November 1996

Use the Windows API for Text Handling Inside VFP

Richard L. Aman

WIN 3.x DOWNLOAD *(1)*

The Windows clipboard can hold many different data formats and corresponding data handles, all representing the same data, but in as many different formats as an application is able to supply. For example, a pie chart might be held in the clipboard as both a metafile picture and a bitmap. An application pasting the pie chart would have to choose which representation best matched its requirements. In this article, Richard Aman discusses how to use the Windows API for cutting and pasting data instead of the internal Visual FoxPro cut and paste functions.

Visual FoxPro 3.0, like most Windows products, provides internal methods for copying to and pasting from the Windows clipboard. These internal functions move the selected text between the applications and the clipboard. When the user presses the appropriate hot-key combination (usually Ctrl-C to copy and Ctrl-V to paste), the internal functions move the selected objects to and from the clipboard automatically and under Windows control.

So why bypass the internal FoxPro text editing functions and use custom functions instead? I can think of three reasons: for security, data integrity, and interapplication communication. Let's discuss the circumstances where these concerns might be valid. However, the example form I provide deals only with the issue of data security.

Data security -- prevent sensitive data from being copied

Before users are allowed to copy or paste text it may be necessary to check the text that the users have selected to see if it contains any sensitive information. Ideally, the users aren't prevented from working with the text using the normal text editing functions while in the source application, but are prevented from moving the text to another application. For example, if users are in a memo field of a form that contains sensitive information, they should be able to use all normal editing functions, except they should not be allowed to move the data to a word processor or the Windows Notepad.

One way to accomplish this would be to encode the data at the time the users copy it and decode the data when it is pasted. If users paste the data in another application, the data would be useless without decoding.

Here are a couple of simple encoding/decoding routines that are used in the sample form included in download file _AMAN.EXE.

```
*-------------------------------------------------------
*-- Encode()
*-- This function is a simple text encoder.  All it does is
*-- scan the given text string and increment each ASCII code
*-- by one in the string, then return the encoded string.
*--
PARAMETERS tcText

*-- tcText is the text to be translated

*-- define variables
PRIVATE lcChar
PRIVATE lnLoop1
PRIVATE lcText

*-- init variables
lcChar = ''
lnLoop1 = 0
lcText = ''

*-- begin main process
FOR lnLoop1 = 1 TO LEN( tcText )
lcChar = SUBSTR( tcText, lnLoop1, 1 )
lcChar = CHR( ASC( lcChar ) + 1 )
IF ASC( lcChar ) = 129
lcChar = CHR( 32 )
ENDIF
lcText = lcText + lcCHar
ENDFOR

*-- clean up and return
RETURN( lcText )

*-------------------------------------------------------
*-- Decode()
*-- This function is a simple text decoder.  All it does is
*-- scan the given text string and decrement each ASCII code
*-- by one in the string, then return the decoded string.
*--

PARAMETERS tcText

*-- tcText is the text to be translated

*-- define variables
PRIVATE lcChar
PRIVATE lnLoop1
PRIVATE lcText

*-- init variables
lcChar = ''
lnLoop1 = 0
lcText = ''

*-- begin main process
FOR lnLoop1 = 1 TO LEN( tcText )
```

Data integrity -- validate the data before pasting

By bypassing the native FoxPro editing functions, checks can be performed on the data before it is pasted. For example, if users are in one field on a form, copy data to the clipboard, and then enter another field to paste the data, the data can be validated before the users paste the data in the new edit field rather than waiting for the users to exit the field and only then performing validation. The sooner users are made aware of incorrect data, the more likely they will still be focused on that particular data.

Inter-application communication using OLE automation

By using the Windows API clipboard functions for text editing, custom functions can be expanded to allow data to be passed between applications via the clipboard and OLE automation or e-mail. For example, users can select data from a field on a form and send the data to an Excel spreadsheet or a Word document for further processing. Or a custom e-mail message can be automatically generated and sent along its way.

The two main editing functions

Copying and pasting are the two main functions used when working with text data. To bypass Visual FoxPro 3.0's internal editing functions and use custom functions instead, the application has to gain direct access to the clipboard. I accomplish this with the help of several functions from the Windows API and two wrapper functions I wrote to encapsulate the functionality of the Windows API.

The implementation involves trapping the native Ctrl-C and Ctrl-V keypresses in a form and replacing them with these custom functions. This is done in the Load event of the form and is explained later in this article. First, however, I'll discuss these custom functions -- CopyText(), a replacement for Ctrl-C, and PasteText(), a replacement for Ctrl-V.

CopyText()

```
*********************************************************
*  FUNCTION CopyText
*********************************************************
*  Author..........: Richard L. Aman
*  E-Mail..........: 73700.141@compuserve.com
*  Project.........: Clipboard Usage Within VFP 3.0
*  Created.........: 05/27/96  16:31:45
*  Copyright.......: (c) 1996, Aman Data Systems
*) Description.....: This program demonstrates the steps
*)                  : necessary to copy data to the clip-
*)                  : board using Windows API function calls
*  Calling Samples.: =CopyText( tcSelectedText )
*  Parameter List..: tcSelectedText - the text to copy
*  Change list.....:

LPARAMETERS tcSelectedText

*-- define variables
LOCAL lcSelectedText
LOCAL GHND
LOCAL CF_TEXT
LOCAL MAXSIZE
LOCAL hGlobalMemory
LOCAL lpGlobalMemory
LOCAL lpClipMemory
LOCAL hClipMemory

*-- init variables
lcSelectedText = ''
GHND = 66
CF_TEXT = 1
MAXSIZE = 4096
hGlobalMemory = 0
lpGlobalMemory = 0
lpClipMemory = 0
hClipMemory = 0

* declare Windows API functions for this module
DECLARE INTEGER GlobalUnlock IN kernel32 INTEGER
DECLARE INTEGER GlobalLock IN kernel32 INTEGER
DECLARE INTEGER GlobalAlloc IN kernel32 INTEGER, INTEGER
DECLARE INTEGER lstrcpy IN kernel32 INTEGER, STRING
DECLARE INTEGER OpenClipboard IN user32 INTEGER
DECLARE INTEGER EmptyClipboard IN user32
DECLARE INTEGER CloseClipboard IN user32
DECLARE INTEGER SetClipboardData IN user32 INTEGER, INTEGER

* Check to see if ther's anything to copy to the clipboard
IF LEN( tcSelectedText ) = 0
 WAIT WINDOW "Nothing to copy right now"
 RETURN
ENDIF

*-- NOTE: the following is only to be used
*-- if it is necessary to encode the text
*-- before it is copied to the clipboard
```

Let's assume that the application needs to transfer a character string to the clipboard. First, allocate a moveable global memory block of LEN(lcSelectedText) size. Include room for a terminating NULL:

```
hGlobalMemory = GlobalAlloc( GHND, ;
  LEN( lcSelectedText ) + 1 )
```

The value of *hGlobalMemory()* will be NULL if the block couldn't be allocated. If the allocation is successful, lock the block to get a far pointer to it:

```
lpGlobalMemory = GlobalLock( hGlobalMemory )
```

Copy the character string into the global memory block. Since the string is NULL-terminated, the application can use the Windows string-copy function lstrcpy():

```
lpClipMemory = lstrcpy( lpGolbalMemory, ;
  lcSelectedText )
```

The terminating NULL doesn't need to be added because the GHND flag for *GlobalAlloc()* zeroes out the entire memory block during allocation. Unlock the block:

```
GlobalUnlock( hGlobalMemory )
```

Now the application has a global memory handle that references a memory block containing the NULL-terminated text. To get this into the clipboard, open the clipboard:

```
OpenClipboard( 0 )
```

It's always a good idea to clear the clipboard to ensure that no data is left over from the previous use:

```
EmptyClipboard()
```

Copy the text to the clipboard by giving the clipboard the global memory handle using the CF_TEXT identifier:

```
hClipMemory = SetClipboardData( CF_TEXT, ;
  hGlobalMemory )
```

When finished, close the clipboard to allow other applications to use the clipboard:

```
CloseClipboard ()
```

All done. The text has been transferred to the clipboard and is accessible from this and other applications.

Some notes concerning copying to the clipboard

Call OpenClipboard() and CloseClipboard()within the same procedure. Don't leave the clipboard open when the procedure exits. Don't let control transfer to another program while the clipboard is open.

Don't give the clipboard a locked memory handle.

