remaining forms moves down one position within the array. By starting at the top of the array and working down, you avoid referencing empty array elements.

Like Cascade, Close All must avoid operating on floating toolbars (docked toolbars aren't a problem because they don't show up in the Forms array). You can't call a floating toolbar's Release method because it doesn't have one. The routine, therefore, tests the base class of the current object and ignores it if it's anything other than a form.

Putting it together

The final step is to wire the Cascade and Close All routines into the Window menu. Start at the menu designer. In the Window sub-menu, add a prompt for Cascade, just above Arrange All. Set its result field to Command, then type a command to execute the Cascade routine (DO Cascade, for example).

Now repeat the process for Close All. This time, place the prompt just below Arrange All. Enter a command to execute the Close All routine, such as DO CloseAll.

Your application now has a fully furnished Window menu containing the MDI window management features your users expect. The new menu might not bring any exciting new functionality to your application, but it will make life easier for your users. It will also bring your application into line with other MDI programs, and -- most important -- it will add that vital professional look to your work.

Mike Lewis is an international technical journalist and a regular contributor to Pinnacle publications. He also teaches Visual FoxPro at the Database Programmers Retreat, a training organization with centers in Painswick, England, and St. Augustine, Florida. 100012.2105@compuserve.com.

More Visual FoxPro 5.0

John V. Petersen

Changing the comment character

Of all the new features in VFP 5.0, the many editor enhancements (colored syntax, procedure lists, comment blocks, and so on) are some of the most useful. Not only has Microsoft given us many new features, they are, in the FoxPro tradition, configurable. The one big exception to this rule, however, is the comment character style. While you can configure the comment color in the options dialog box, you can't change the comment character -- or can you?

As FoxPro has evolved, the role of the resource file has been mostly one of backward compatibility. VFP relies on the Windows registry for storing preferences. All of the options for Visual FoxPro 5.0 are stored under the following key (see Figure 1):

```
Software\Microsoft\VisualFoxPro\5.0\Options
```

Although you can modify just about anything in the Visual FoxPro options dialog box, Microsoft didn't expose the ability to change the comment character. By hacking the Windows registry, however, you can provide this capability yourself to override the default characters of "*!*".

Underneath the Options key is a listing of options for VFP. Among the options are TALK, PATH, and SAFETY. To customize the comment character, add the following option:

```
EditorCommentString
```

Once you've opened the registry to the point shown in , you can follow these steps to make a new option entry:

1. In the right pane of the registry, click the right mouse button.

2. From the New menu, select String Value to add a new String Value.

3. Replace the default name, New Value #1, with *EditorCommentString*.

4. Click the right mouse button on the newly added option and select Modify.

5. In the Edit String dialog box, enter a new comment character (be sure it begins with the "*" character!).

Your registry should now appear as in Figure 2.

Now, when you comment a block of code, it will include your custom comment characters (see Figure 3).

An important note: If you have code that uses another comment character, choosing the Uncomment option from the right click menu of the editor will no longer work. Because you've changed the editor comment string, VFP will no longer recognize blocks of code that use other comment character strings. Be sure to keep this in mind when making this change. Finally, hacking the registry isn't for the light of heart. Be sure you back up your registry before making extensive modifications.

DIRECTORY()

Finally, Microsoft has included native support for testing the existence of a directory. In prior versions, a developer had to resort to a UDF that would use a combination of setting the default to a directory and in-line error trapping. If an error occurred, the directory was invalid. While this method was effective, it's still more desirable to use a native function to handle such tasks.

This function can be used in one of two ways. The first way is to use a hard-coded drive letter:

```
llExists = DIRECTORY("P:\AAA")
```

The second, and perhaps better, alternative is to use VFP's new ability to use the Universal Naming Convention (UNC):

```
llExists = DIRECTORY("\\FS1\VOL2\AAA")
```

If you need to write applications in version 3.x, create a UDF called DIRECTORY that will handle the task of testing for the existence of a directory. For most of you, this will only entail renaming a current routine. Then, when you're able to migrate your app to version 5.0, you'll be able to take advantage of the native functionality without modifying a single line of code.