After the call to SetClipboardData(), don't continue to use the global memory block. It no longer belongs to the application, and the handle should be treated as invalid. If the application needs to continue to access the data, it should make another copy of it or read it from the clipboard (as described in the next section). The application can also continue to reference the global memory block between the SetClipboardData() call and the CloseClipboard() call, but must use the global handle that is returned from SetClipboardData(). Be sure to unlock this handle before the call to CloseClipboard():

PasteText()

```
*******************************************************
*  FUNCTION PasteText
*******************************************************
*  Author.........: Richard L. Aman
*  E-Mail.........: 73700.141@compuserve.com
*  Project........: Clipboard Usage Within VFP 3.0
*  Created........: 05/27/96  16:31:45
*  Copyright......: (c) 1996, Aman Data Systems
*) Description....: This program demonstrates the steps
*)                : necessary to paste data from the clip
*)                : board using Windows API function calls
*  Calling Samples.: =PasteText()
*  Parameter List..:
*  Change list.....:

*-- define variables
LOCAL CF_TEXT
LOCAL MAXSIZE
LOCAL hClipMemory
LOCAL lpClipMemory
LOCAL lcMyString
LOCAL lnRetVal

*-- init variables
CF_TEXT = 1
MAXSIZE = 4096
lpClipMemory = 0
hClipMemory = 0
lcMyString = ''
lnRetVal = 0

*-- declare Windows API functions for this module
DECLARE INTEGER OpenClipboard IN user32 INTEGER
DECLARE INTEGER CloseClipboard IN user32
DECLARE INTEGER GetClipboardData IN user32 INTEGER
DECLARE INTEGER GlobalAlloc IN kernel32 INTEGER, INTEGER
DECLARE INTEGER GlobalLock IN kernel32 INTEGER
DECLARE INTEGER GlobalUnlock IN kernel32 INTEGER
DECLARE INTEGER lstrcpy IN kernel32 STRING @, INTEGER
DECLARE INTEGER GlobalSize IN kernel32 INTEGER, INTEGER
DECLARE INTEGER IsClipboardFormatAvailable IN user32 INTEGER

* check to see if there is any text available for pasting
IF IsClipboardFormatAvailable( CF_TEXT ) <> 1
 WAIT WINDOW "No text available in the ;
   clipboard right now."
 RETURN
ENDIF

*-- try to open the clipboard
IF OpenClipboard(0) = 0
 WAIT WINDOW "Could not open the Clipboard. ;
   Another application may have it open"
 RETURN
ENDIF
```

Getting text from the clipboard is only a little more complex than transferring text to the clipboard. First, the application must determine whether the clipboard does in fact contain data in the CF_TEXT format. One of the easiest methods is to use this call:

```
bAvailable = IsClipboardFormatAvailable( CF_TEXT )
```

This function returns TRUE (nonzero) if the clipboard contains CF_TEXT data. IsClipboardFormatAvailable() is one of the few clipboard functions that can be used without first opening the clipboard. However, if the clipboard is opened later to get this text, the application should check again (using the same function or one of the other methods) to determine if the CF_TEXT data is still in the clipboard. To transfer the text out, first open the clipboard:

```
OpenClipboard( 0 )
```

Obtain the handle to the global memory block referencing the text:

```
hClipMemory = GetClipboardData( CF_TEXT )
```

This handle will be NULL if the clipboard doesn't contain data in the CF_TEXT format. This is another way to determine if the clipboard contains text. If GetClipboardData() returns NULL, close the clipboard without doing anything else.

The handle received from GetClipboardData() doesn't belong to the application -- it belongs to the clipboard. The handle is valid only between the GetClipboardData() and CloseClipboard() calls. The application can't free that handle or alter the data it references. If the application needs to have continued access to the data, it should make a copy of the memory block.

Lock the handle to the clipboard memory:

```
lpClipMemory = GlobalLock( hClipMemory )
```

Because the character string is NULL-terminated, the data can be transferred using Windows lstrcpy() function:

```
lstrcpy( @lcMyString, lpClipMemory )
```

Unlock the global memory handle:

```
GlobalUnlock( hClipMemory )
```

Peel off the terminating NULL:

```
lcMyString = SUBSTR( lcMyString, 1, AT( CHR(0), ;
  lcMyString ) - 1 )
```

Finally, close the clipboard:

```
CloseClipboard()
```

Now the application has the clipboard text in the variable lcMyString for use within the application.

A few other considerations for clipboard use

Here are a few other considerations when using the Windows API functions to access the clipboard from within a Visual FoxPro application:

Only one program can open the clipboard at any time. The purpose of the OpenClipboard() call is to prevent the clipboard contents from changing while a program is using the clipboard. OpenClipboard() returns a logical value indicating whether the clipboard was successfully opened. It won't be opened if another application failed to close it. During the early stages of programming for the clipboard, it's probably wise to check this value, but the check isn't crucial in a nonpreemptive multitasking environment. If every program politely opens and then closes the clipboard during a single procedure without giving control to other programs, then being unable to open the clipboard should never be a problem.

Another subtle problem to avoid involves message boxes: If a global memory segment can't be allocated, the system should normally display a message box. If this message box isn't system modal, however, users can switch to another application while the message box is displayed. Either the message box should be a system modal message box, or the clipboard should be closed before the message box is displayed.

Dialog boxes can be another source of trouble: If the clipboard is left open while displaying a dialog box, remember that the edit fields in a dialog box also use the clipboard for cutting and pasting text.

Also, if the application encodes and decodes data while copying and pasting, this will also prevent users from copying from another source and pasting into the application unless the decoding function is bypassed when pasting the text.

Using the functions on a form

Here's an example of how to use these functions on a form (this form is also included as part of download file _AMAN.EXE):

1. Start Visual FoxPro 3.0 and create a new form.

2. Add four new methods to the form and name them as follows:

- **CopyText** -- this will hold the code to call when

Ctrl-C is pressed.

- **PasteText** -- this will hold the code to call when

Ctrl-V is pressed.

- **Encode** -- this will hold the code to encode the

clipboard text.

- **Decode** -- this will hold the code to decode the

clipboard text.

3. Copy the code for CopyText into the method.

4. Copy the code for PasteText into the method.

5. Copy the code for Encode into the method.

6. Copy the code for Decode into the method.

7. In the Load() method of the form, add the following two lines of code:

```
SET SYSMENU OFF
SET SYSMENU TO
```

This disables Visual FoxPro's internal copy and paste methods and will allow the new CopyText() and PasterText() functions to be called when Ctrl-C and Ctrl-V are pressed.

In the form's UnLoad() method add the following line of code:

```
SET SYSMENU TO DEFAULT
```

This resets the system menu back to its default.

Add two edit boxes to the form and resize them to a comfortable width for working with lines of text. In the KeyPress() method of the first edit box, add the following three lines of code:

```
IF nKeyCode = 3 AND nShiftAltCtrl = 2
  thisForm.CopyText( this.SelText )
ENDIF
```

In the KeyPress() method of the second edit box, add the following four lines of code (the keyboard command is included to get rid of a spurious chr(14) that the API call wants to paste in along with the text):

```
IF nKeyCode = 22 AND nShiftAltCtrl = 2
  this.Value = thisForm.PasteText()
  keyboard "{backspace}"
ENDIF
```

Test out the clipboard functions:

1. Save and run the form.

2. Type some text in the first edit box.

3. Highlight the text in the first edit box.

4. Press Ctrl-C to copy the highlighted text to the clipboard.

5. Tab to the second edit box.

6. Press Ctrl-V to paste the text from the clipboard to the second edit box.

7. Start Notepad.

8. Press Ctrl-V to paste the text from the clipboard to the Notepad. (The text should be encoded.)

Conclusion

In the wonderful world of software development, there are usually as many ways to perform a function as there are developers. I've presented another way of copying and pasting text in Visual FoxPro 3.0 via the clipboard and the Windows API. By intercepting the text before it is copied to the clipboard and after it is copied from the clipboard, all sorts of operations can be performed on the text. Using the Windows API functions can open up a whole new world and give the developer more control over the application from within Windows. Every time I start digging around in the Windows API, I always seem to stumble upon some useful function or two that makes programming easier, more interesting, and more enjoyable. The documentation about the Windows API functions on the Developer's Network Library and Technet CDs are an excellent source of information on the inner workings of the Windows API. I've come to appreciate what great tools the Windows API offer developers as we create Windows applications. The Windows operating system now does a lot of work for us that we used to have to do ourselves.

DOWNLOAD

Richard Aman is director of software engineering at Loren Industries, a jewelry manufacturing company in Hollywood, Florida. Richard has been developing business solutions with FoxBase/FoxPro since 1988 and regularly gives presentations at his local Fox User Group. 73700.141@compuserve.com.

It's Not That Sexy -- Well, OK, It Actually Is

Whil Hentzen

Most guys, when they get home from a really long day at work, probably settle down with a copy of *Sports Illustrated*, *Golf Digest*, *Road and Track,* or *Field and Stream*. A few of you probably go for the new copy of *PC Magazine*, or perhaps *Wired*, if you're trying to fruitlessly maintain some semblance of youth. Over the past month, I've blown off every magazine I get, instead choosing to spend my few free minutes with a book by G. Pascal Zachary entitled *Showstopper!*, the saga that details the creation of Windows NT. My wife has long stopped being curious why I'm spending late evenings reading about a new operating system, but every once in awhile I'll step outside myself and ponder if it's normal. Then I'll frantically turn the page, because I can't go to bed until I find out how the latest cliffhanger comes out -- whether Allen was able to quash that bug in the graphics icon rewrite routine in time for the final build for Beta 1.

The fascinating part of this book, however, is not learning whether the team is going to go with the new, radical file system (they did) or if they could get it to run in 8M (they didn't). Rather, it's the stories of the people behind the scenes and how they built a rock-solid, robust (if somewhat memory hungry and sluggish) operating system out of a mishmash of white board scribbles, broken dreams from other companies, and long-term guesses about where the industry was headed. If you pick up this tome, you'll meet folks like Dave Cutler and Eric Fogelin. Cutler, a fossil in terms of relative age, but described by a number of people as knowing more about operating systems than anyone else on the planet, ran the entire NT project. Fogelin, who applied for a technical support job at Microsoft because he was knocking around the West Coast and was running low on cash, moved a cot into his office because he lived 90 minutes away from Redmond and he'd committed to producing 13 manuals for a release less than eight months away.

As you get engrossed in the story, you get swept up by the passion, zeal -- and panic -- felt by these folks as they wrestled with a program that was, arguably, the most demanding program ever to run on a PC platform. You watch as deadline upon deadline slips, due to miscommunicated specs, feature creep, limited resources, and unanticipated technical hurdles. You live with them as they drop hobbies, lose friends, have their electricity and water turned off because they forgot to pay bills, and find their marriages breaking up -- all in the quest of the operating system of the next decade.

And you grow more and more envious of them. It sounds grand, romantic, exciting, and yes, in this day and age, sexy. And we're just like them. We're the last of the cowboys -- the adventurers

of the electronic frontier (sorry, that *does* sound putrid, doesn't it?). We're the only ones bragging about 20-hour days, leaving work one day and finding out the United States invaded another country -- three days ago -- and having weeks measured by empty cases of Jolt and discarded Twinkie wrappers. Les Pinter put it very eloquently a year or so ago: Bankers and lawyers don't go to parties and brag about staying up till 4 a.m. working on a loan or rewriting an appeal, but they'll strut around like the cock of the walk after cobbling together a program and three screens that seem to work more than 90 percent of the time.

As I write this editorial, the longest unbroken string of sleep I've had in about a week has been five hours. Last night was worse than most, but still typical: calls at 7 p.m., 9:30, midnight, 2 a.m., 5:30, 8, and 9:30. The application is a complete rewrite of a mission critical (that means 24 hours a day, remotely accessible by approximately 25,000 sites around the country) application tied to seasonal events. In other words, the deadline wasn't imposed by some power-hungry bureaucrat who picked some arbitrary date, but was due to external constraints that couldn't be altered. The customer's previous developer bailed on them four weeks before the app was to go live, and they were left with partial specs and a previous version that was your basic model of how to keep things a secret. To compound matters, a cornerstone of the application involved importing a number of data files from various sources, and the suppliers of those data files were late with the file layouts and made numerous changes to the layouts over the four weeks.

Well, that's what we do: write code and ship apps. And here was another opportunity for adventure -- the 20-hour days, the cot in the office, and a chance to help improve Coca-Cola's profits again. I shifted workloads around so that the whole shop was able to help out in the effort of getting this guy out the door.

After seeing John nearly fall asleep on his feet after two consecutive 16-hour days at the office (he has a 70-minute drive home), and then watching Shauna leave work Monday night (well, it was really Tuesday morning) at about 2:20 with enormous bags under her eyes, I started thinking that maybe this wasn't that exciting after all. It looked pretty cool, Clint Eastwood riding into town, shooting the bad guys, getting the girl (for a while, at least), and then riding into the sunset with a fistful of dollars. But you didn't see the four days he spent on a horse that smelled better than he did. His festering blisters weren't obvious, either. And the breakfast, lunch, and dinner of canned beans just didn't seem as funny as they did in *Blazing Saddles*. The romance isn't quite there when the mercury doesn't dip below 95 and a rusty canteen of warm water is the only drink you'll have until the weekend.

So, in retrospect, it didn't seem that cool to bill 88 hours in five days. We couldn't figure out where the fun was. It was just hard work and we're all exhausted.

But I'll wait, and we'll swap stories in a couple of weeks. We'll have our little in-jokes and sly references to goofy customers with outlandish expectations. Despite our best attempts at describing the events, the new people coming on board are going to wonder what they missed. They'll have to wait till the next customer emergency to become indoctrinated into the secret society of software development.

I think maybe I won't check out who's hiring bankers in the Sunday paper after all.

Data Handling Issues

Doug Hennig

UNIX MAC DOS WIN 2.x 3.x

DOWNLOAD *(2)*

The ability to handle multiple sets of data is a frequent requirement in business applications. So is the management of primary key values for tables. In the first section of this article, Doug takes on these issues and looks at the best way to deal with them. Next, he provides a solution to a common problem with field and table validation rules: While these rules protect your tables from invalid data, they also make data entry forms harder to use. Finally, Doug examines the use of multipurpose lookup tables, and discusses how to take advantage of new data dictionary features in VFP 5.0.

FoxPro is a database management system, so data handling is the thing it does best. However, many issues related to data handling can complicate even the simplest application. This article will look at common data handling issues. In the first section, we'll explore some strategies for handling multiple sets of data (such as test and production versions) in FoxPro 2.x and Visual FoxPro (VFP), and look at several problems that come up when you use primary keys and how to solve these problems.

Handling multiple sets of data

The idea of multiple sets of data comes up frequently in database applications. For example, accounting applications usually allow the user to manage more than one company by having each company's data stored in a separate set of tables. Even simpler applications benefit from having multiple data sets by providing test and production versions of the tables. This way, inexperienced users can learn how to use the system or new features can be tested against test data without worrying about ruining production data.

Multiple data sets are handled in one of two ways: the tables exist in a single directory with different table names for different sets, or the tables in different sets have the same names but are located in different directories. An example of the former method is an accounting system that has table names such as GLACCT<nn> and ARCUST<nn>, where <nn> is a two-digit value representing a data set number. The disadvantage of this mechanism is that the directory will quickly accumulate a lot of files, which makes managing the files more complex and causes file searches (in DOS, anyway) to dramatically slow down. As a result, the latter mechanism is preferred.

Working with multiple data sets is relatively easy. Usually you'll have a set of tables common to the application regardless of the data set (for example, security or application configuration tables) and a set of tables specific to each data set. The common tables might be in the same directory as the application or perhaps in a DATA or COMMON subdirectory, while the data set tables will exist in one or more separate directories. In a simple case, such as test and production

data, the directory names might be hard-coded in the application, such as DATA and PRODUCTION. In more complex applications, such as multi-company data sets, you might have a DATASET table (in the common set) with at least two fields: the name of the company and the directory where the company's data set is located.

Here's how selecting a data set works:

- Close any currently open tables that aren't in the common set.
- Decide which directory the desired data set is stored in. For example, you might have a menu item that switches between test and production data. For a multi-company application, the user might select a company to work with from a dialog box, and your program would look up the company name in the DATASET table to find the directory that company's data set is stored in.
- Open the tables in that directory.

In FoxPro 2.x, the specifics of opening the tables can be handled in a variety of ways. You can SET DEFAULT TO or SET PATH TO the specified directory and open the tables, your table-opening routine can accept a directory parameter and open the tables in that directory, or you can use a data-driven approach. Using SET DEFAULT TO and SET PATH TO probably aren't the best ways to go unless your application is very simple, because it makes locating other pieces of the application more complex. You can easily create a table-opening routine that handles both common and data set tables. The following routine does just that (this example is for illustration purposes only; a real routine would include parameter and error checking):

```
parameters tcDirectory
if tcDirectory = 'COMMON'
use SECURITY in 0
use USERS in 0
use CONFIG in 0
use DATASET order COMPANY in 0
else
if used('CUSTOMER')
use in CUSTOMER
endif used('CUSTOMER')
use (tcDirectory + 'CUSTOMER') in 0
if used('ORDERS')
use (tcDirectory + 'ORDERS') in 0
endif used('ORDERS')
if used('ACCOUNTS')
use (tcDirectory + 'ACCOUNTS') in 0
endif used('ACCOUNTS')
endif tcDirectory = 'COMMON'
```

A data-driven table-opening routine is a better idea. You create a table (perhaps called TABLMAST) containing the name of each table in your application and a flag indicating whether the table is common or not. The data-driven routine opens each table defined in TABLMAST rather than hard-coding the names of the tables to open. This routine, called

OPENDBFS.PRG, is in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP:

```
parameters tcDirectory
private lcTable
select TABLMAST
scan
  lcTable = trim(TABLMAST.TABLE)
  do case
    case tcDirectory = 'COMMON' and TABLMAST.COMMON
      use (lcTable) in 0
    case tcDirectory <> 'COMMON' and ;
      not TABLMAST.COMMON
      if used(lcTable)
        use in (lcTable)
      endif used(lcTable)
      use (tcDirectory + trim(TABLMAST.TABLE)) in 0
  endcase
endscan
```

At application startup, your program would use the following code to open the common tables:

```
do OpenDBFS with 'COMMON'
```

If you allow the user to choose a company to work with, the following code would create an array of companies defined in DATASET:

```
select COMPANY ;
from DATASET ;
into array laCompany ;
order by 1
lcCompany = GetComp(@laCompany)
if not empty(lcCompany)
select DATASET
seek upper(lcCompany)
do OpenDBFS with trim(DIRECTORY)
endif not empty(lcCompany)
```

Ask the user to choose one (using a routine called GETCOMP.PRG, which isn't shown here) and then open the tables for the selected company.

As usual, things aren't quite as simple in Visual FoxPro. The problem is that the database container (DBC) contains a hard-coded directory reference to each table, and in each table the location of the DBC it belongs to is hard coded in its DBF header. This means you can't have a single DBC for the tables in multiple data sets. While you might be tempted to use free tables in this case, you lose the advantages a DBC provides, including field and table validation, referential integrity, and transaction processing. Although you can write a program that changes

the directory to a table in the DBC, that won't work in a multi-user environment since when you do that you change the directory to a table for all users.

The only realistic solution is to put a copy of the DBC and its tables in each data set directory and then open the appropriate DBC when you want to work with a data set. "What about common tables?" I'm sure you're thinking. You'll need to create at least two DBCs for the application: one for the common tables and one for the data set tables. The common DBC and its tables go in your common table location.

Here's a VFP version of OpenDBFS called OpenData. It uses a data-driven approach because you can get a list of the tables for each database using ADBOBJECTS() (this program is in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP):

```
lparameters tcDirectory
local lcDirectory, ;
laTables[1], ;
lnTables, ;
lnI, ;
lcTable
if tcDirectory = 'COMMON'
open database COMMON
else
if dbused('DATA')
set database to DATA
close database
endif dbused('DATA')
lcDirectory = alltrim(tcDirectory)
lcDirectory = lcDirectory + ;
iif(right(lcDirectory, 1) = '\', '', '\')
open database (lcDirectory + 'DATA')
endif tcDirectory = 'COMMON'
lnTables = adbobjects(laTables, 'Table')
for lnI = 1 to lnTables
use (laTables[lnI]) in 0
next lnI
```

See the README.TXT file on the Developer's Disk for examples of using OpenData with the two data sets in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP (in the COMPANY1 and COMPANY2 subdirectories).