Enhanced format property settings

The Format Property has two new settings in version 5.0:

- YS: Displays date values in a short date format determined by the Windows Control Panel short date setting.
- YL: Displays date values in a long date format determined by the Windows Control Panel long date setting.

To determine the date settings in Control Panel, double-click the Regional Settings icon in the Control Panel and select the Date Page Tab. Windows offers two different ways to present a date. For example, a short date might be *11/Nov/96*. A long date might take on the following format: *November 11, 1996*. Presenting dates in the long format to users in a read-only fashion is a relatively simple task (see Figure 4). Allowing users to enter dates in such a format is another story.

To illustrate how to use these new format settings, create a form with one text box and at least one other control that allows the text box to lose focus (see Figure 5). In the Format Property of the text box, assign a value of "YL". Finally, assign the value property of the text box a date value. When running the sample form, be sure to have the Regional Settings dialog box activated as well. To have the text box display the short date format, just change the format property setting to "YS".

When the long and short date formats are changed in the Regional Settings dialog box, Visual FoxPro will immediately reflect the changes.

John V. Petersen, MBA, specializes in the development of Visual FoxProbased solutions for business. John has written for publications in the United States and has been a speaker at user group meetings and at the 1996 Database Development Workshop in Richmond, Virginia. John is also a coauthor of Developing Visual FoxPro 5.0 Enterprise Applications from Prima Publishing. 610-651-0879, johnpetersen@mail.com.

SET KEY Tip

Barbara Peisch

SET KEY and SKIP/SKIP-1 Interactions

SET KEY is one of those commands I never found all that useful in version 2.x. I always preferred to use the BROWSE...KEY command instead, or other techniques with SCAN...ENDSCAN. When using grids in VFP, we no longer have that option. If you're not using views to limit the display of your grid you might find that SET KEY is the answer. Whether you're using SET KEY with a grid, SCAN, or browse, here's a tip from **Paul Maskens** you should consider.

When using SKIP and SKIP-1 there is a possible problem (not everyone I spoke to has been able to reproduce this) in FoxPro 2.6 for DOS and Windows, and Visual FoxPro; this example shows a solution in VFP but can easily be translated into 2.x code.

If you're using a VCR control to move through records, and you find records that don't match the key that's being shown, then you've found the problem.

Code such as the following, in the "Prior" and "Next" buttons, fails because BOF() isn't triggered properly when the key is set to a value that isn't in the first record in your table:

```
* PRIOR
LOCAL lcState, knRec

lcState = "MID"

SKIP -1
IF BOF()
   GO TOP
   lcState = "BOF"
ELSE
   lnRec = RECNO()
   SKIP -1
   IF BOF()
      lcState = "BOF"
   ENDIF
   GO lnRec
ENDIF

* Enable/disable buttons
* as necessary.
* (This method isn't
* provided in this
* example.)
This.Parent.SetBut(lcState)
```

```
* NEXT
LOCAL lcState, lnRec

lcState = "MID"

SKIP
IF EOF()
   GO BOTTOM
   lcState = "EOF"
ELSE
   lnRec = RECNO()
   SKIP
   IF EOF()
      lcState = "EOF"
   ENDIF
   GO lnRec
ENDIF

* Enable/disable buttons
* as necessary.
* (This method isn't
* provided in this
* example.)
This.Parent.SetBut(lcState)
```

Each IF BOF() and IF EOF() test also needs to test that the key field value in this record is between the range of the SET KEY values.

If you know the field name used in the SET KEY expression, then you can code around this by calculating the low and high values and using BETWEEN(). I called this InvalidKey() and so I

return .F. if it's a valid record:

```
llResult = .F.
lcChildField = THIS.ChildField
lnLowKey  = val(set("key"))
lnHighKey = val(substr(set("key"),at(",",set("key"))+1))
IF !between(eval(lcChildField),lnLowKey,lnHighKey)
   * the key value is not in the SET KEY range
   llResult = .T.
ENDIF
```

Then I changed my tests of BOF() and EOF() to "IF BOF() OR InvalidKey()", and "IF EOF() OR InvalidKey()".