An additional complication with VFP is that the DataEnvironment of forms and reports has a hard-coded reference to the DBC that each table in the DataEnvironment belongs to. Fortunately, the Database property of each Cursor object in the DataEnvironment can be changed at runtime. The following code can be placed in the BeforeOpenTables method of the DataEnvironment. It assumes that the name of the directory for the database is stored in a global variable called gcDirectory and changes the location of the database for every table not in the COMMON database to that directory. Of course, to avoid using global variables you'd probably store the location of the current database in a property of your application class instead:

```
if type('gcDirectory') = 'C'
  lnEnv = amembers(laEnv, This, 2)
  for lnI = 1 to lnEnv
    oObject = evaluate('This.' + laEnv[lnI])
    if upper(oObject.BaseClass) = 'CURSOR' and ;
      not 'COMMON' $ upper(oObject.Database)
      lcDBC = oObject.Database
      oObject.Database = fullpath(gcDirectory + ;
        substr(lcDBC, rat('\', lcDBC)), curdir())
    endif upper(oObject.BaseClass) = 'CURSOR' ...
  next lnI
endif type('gcDirectory') = 'C'
```

The Employee form in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP shows an example of using this mechanism. To test this from the Command window, initialize gcDirectory to either COMPANY1 or COMPANY2 (the two subdirectories containing data sets), and DO FORM EMPLOYEE. You'll see a different set of records for each directory.

Several maintenance issues are more complicated when you provide multiple data sets to an application. For example, your "re-create indexes" routine needs to re-create indexes for all data sets. When you install a new version of an application that has database changes from the existing version, you need to update the DBC and table structures for each data set. These aren't difficult issues, just ones you need to be prepared for.

Primary key issues in VFP

VFP automatically ensures that records have a unique key value. You simply define one of the tags as a primary index, and VFP will automatically prevent two records from having the same key value. Let's look at some issues related to primary keys.

The primary key for a table can be of any data type, but Integer and Character are the most common choices. I prefer Integer primary keys for the following reasons:

- They take only four bytes.
- You can have up to 2 billion keys (or 4 billion if you use negative values as well).
- No data conversion is required when incrementing the next available value.
- SQL SELECT statements that join two tables perform fastest using Integer keys.

Character keys have two advantages over Integer:

- You can't easily concatenate two Integer values when creating compound primary keys (a compound key is one that consists of more than one field). However, painful experience has led me to avoid using compound keys whenever possible, so this isn't usually a concern.
- Combo boxes used to select a value from a related table are more complicated to work

with when the primary key for the related table isn't Character.

There are two kinds of primary keys for a table: user-defined and system-defined (also known as "surrogate"). I prefer surrogate keys that the user never sees because they avoid all kinds of complications like cascading primary key changes (because the primary key never changes, there's no need to cascade it), especially using the current Referential Integrity Builder, which doesn't properly handle compound primary keys.

A simple routine such as the following can be used to assign the next available value to the primary key for a table. This routine (called NEXT_ID.PRG in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP) works with both Character and Integer (or other numeric) keys. It looks up the next available key value for the specified table in a table called NEXTID, which consists of two fieldsTABLE C(128) and ID I(4):

```
lparameters tcTable
local lnCurrSelect, ;
llUsed, ;
lnCurrReprocess, ;
luKey, ;
lcKey, ;
lcField

* Save the current work area, open the NEXTID table
* (if necessary), and find the desired table. If it
* doesn't exist, create a record for it.

lnCurrSelect = select()
llUsed       = used('NEXTID')
if llUsed
select NEXTID
set order to TABLE
else
select 0
use NEXTID order TABLE again shared
endif llUsed
seek upper(tcTable)
if not found()
insert into NEXTID values (tcTable, 0)
endif not found()

* Increment the next available ID.

lnCurrReprocess = set('REPROCESS')
set reprocess to 10 seconds
if rlock()
replace ID with ID + 1
luKey = ID
unlock
endif rlock()

* Set the data type of the return value to match
* the data type of the primary key for the table.

lcKey = dbgetprop(tcTable, 'Table', 'PrimaryKey')
if not empty(lcKey)
lcField = key(tagno(lcKey, tcTable, tcTable), ;
tcTable)
if type(tcTable + '.' + lcField) = 'C'
luKey = str(luKey, fsize(lcField, tcTable))
endif type(tcTable + '.' + lcField) = 'C'
endif not empty(lcKey)

* Cleanup and return.

set reprocess to lnCurrReprocess
if not llUsed
use
endif not llUsed
select (lnCurrSelect)
return luKey
```

Where should the code in NEXT_ID go? The place that immediately comes to mind is in the stored procedures for the database. The advantage of putting the code there is that the database becomes self-contained; you don't need to ship a separate PRG to someone in order to use your database. However, because NEXT_ID is likely to be used in every database you create, this raises a maintenance issue. What if you discover a bug in NEXT_ID or want to enhance its functionality? You'd have to change it in every database you ever created, including those at every client site. In addition, if you succumbed to the temptation to tweak the code slightly in different databases because you had different needs in each, you'd really have a tough time updating the code for each one. In my opinion, only database-specific code (such as validation and other business rules) belongs in the stored procedures of a database. Generic routines such as NEXT_ID belong in stand-alone PRGs or in a library of routines (whether procedural or in a class library).

When should you call NEXT_ID to assign a surrogate key to a record? The obvious place is in the insert trigger for the table, but as you've probably discovered by now, VFP won't allow you to change the contents of any field in the current table in trigger code. You could call it in the Save code for your form, but that breaks encapsulation for the database -- the key value won't be assigned properly when records are added any place other than the form. The correct place to assign the primary key is in the Default value for the primary key field; you do this by specifying NEXT_ID('<table>') as the expression, where <table> is the name of the table. Whenever a new record is added, VFP will call NEXT_ID to assign a default value to the primary key field.

What happens if the user adds a new record to the table, then cancels the addition? Using buffering and TABLEREVERT(), you can easily discard the added record, but the KEY field in NEXTID.DBF is still incremented. If you're using meaningless surrogate keys, this isn't a problem, but if the primary key is a check or invoice number, you might not want to "waste" a primary key value. To prevent this, you can have your forms work on a view of the table. Only when TABLEUPDATE() is used to write the new view record to the table is NEXT_ID called; if the user discards the new view record, the table isn't affected so the next available key value isn't incremented.

Many developers minimized the requirement to PACK a table in FoxPro 2.x applications by recycling deleted records. This simple yet powerful idea works as follows:

- When a record is deleted, blank the fields in the record (using BLANK in FoxPro 2.6 or replacing each field with a blank value in prior versions). This causes the record to "float" to the top when a tag is active.
- When a new record is needed, first SET DELETED OFF (so FoxPro can "see" deleted records), then GO TOP (or LOCATE) to move to the first record in index order and check to see if it's deleted. If so, RECALL the record. If not, there are no deleted records to recycle, so use INSERT INTO or APPEND BLANK to create a new record.

This mechanism doesn't work in VFP for one simple reason: you can't have two records with the same primary key value, even a blank value. Even deleted records are checked for primary key

duplication, so as soon as you try to delete a record and blank its key value when another such record already exists, you'll get the dreaded "Uniqueness of index is violated" error.

There are two solutions to this problem. One, a technique pioneered by the late Tom Rettig, involves using a filter on the primary key of NOT DELETED(). With such a filter, VFP no longer checks deleted records for duplicate primary keys, so this mechanism works. Unfortunately, this filter can only be created visually using the Table Designer, not programmatically using INDEX ON or ALTER TABLE, so you can't re-create this index at a client site if indexes need to be rebuilt. The other solution is to create a tag on DELETED() (which is a good idea anyway because it helps Rushmore do its job better) and use that tag to locate a deleted record to reuse. The logic behind this technique is now as follows:

- When a record is deleted, don't blank the fields; just delete the record. This causes the record to "float" to the set of deleted records when the DELETED tag is active but keeps its primary key value so the record is unique.
- When a new record is needed, first SET DELETED OFF (so FoxPro can "see" deleted records), then set the order to the DELETED tag and go to the first record. If it's deleted, RECALL it and assign a new primary key (if desired). If not, there are no deleted records to recycle, so use INSERT INTO or APPEND BLANK to create a new record.

The following routine (NEW_REC.PRG in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP) can be used to recycle deleted records:

```
local lcCurrDelete, ;
lcCurrOrder, ;
llDone, ;
lcBlank, ;
laFields[1], ;
lnFields, ;
lnI, ;
lcField, ;
lcDefault

* Save the current DELETED setting and order and
* set them as we require.

lcCurrDelete = set('DELETED')
lcCurrOrder  = order()
set deleted off
set order to DELETED descending

* Go to the top of the file, which should be the
* first deleted record if there are any. Then
* process records until you get what you want.

locate
llDone = .F.
do while not llDone
do case

* If this record is deleted and you can lock it, set
* each field to its default value, blank any fields
* you don't have a default for, and recall the
* record.

    case deleted() and rlock()
      lcBlank  = ''
      lnFields = afields(laFields)
      for lnI = 1 to lnFields
        lcField   = laFields(lnI, 1)
        lcDefault = laFields(lnI, 9)
        if empty(lcDefault)
          lcBlank = iif(empty(lcBlank), 'fields ', ;
            lcBlank + ',') + lcField
        else
          replace (lcField) with evaluate(lcDefault)
        endif empty(lcDefault)
      next lnI
      blank &lcBlank next 1
      recall
      llDone = .T.

* If this record is deleted but you can't lock it,
* try the next one.

    case deleted()
      skip

* If this record isn't deleted, there aren't any
```

Hassles with validation

In FoxPro 2.x, most developers used a data buffering scheme involving memory variables. The idea was to use SCATTER MEMVAR to create a set of memory variables with the same names as the fields in a table, use GETs to edit those memory variables in a screen, and then save the changes using GATHER MEMVAR only if the user chose the Save function. VFP's built-in data buffering means you can now create forms that edit the fields in a table directly, and either save the changes using TABLEUPDATE() or cancel them using TABLEREVERT(). Unfortunately there's one flaw in this new scheme: validation rules.

Field and table validation rules act like soldiers guarding your data. VFP won't allow any data that disobey these rules to be stored in your tables. However, these soldiers seem to be a bit overzealous: there's no way to control when they fire, nor is there a way to trap the error that occurs when the rules fail. Even with table buffering turned on, VFP checks a field validation rule when the user tries to exit a control bound to that field, and a table validation rule when the user tries to move to another record. If the rule fails, VFP displays either the error message you defined in the database for the rule or a generic error message. In either case, you don't have any control over the appearance of the error dialog box. You also can't defer validation checking to a more convenient time. For example, if a user enters invalid information in a field, then clicks the Cancel button in a form, the user is still going to get an error message because there's no way to suppress the validation checking.

A less restrictive problem is system-assigned primary key values. Because VFP doesn't allow you to assign a primary key value in a trigger (which is the logical place to do it), most developers do it in the Default property for the primary key field by specifying a function name. For example, the routine called NEXT_ID.PRG in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP assigns the next ID for the specified table by incrementing the ID field in NEXTID.DBF. The default property for the CUST_ID field in the CUSTOMER table is NEXT_ID ('CUSTOMER'). When a new record is added to the CUSTOMER table, NEXT_ID is called by VFP to increment NEXTID.ID and assign the new value to CUST_ID. The only problem with this mechanism is that if the user decides not to add the record after all, NEXTID.ID has already been incremented. In the case of surrogate keys (keys that have no meaning other than to provide a unique value), this isn't a problem, but if the key represents the next invoice or check number, you'll have a "hole" in the numbering scheme.

To see an example of how these problems can be a pain, DO the CUSTOMER1 form in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP. This form uses row buffering with controls directly bound against the CUSTOMER table. Try to enter "NJ" into the Region field; the error dialog that appears is controlled by VFP, not the form. Try clicking the Cancel button. You'll continue to get the same error message until you enter a valid value (blank, SK, CA, or NY) into the field. Click the Add button, then the Cancel button. Do this several times. Even though no new records have been added to the table, the next available value for CUST_ID keeps rising.

The solution to these problems is to edit an updatable view rather than the table directly in a

form. Because the view won't have field validation rules (unless you explicitly define them, of course), users can enter whatever they like into any field. However, when they click the Save button, TABLEUPDATE() tries to write the view record to the table, at which time field and table rules are checked. If any rules fail, VFP doesn't display an error message; instead, TABLEUPDATE() returns .F., in which case you can use AERROR() to determine which rule failed and handle it appropriately.

To see an example of this, DO the CUSTOMER2 form. You'll find that you can enter "NJ" into the Region field and won't get an error message until you click the Save button. The error message that's displayed is defined by the form, not VFP. You'll also find that if you add and cancel several times, the next available CUST_ID isn't incremented because NEXT_ID.PRG isn't called until a new CUSTOMER record is inserted when you click the Save button.

Here's the code from the Click() method of the Save button; this code is simpler than you'd use in a "real" form but shows how field validation rule failure can be handled.

```
local lcAlias, ;
laError[1], ;
lcField, ;
lcDBCField, ;
lcMessage, ;
lnI
lcAlias = alias()
if tableupdate()
if cursorgetprop('SourceType') <> 3
= requery()
endif cursorgetprop('SourceType') <> 3
Thisform.Refresh()
else
select (lcAlias)
= aerror(laError)
lnError = laError[1]
do case

* If the error was caused by a field validation
* rule being violated, get the error message for
* the field from the DBC (or use a generic message
* if there isn't one) and display it. Find the
* control for the field and set focus to it.

case lnError = 1582
  lcField    = alias() + '.' + laError[3]
  lcDBCField = iif(cursorgetprop('SourceType') = 3, ;
    lcField, dbgetprop(lcField, 'Field', 'UpdateName'))
  lcDBCField = substr(lcDBCField, ;
  at('!', lcDBCField) + 1)
  lcMessage  = dbgetprop(lcDBCField, ;
  'Field', 'RuleText')
  lcMessage  = iif(empty(lcMessage), ;
    'Improper value entered into ' + laError[3] + '.', ;
    evaluate(lcMessage))
  = messagebox(lcMessage, 0, Thisform.Caption)
  for lnI = 1 to Thisform.ControlCount
    if type('Thisform.Controls[lnI].ControlSource') ;
    <> 'U' and ;
    upper(Thisform.Controls[lnI].ControlSource) = lcField
      Thisform.Controls[lnI].SetFocus()
      exit
    endif type('Thisform.Controls[lnI].ControlSource') ...
  next lnI


* Display VFP's error message.

otherwise
= messagebox('Error #' + ltrim(str(laError[1])) + ;
chr(13) + laError[2], 0, Thisform.Caption)
endcase
endif tableupdate()
```

One thing about updatable views frequently catches developers: If the table the view is defined from is buffered, the TABLEUPDATE() for the view will write the view record into the table's buffer, not directly to the disk. Also, if you forget to issue TABLEUPDATE() for the table, you'll get an error message when VFP tries to close the table because there are uncommitted changes in the table's buffer.

Lookup tables

All but the simplest applications use lookup tables. Because most lookup tables have the same structure (code and description), some developers like to use a "multi-lookup" table. A multi-lookup table combines many small lookup tables into one larger table. It usually has the same structure as an individual lookup table, but with the addition of a TYPE field. This field contains a value indicating what type of lookup each record is for. For example, *A* might represent customer type lookups, *B* customer sales regions, and *C* employee types. The primary key for this table is TYPE + CODE, so the CODE must be unique within each type. Here are some sample records:

Type	Code	Description
A	PRO	Prospect
A	REG	Regular
A	SPE	Special
B	NE	Northeast
B	SW	Southwest
C	FTP	Full-time permanent
C	FTT	Full-time temporary
C	PTP	Part-time permanent
C	PTT	Part-time temporary

The advantage of having a multi-lookup table is that there are fewer tables to open and maintain than when many smaller lookup tables are used. The disadvantage is that setting up the relationships is a little more complex because the type must be part of the relationship and a filter has to be set on the lookup table so the user can see only the codes matching the desired type. Here's an example that opens CUSTOMER.DBF and two copies of LOOKUPS.DBF and sets up relationships for the customer type and customer sales region lookups:

```
select 0
use LOOKUPS again alias CUST_TYPE order TYPECODE
set filter to TYPE = 'A'
select 0
use LOOKUPS again alias CUST_REGION order TYPECODE
set filter to TYPE = 'B'
select 0
use CUSTOMER
set relation to 'A' + TYPE into CUST_TYPE, ;
'B' + REGION into CUST_REGION
```

Working with multi-lookup tables is much easier in VFP because you can use views. You define one view for each type of lookup needed and include records only of the desired type in the view:

```
create sql view CUST_TYPE_LOOKUP as ;
select CODE, DESCRIP ;
from LOOKUPS ;
where TYPE = 'A'
create sql view CUST_REGION_LOOKUP as ;
select CODE, DESCRIP ;
from LOOKUPS ;
where TYPE = 'B'
```

Using views gives you the best of both worlds: one physical table to manage but many easy-to-use logical representations. You don't need to worry about the lookup type when setting up relations, doing SEEKs, or displaying a pick list. This code sets up the same type of relations as the previous code did, but using views:

```
select 0
use CUST_TYPE_LOOKUP
index on CODE tag CODE
select 0
use CUST_REGION_LOOKUP
index on CODE tag CODE
select 0
use CUSTOMER
set relation to TYPE into CUST_TYPE_LOOKUP, ;
REGION into CUST_REGION_LOOKUP
```

This code shows the one downside of using views: because views don't have predefined indexes, you can't set up the relationships in the DataEnvironment of a form or report because you have to re-create the indexes every time the view is opened. Other than that, this code could open separate lookup tables or open different views of the same lookup table.

Visual FoxPro 5.0

Although there are a lot of new features in 5.0, not many changes were made to the database container. However, the ones that were made will make a big difference in your productivity. Let's look at these.

One change that will simplify things is having a multiuser database container. This means you no longer have to ensure that no one else on your network has the database open before adding new tables, making table structure changes, or rebuilding indexes. Because another user can add or remove tables or views from the database container while you have it open, a new Refresh function in the Database menu rereads the database from disk. Other new functions in the

Database menu, such as Find Object and Arrange, make it easier to work with databases containing many objects.

The Table Designer has a slightly new appearance. Instead of a Table button to display a table properties (such as rule and message) dialog box, there's a Table page. This page also includes the name of the .DBF, and database and statistical information about the table (number of records, number of fields, and record size). The biggest change, though, is on the Fields page. Four new properties appear: InputMask, Format, DisplayLibrary, and DisplayClassLibrary. (These are the names of the properties as the DBGETPROP() and DBSETPROP() functions expect them.) These new properties, together with a new VFP feature called Intellidrop, will greatly improve your productivity in creating forms. Here's how it works.

In VFP 3.0, when you drag a field from the DataEnvironment, Project Manager, or Database Designer to a form, you get a control with the following attributes:

- The control is a VFP base class: Checkbox for Logical fields, Editbox for Memo fields, OLEBoundControl for General fields, and Textbox for all other data types.
- Text box controls are sized to hold about 10 characters, regardless of the actual field size, forcing you to resize the control.
- The name of the control is the name of the class followed by an instance number (for example, the first Textbox control is Textbox1 and the second is Textbox2). Usually, you'd rename the control to something more meaningful like txtCompany.
- No label is automatically created as a caption for the field.