This example works only for numeric keys, of course, but a little creative coding should get it to work for character keys as well.

I always use numeric keys in my applications, so I know that SET("KEY") returning " 1, 1" is referring to a range of numbers and not two strings. ▲

The Grand Scheme of Things

Les Pinter

Fifteen billion years ago, the singularity exploded once again, and once again the Universe was born. For the hundred trillionth time, cosmic dust rushed outward at the speed of light, eventually to congeal into whirling plasma spirals. As they cooled, spheres of nuclear fire appeared and the galaxies were born. Billions of years passed. Higher elements grew in the nuclear soup. Suns were born and died, their dust dispersing throughout the void and reforming into other star systems. In one of the billions of galaxies, a type G star spun its retinue of 10 planets. One was torn apart in the gravitational battle between its two giant neighbors, and formed a belt of asteroids. Four billion years later, one of these asteroids fell to Earth and killed all of the dominant saurian species. As a result, an insignificant amphibian survived and began a lineage that would end with the primates.

Our ancestors were a minor species of hunter-gatherers until 20,000 years ago. They scratched out an existence without much distinction, living as both hunter and hunted among the carnivores of the Earth. Then technology, in the form of basic agriculture, permitted a few members to devote their growing brains to something else. Art was born, then elementary ceramic and metal forming skills, then philosophy, then literature and mathematics. Religion caused half of the population to kill the other half 100 times over. Then, scarcely 200 years ago, the age of machines came upon mankind, and suddenly every human hunger had a mechanical solution. There was almost no problem that man's mind couldn't solve.

An English mathematician and his lover, the daughter of a poet, designed a device called a Difference Engine, which, like some looms of the time, could be given a program of instructions.

But this machine could perform calculations. The machine wasn't built for nearly 200 years, but finally, in the years following World War II, the first working model was constructed. It filled a building, and every seven minutes one of its 2,000 vacuum tubes burned out, crashing the machine. But it could replace the ultimate machine -- the human mind.

A fascinated world considered the possibilities. A 1953 issue of *Popular Mechanics* magazine predicted three things would occur by 1988: Everyone would have a helicopter; there would be atomic power generators in every back yard; and computers would be small enough to fit inside a single room. The first two predictions were wildly overzealous, but the third fell far, far short. The average microwave oven has a more powerful computer in it than did that first ENIAC machine. But it was a start.

The first computers were programmed in binary -- zeroes and ones. How privileged the first programmers must have felt when an assembler was built, and then a text editor to enter the mnemonics that represented the binary instruction set. COBOL was touted as a language that businesspeople could learn to write. We're still waiting for that to happen. But as good as COBOL was, better languages were developed. Little tiny BASIC, a 30-page assembly language program that compiles to a 4K .EXE, was translated to several computers. A 13-year-old in Seattle translated it to work on the Intel 8080 chip, and his father, a patent attorney, copyrighted it. He then signed an agreement with Intel to put BASIC on its chip and collect a $5 royalty for each chip sold. Intel sold 200 million chips. Every one of you is doing multiplication at this instant.

With that kind of money, Microsoft was able to buy and enhance technology. They bought an operating system, enhanced it, and marketed it as MS-DOS. They bought a word processor called the Magic Wand -- from *me* -- and brought out Microsoft Word two years later. They wrote a database called Access, then looked around at the competition and bought Fox Software. FoxPro 2.5, 2.6, 3.0, and 5.0 followed. At the Phoenix DevCon, Microsoft announced FoxPro version 6.0.

It's not a big deal in the cosmic scheme of things, but it ain't bad. It seems we're not going to die as scheduled. And evolution appears to be in our favor.

So keep the pressure on; I think we're winning.

Les Pinter publishes the Pinter FoxPro Letter in the United States and Russia and on the Internet at . His first novel, The Valley, will be out in the spring.