Developers complained long and hard about the shortcomings of dropping a field on a form, and Microsoft responded with the Intellidrop feature in VFP 5.0. When you drop a field on a form, Intellidrop does the following:

- Creates a control of the class defined in the DisplayClassLibrary property for the field defined in the database (the DisplayLibrary property tells VFP where this class is stored).
- Sizes the control appropriately for the field size.
- Copies the InputMask, Format, and Comment properties of the field to the same properties of the control.
- Creates a Label object to the left of the control whose Caption property is set to the Caption of the field.

You can turn off some or all of these features by bringing up the Tools Options dialog box, selecting the Field Mapping page, and clearing the appropriate check box. You can also define which class to use by default for each data type; this class is used whenever a field with "<default>" as the DisplayClassLibrary is dropped on a form.

Like many things, Intellidrop has shortcomings:

- The name of the control is still the class name followed by an instance number.
- The controls aren't always sized perfectly; you might still need to tweak the width a bit.
- The label it creates is of the Label base class; there's no way to define a different class to use.
- Only classes stored in visual class libraries (VCXs) can be used (this isn't so much a complaint as an observation).
- If you change the field properties in the database, you have to delete the label and field control and re-drop the field on the form, because these properties aren't dynamically tied to the control.

In addition to the benefits provided by Intellidrop, storing the InputMask and Format for a field in the database means you can create controls that dynamically set their InputMask and Format properties as appropriate for their ControlSource at runtime. The advantage is reduced maintenance: If you decide a field should contain only uppercase data, you have to edit every form and report, displaying the field if the format is hard-coded into its control. If the control asks the database at runtime for the format for the field, no editing is required at all. This is especially useful for user-customizable applications, such as an accounting system where the user can define the format for account codes or inventory part numbers.

Here's code you can put in the Init() method of a control (or better yet, in the Init() method of a class) to set the InputMask and Format properties at runtime (see the SFTextbox class in the CONTROLS.VCX class library in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP):

```
if not empty(This.ControlSource) and ;
not empty(dbc()) and ;
indbc(This.ControlSource, 'Field')
local lcInputMask, lcFormat
lcInputMask = dbgetprop(This.ControlSource, ;
'Field', 'InputMask')
lcFormat = dbgetprop(This.ControlSource, ;
'Field', 'Format')
This.InputMask = iif(empty(lcInputMask), ;
This.InputMask, lcInputMask)
This.Format = iif(empty(lcFormat), ;
This.Format, lcFormat)
endif not empty(This.ControlSource) ...
```

An example of this scheme is shown in the EMPLOYEE.SCX form in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP. If you examine the controls on this form, you'll find the InputMask or Format properties haven't been set. However, when you DO the form, the Phone field is formatted as 999-999-9999 and the Category field is forced to uppercase because that's how those fields are defined in the database.

I tested this code's performance by including or excluding the code from the SFTextbox class used for the controls on the form, and found the form takes about 15 percent longer to instantiate. However, that means the form takes 0.327 seconds to display rather than 0.283 seconds, so the user really won't notice the difference. Thus, perhaps with the exception of the busiest of forms, I suggest using this scheme to ensure your forms keep up to date with changes made to the database.

Another new feature you're going to love is being able to update fields in the current table in field and table validation rules. VFP 3.0 allowed you to update fields in another table in validation code, but attempting to change a field in the current table caused an "illegal recursion" error. For example, this meant you couldn't define code to timestamp a record (put the date and time of the last change into a field) as a table rule, but instead had to do it in a form. The advantage of putting this code in a rule is that it automatically happens without the form developer having to code for it, and it also works in browses or programmatic changes to the table. In VFP 5.0, you can use code similar to the following (this code is in the stored procedures of the DATA database in download file HENNIG_A.ZIP and download file HENNIG_B.ZIP and is called as the table validation rule for the EMPLOYEE table):

```
function EmployeeTableRule
local ltStamp

* Assign the EMP_ID field if it hasn't been already.

if empty(EMP_ID)
replace EMP_ID with NEXT_ID('EMPLOYEE')
endif empty(EMP_ID)

* Timestamp the record.

ltStamp = datetime()
if LAST_UPDATE <> ltStamp
replace LAST_UPDATE with ltStamp
endif LAST_UPDATE <> ltStamp
return .T.
```

This code does two things: assigns the primary key for the current record if it hasn't already been assigned (this is another way to prevent "holes" in sequential primary keys as I discussed earlier), and timestamps the record every time it's changed (you could also store the name of the user who made the change). You can use field level validation rules to ensure that the contents of a field are forced to uppercase or to automatically fill in partial entries with a complete value (similar to the AutoCorrect feature in Word 95).

There are a few important things to know when writing rules that update the current table:

- Validation rules can modify the current table, but triggers can't; they still give an "illegal recursion" error.
- The code that updates a field should do so only if the field's value is different from what

should be stored. The previous code updates the two fields only if their values need to be updated. This prevents recursion (yes, recursion can still occur in validation rules if you don't do it properly).

- If it's possible that recursion may occur, you can prevent it from going past more than one level (and causing VFP to bomb) by checking the calling stack (using the PROGRAM() function) and not doing the update if the validation code called itself:

```
  lcProgram = ''
lnLevel   = 1
do while not empty(program(lnLevel))
lcCaller  = lcProgram
lcProgram = program(lnLevel)
lnLevel   = lnLevel + 1
enddo while not empty(program(lnLevel))
if not lcCaller == program()
* do the REPLACE here since we haven't
* called ourselves
endif not lcCaller == program()
```

Thanks to Linda Teh, Andy Neil, and Jim Slater for exploring and clarifying these issues in the VFP 5.0 Beta Forum on CompuServe.

Conclusion

Next month's column will be "Christmas Stocking Stuffers," a potpourri of useful ideas. We'll look at lots of little things, such as where to put utility functions, why you shouldn't use the GO command, and how to visually indicate that a control is read-only.

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 Data Dictionary for FoxPro 2.x and Stonefield Database Toolkit for Visual FoxPro. 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 throughout North America. 75156.2326@compuserve.com.

A User-Configurable Application Toolbar

Stephen A. Sawyer

WIN 3.x

Allowing your users to configure their own toolbars can greatly improve their productivity, as well as their attitude toward your application. This article shows how easy this can be, and how to implement user-configurable toolbars within the framework of a simple, intuitive, form-opening

interface.

Toolbars permit users to quickly invoke certain actions in an application, with the simple click of a mouse. In a complex application, a maze of submenus can be very frustrating to users who have to interact with them eight hours a day, five days a week. While a certain amount of navigation "overhead" is unavoidable, I want to reward experienced users of my applications by allowing *them* to choose their most commonly invoked functions, and allowing *them* to place these functions on an application-level toolbar for instant access. Each user's needs are unique, and each user will have different candidates for "most used" application functions.

Another way I hope to streamline access to my applications is to use a document- or form-centric, form-opening interface. These two approaches work together to greatly improve the user interface and keep users focused on their task, rather than on manipulating my application.

I've discussed with many other developers the lack of "document-centricity" in database applications. Some claim this makes it more difficult to preserve the "standard" Microsoft Office menu structure, with File as the left-most menu pad, Help as the right-most pad, and Edit, View, and Format arranged between them.

The problem stems not so much from the nature of database applications, but from the mindset we inherit from the file-centric nature of DOS (and Windows), as opposed to the document-centric nature of Macintosh operating systems. As a developer, I see a database application as *tables* implemented as .DBF *files*, so I tend to think of the File menu pad as the starting point for opening a *file* or *table*. But database applications seldom deal with a single file or table at a time. The data in the real-world business objects that I'm implementing usually is distributed among several related data tables. The solution to this problem, I believe, is to take a step backward to the world as it existed before we started automating all of these business processes -- to the days when users kept racks of blank forms, and file cabinets or tubs of completed forms arrayed around their physical desktop.

In our database applications, forms are still "forms," but they're presented as pixels on a monitor instead of preprinted forms on paper. I don't see why we can't take advantage of this fortuitous coincidence in our interface design. As a result, I'm implementing my menu to open a single dialog box, similar to the familiar File|Open dialog box, that gives the user access to *all* of the forms in the system allowed by his or her security level.

In a system such as ours, the downside of this approach is that scrolling through a long list of forms can be as annoying as navigating a complex menu tree, without the helpful structure of pop-up menus and submenus. In the existing FoxPro/DOS application that I'm rewriting in Visual FoxPro, I hope to trim the 168 forms to a more manageable number. However, the system will probably still contain a large number of forms -- enough to make finding a particular form time-consuming, and cause users to focus more on managing the application than getting their work done. Allowing users to select forms from this list to personalize their application-level toolbar will give them the best of both worlds: a single, consistent point from which they can launch *any* form in the system, and a quick-access toolbar for the forms they use on a daily basis.

In this article, I'll discuss the use of a toolbar that can be user-configured to suit individual preferences, and then cover the management of a large number of forms through a single dialog box.

Implementation of a dynamic, configurable toolbar

Because all of the object classes I've used to implement this solution are part of a complete application framework, I haven't provided any source code for these examples. Having studied other developers' application frameworks, I know how difficult it can be to trace the logic among object classes, subclasses, and instances, so I'm not going to frustrate you with a gaggle of class libraries. Instead, I'll discuss the general approach I used, and quote enough code to communicate the ideas so you can adapt them to your own framework.

The lynchpin of the technique outlined here is a table that contains all the necessary information on the system's forms (see **Table 1**). In my framework, it's named FORMLIST.DBF.

Table 1. ***The structure of FORMLIST.DBF.***

Field name	Field type	Field length
CDESCRIPT	C	30
CPICTURE	C	20
CTYPE	C	7
LONTOOLBAR	L	1
CSCX	C	20
IORDER	I	4
ISECURITY	I	4
CUSER	C	10
IRECNO	I	4

- **cDescript** -- The description of the form that the user sees, both in the Form|Open dialog box and as the ToolTip for the buttons on the toolbar.
- **cPicture** -- The name of the .BMP file for the toolbar buttons.
- **cType** -- An identification of the form *type*. In our system, forms are either *data entry* forms (identified as type "Form" in the cType field), *list* forms (which display a list of items such as all open orders that can be used to launch a form to edit or display that item), or *inquiry* forms, which display non-editable information, and may or may not be associated with a report for printed output.
- **lOnToolBar** -- A field to indicate whether this form should appear on the user's toolbar.
- **cScx** -- The form name.
- **iOrder** -- An integer value that determines the sort order of the forms in the list.
- **iSecurity** -- An integer field whose value indicates the security level necessary for a user

to access the form.

- **cUser** -- Identifies a subset of the FORMLIST records as belonging to a particular user. This field is empty in the records that represent the "master" list from which all user lists are constructed.
- **iRecNo** -- Contains each record's record number. iRecNo is used in user-maintenance of this list which will be examined later.

Creating the user's form list

Once this "master" list is established, each user must have either a copy or a subset of the list. In my application framework, several operations are performed in the application object's Init() method, including instantiation of a menu manager object, a form manager object, an environment object, and so forth. The following Init() code shows the calls that impact the user-configurable form list. I've chosen the unusual name of oApp for the instance of my application object:

```
* Portion of oApp.Init()

*!*   Identify the user and establish
*!*   the user's security level
*!*   Sets This.cUser and This.iSecurityLevel
This.SetSecurityLevel()

*!*   Prepare the formlist table
This.CreateFormList()

*!*   Put up the main toolbar
THIS.ShowToolbar()
```

By leaving the SetSecurityLevel() method in my cApplication foundation class blank, an application-specific procedure instead establishes the user's security level. The user's security is stored as an integer value to an application property, iSecurityLevel. The SetSecurityLevel() method should also store the username to another application-object property, cUser. In our system, this method gets the username from a system environment variable and looks up the security level in a security table. However, a more secure system might invoke a login procedure to perform the same functions.

Once these properties have been established by SetSecurityLevel(), the CreateFormList() method can construct (if necessary) the user's own form list, which is appended to the FORMLIST table:

```
*oApp.CreateFormList Method

LOCAL liSecurityLevel, ;
   lcTemp

IF ! EMPTY(This.cUser) AND OPENDBF("FormList")
   *!* Check to see if the user already has a
   *!* set of records in the FORMLIST table
   LOCATE FOR cUser = This.cUser
   IF ! FOUND()
      *!* If the user has no records in the FORMLIST
      *!* table, extract a subset of records,
      *!* determined by the user's security level,
      *!* and append them with the username placed
      *!* into the cUser field to the FORMLIST table
      lcTemp = GENTEMP()
      liSecurityLevel = This.iSecurityLevel
      SELECT cDescript, ;
            cPicture, ;
            cType, ;
            lOnToolBar, ;
            cScx, ;
            iSecurity, ;
            iOrder, ;
            cUser, ;
            iRecNo ;
         FROM formlist ;
         WHERE iSecurity <= liSecurityLevel ;
            AND EMPTY(cUser) ;
         ORDER BY iOrder ;
         INTO TABLE (lcTemp)
      REPLACE ALL cUser WITH This.cUser
      USE
      SELECT formlist
      APPEND FROM (lcTemp)
      REPLACE iRecNo WITH ;
         RECNO("formlist") ;
         FOR  ALLTRIM(cUser) = ALLTRIM(This.cUser)
      ERASE (lcTemp)
   ENDIF
ENDIF
USE IN FormList
```

Only those records for which the user has a sufficiently high security level are duplicated -- witness the WHERE clause in the method code.

This method uses two functions, OPENDBF() and GENTEMP(), that reside in my procedures library. OPENDBF() simply determines if the table is already open, and if not, opens it and returns a logical value indicating whether the operation was successful. GENTEMP() returns a temporary filename that by default is created in the same directory in which all window "temp" files are created.

If the master list changes, the user's records can be "cleared" simply by deleting the records, and the user list will be rebuilt. This method can be made more sophisticated, if you wish, to look for and append "missing" records, thereby preserving the user's preferences.

Creating the application toolbar at runtime

Once the username, security level, and user-specific form list are established, you have to provide some means for the user to indicate which of the forms in the FORMLIST table should be accessible from the application-level toolbar. Initially, the FORMLIST table indicates that *none* of the forms is accessible from the toolbar, so no toolbar is created. Once the user *has* indicated his or her preferences, the next step is to instantiate the application toolbar. This is accomplished by the application object's ShowToolBar() method.

First, you need to determine if there's already a toolbar in existence. If there is, the user must be making a change to it, so store a .NULL. value to the application object property that contains the reference to the current toolbar. Once that's done, you're free to instantiate your new toolbar, using the class name from the cMainToolBarClass property you designated at design time.

```
*oApp.ShowToolBar() method

LOCAL loFormManager

IF !EMPTY(this.cMainToolbarClass)
   IF TYPE("This.oFormManager") == "O" ;
         AND ! ISNULL(This.oFormManager)
      loFormManager = This.oFormManager
   ELSE
      loFormManager = .NULL.
   ENDIF
   *!*   If the user already has an application
   *!*   toolbar instantiated, clear it so that
   *!*   a new version can be instantiated.
   IF TYPE("This.oToolBar") = "O" ;
         AND ! ISNULL(This.oToolBar)
      This.oToolBar = .NULL.
   ENDIF
   This.oToolBar = ;
      CREATEOBJECT(This.cMainToolBarClass,loFormManager)
ENDIF
```

While any toolbar class can be specified, the application toolbar class that creates itself dynamically, using the FORMLIST table, is called AppToolBar. AppToolBar is defined with a private datasession, no control box, and a property inherited from my foundation toolbar class, lSavePosition. This property is used by the foundation class's Init() and Destroy() method code, which saves the toolbar's position to the Windows registry and restores its position in subsequent sessions.

All of the important work of AppToolBar is done in its Init() method, which dynamically creates

the necessary toolbar buttons based on current user information in the FORMLIST table:

```
*AppToolBar class Init()

LPARAMETERS toFormManager
DoDefault(toFormManager)
USE formlist ORDER TAG iOrder
lcUser = This.oFormManager.Parent.cUser
SET FILTER TO FormList.lOnToolBar AND FormList.cUser = lcUser
LOCATE
IF ! EOF()
   lnCounter = 0
   SCAN
      lnCounter = lnCounter + 1
      lcButtonName = "cmd" + LTRIM(STR(lnCounter))
      This.AddObject(lcButtonName,"AppToolBarButton")
      lcButtonName = "This." + lcButtonName
      &lcButtonName..Picture = formlist.cPicture
      &lcButtonName..cForm = ALLTRIM(formlist.cscx)
      &lcButtonName..ToolTipText = ;
           ALLTRIM(formlist.cdescript)
   ENDSCAN
   USE
   This.SetAll("Visible",.T.)
   This.Visible = .T.
ELSE
   RETURN .F.
ENDIF
USE IN formlist
```

The method makes use of the new DoDefault() function of Visual FoxPro 5.0 rather than the scope resolution operator to invoke the parent class's Init() method code. The Init() of the foundation class stores the reference to the form manager object (passed by ShowToolBar) to an object property (oFormManager) and restores the toolbar's position. The Init() code uses the toolbar's AddObject() method to add a command button for each item in the FORMLIST table for which the lOnToolBar field is set to .T. The code then sets the properties of this command button according to information contained in FORMLIST.DBF, including the ToolTipText (from formlist.cDescript), Picture (from formlist.cPicture), and cForm (from formlist.cSCX) properties.

The command buttons added by the AppToolBar.Init are based on a class named AppToolBarButton. This class includes a character property, cForm, and code in the Click() method to launch the form that was stored to the cForm property by the AppToolBar.Init(). The Click() code of the AppToolBarButton class launches the form whose name is stored in its cForm property by calling the DoForm() method of the form manager object:

```
*AppToolBarButton class Click() method

IF TYPE("This.Parent.oFormManager") = "O" ;
      AND ! ISNULL(This.Parent.oFormManager)
   This.Parent.oFormManager.DoForm(This.cForm)
ENDIF
```

All of the foregoing code allows the application toolbar to be maintained for each individual user simply by managing a single table, FORMLIST.DBF.

Managing FORMLIST.DBF using a Form|Open dialog

As mentioned earlier, the FORMLIST.DBF table serves several functions in my framework, not only providing the information for toolbar creation, but also allowing user access to *all* forms in the system, *and* allowing the user to determine which forms can be launched from the toolbar. These last two functions are actually performed by a form, frmFormList (see Figure 1). This dialog box is invoked by selecting File|Open or Form|Open from the system menu. (I haven't yet decided if I'm going to go for internal consistency with "Form" or consistency with 99 percent of the Windows apps in the world with "File.")

Once a form is selected in this form-opening dialog box, the user can either double-click the selected form or click the Open command button to launch it. The user (as prompted by the ToolTip for the ListBox) can also right-click on an item to toggle its status as a form that is or is not included on the application toolbar. If the user simply wants to change the "on toolbar" status of the form, he or she can select the Ok command button, save the changes, and close the Form|Open dialog box. Selecting the Open button (or double-clicking on a form in the list) also automatically saves any changes the user has made to the "on toolbar" status of any form. If the user wants to close the dialog box without opening another form or saving any changes to the on-toolbar status of any form, he or she can click the Close button or select "close" from the control menu.

The form-opening dialog box's WindowType is "1Modal," DataSession is "2Private," and it has three added properties:

- **cUser** -- Initialized to a null string, used to hold the current username.
- **lToolBarChanged** -- A logical value to avoid rebuilding the toolbar unless the user has made some changes to its configuration.
- **aFormList** -- An array property that is populated with the user's list of available forms and used by a ListBox object (lstFormList) to present the list to the user.

This form's Init() code follows. Using an SQL-SELECT command, it extracts the user's list records from the FORMLIST table to the aFormList form array property, substituting an "X" character for the logical value stored in the lOnToolBar field, including the form name from the cScx field, and the record's record number from the iRecNo field. Because this is in the form's Init() and the ListBox is already instantiated, it finishes by calling the ListBox's Requery()

method:

```
frmFormList.Init()

LPARAMETERS toParam, tuParam
DoDefault(toParam, tuParam)
IF TYPE("ThisForm.oFormManager.Parent") = "O" ;
      AND ! ISNULL(ThisForm.oFormManager.Parent)
   ThisForm.cUser = ThisForm.oFormManager.Parent.cUser
ELSE
*!* The following line is useful during testing
   ThisForm.cUser = GETENV("USER")
ENDIF
lcUser = ThisForm.cUser
SELECT cDescript, ;
      cType, ;
      IIF(lOnToolBar,"      X","       "), ;
      cScx, ;
      iRecNo ;
   FROM Formlist ;
   WHERE cUser = lcUser AND ;
      ! DELETED("FormList") ;
   INTO ARRAY ThisForm.aformList
ThisForm.lstFormList.Requery()
```

The ColumnCount of the ListBox (lstFormList) is set to 3, so only the columns containing information from the FORMLIST table's cDescript, cType, and lOnToolBar fields are visible. The ListBox's ToolTipText property is set to "Right-click to toggle toolbar inclusion". The ListBox's DblClick() method calls the Click() method of the Open command button, cmdOpen.

Here's the code from the ListBox's RightClick() method that toggles the on-toolbar status of the selected form:

```
* frmFormList.lstformlist.RightClick()

IF This.List[This.ListIndex,3] = "      X"
   ThisForm.aFormList[This.ListIndex,3] = ""
ELSE
   ThisForm.aFormList[This.ListIndex,3] = "      X"
ENDIF
ThisForm.lToolBarChanged = .T.
ThisForm.cmdSave.Enabled = .T.
This.Requery()
```

A double-click on the ListBox calls the Click() method of the cmdOpen command button, shown in the following code. The code makes use of a GetColumnValue() method in my ListBox baseclass to determine the name of the form, but you could just as easily retrieve the form name from the array as shown in the optional code. Because my framework passes a reference to the form manager to each form opened, the cmdOpen.Click() method can call the form manager's

DoForm() method to launch the selected form. Rather than duplicating code to close the form and save any changes the user has made to the forms in the on-toolbar list, the cmdOpen.Click() method (after launching the requested form) calls the Ok button's Click() method which performs these functions:

```
* frmFormList.cmdOpen.Click()

LOCAL lcForm

lcForm = ;
  ALLTTRIM(ThisForm.lstFormList.GetColumnValue(4))
*!* Alternate code
*!* lcForm = ;
  ThisForm.aFormList[ThisForm.lstFormList.ListIndex,4]

IF TYPE("ThisForm.oFormManager") = "O" ;
   AND ! ISNULL(ThisForm.oFormManager)
   ThisForm.oFormManager.DoForm(lcForm)
ELSE
   DO FORM (lcForm)
ENDIF
ThisForm.cmdOk.Click()
```

Finally, in the cmdOk.Click() method, you can see the purpose for the iRecNo field data in the form array. It's used to transfer the user's preferences from the form array used by the ListBox back to the FORMLIST table:

```
* frmFormList..cmdOk.Click()
IF ThisForm.lToolBarChanged
   SELECT FormList
   FOR i = 1 to ALEN(ThisForm.aFormList,1)
      GOTO ThisForm.aFormList[i,5]
      IF "X" $ ThisForm.aFormList[i,3]
         REPLACE FormList.lOnToolBar WITH .T.
      ELSE
         REPLACE FormList.lOnToolBar WITH .F.
      ENDIF
   ENDFOR
   IF TYPE("ThisForm.oFormManager.Parent") = "O" ;
         AND ! ISNULL(ThisForm.oFormManager.Parent)
      ThisForm.oFormManager.Parent.ShowToolBar()
   ENDIF
ENDIF
ThisForm.Release()
```

It's important to point out that you don't have to buy into my interface philosophy to get some mileage out of these techniques.

While *I* find a great deal of merit in having a single dialog box for opening all forms in the

system, you can implement this type of interface or, if you prefer, the user-configurable toolbar can be implemented independently. You can invoke a dialog box designed specifically for configuring the toolbar under a Tools|Options menu item, which would be very compatible with the Microsoft Office "standard." However, you can just as easily use this general method of creating a toolbar at runtime to create static (non-user-configurable) application-level toolbars (which could be more easily modified when the application is in production and you've entered the maintenance phase of the software life-cycle).

Likewise, you can implement the single Form|Open dialog using the FORMLIST table without implementing the application-level toolbar, if it's inappropriate for a particular application.

Whatever philosophy you follow for your user-interface design, I think that with varying degrees of modification, there's something here for almost everyone.

Steve Sawyer is a corporate developer for a manufacturing company in Detroit, Michigan. He's the author of The Visual FoxPro Form Designer in Pinnacle Publishing's The Pros Talk Visual FoxPro series. 75730.455@compuserve.com.

New in Visual FoxPro 5.0:

Improved Pixel Support

John V. Petersen

(3)

One of the biggest limitations of version 3.0 in using functions such as MCOL() and MROW() was that its scale mode was the foxel. In Windows development, the common unit of measure for screen resolution and positioning is the pixel. The foxel was intended to aid in the development of cross-platform applications. Not too long ago, cross platform meant DOS/MAC/Windows/UNIX. Today, cross platform really means Windows 95/Windows NT. While there is a Macintosh version of VFP 3.0, the jury is still out on whether there will be a 5.0 version for the Macintosh. In any case, any Windows application development environment has to fully utilize pixelsæespecially when using functions such as MCOL() and MROW().

With version 5.0, MCOL() and MROW() can now return the position of the mouse in pixels. If you examine the generated code from a shortcut menu .MPR file, you'll see that calls to MCOL() and MROW() now properly place the right click menu at the location of the mouse pointer. The only way to determine the pixel position of the mouse pointer prior to version 5.0 was through one of the mouse events (MouseUp, MouseDown, or MouseMove) in which the X and Y coordinates were passed as parameters to the method code snippet, or through a conversion routine in which you temporarily swapped the font properties of the form with the font properties of _SCREEN. Unfortunately, the mouse events didn't apply to every situation where exact mouse coordinates were needed, and a conversion routine was rather cumbersome.

This new feature is most important when you use builders. Before version 5.0 it was difficult, if not impossible, to place a control in the exact position of the mouse pointer. Now, the task is

simple. The following sample builder program, in download file _PETERS.EXE, illustrates the new capabilities of MCOL() and MROW():

```
*/ Placetextbox.prg
*/ Sample builder program to illustrate new
*/ capabilities of the MCOL() and MROW() functions.

*/ John V. Petersen

*/ To Call: Create an OKL such as ALT+ F2

*/ On Key Label Alt+F2 Do Placetextbox

*/ To use the builder, position mouse pointer where
*/ you want the new text box to be placed and then
*/ initiate your OKL.LOCAL ARRAY laForm[1],laFont[3]
LOCAL lnRow,lnCol,lnCount
LOCAL lcFontInfo,lcName
STORE 0 TO lnRow,lnCol,lnCount
lcFontInfo = SPACE(0)

*/ Define Constants for second argument in MCOL()
*/ and MROW() Calls.
#DEFINE pixel_scale3
#DEFINE foxel_scale 0

IF ASELOBJ(laForm,1) = 1
   lcFormName = laForm[1].NAME
   lnRow = MROW(laForm[1].NAME,pixel_scale)
   lnCol = MCOL(laForm[1].NAME,pixel_scale)
   lcFontInfo = GETFONT()
   IF !EMPTY(lcFontInfo)
      ParseFontInfo(lcFontInfo,@laFont)
*/ Initiate a loop to find a valid name for object
      DO WHILE .T.
         lnCount = lnCount + 1
         lcName = "Text"+ALLTRIM(STR(lnCount))
         IF TYPE("laForm[1]."+lcName) = "U"
            EXIT
         ENDIF
      ENDDO
*/ Add the object
      laForm[1].ADDOBJECT(lcName,"textbox")
*/ Assign the properties
      WITH laForm[1].&lcName
         .LEFT = lnCol
         .TOP = lnRow
         .FONTBOLD = "B"$laFont[3]
         .FONTITALIC = "I"$laFont[3]
         .FONTSIZE = VAL(laFont[2])
         .FONTNAME = laFont[1]
      ENDWITH
   ENDIF
ENDIF

PROCEDURE ParseFontInfo

*/ This procedure parses the chosen font
*/ characteristics from GETFONT into an array
```

DOWNLOAD

John V. Petersen, MBA, specializes in the development of Visual FoxPro-based solutions for business. He is also a coauthor of Developing Visual FoxPro 5.0 Enterprise Applications by Prima Publishing. 610-651-0879, 103360.1031@compuserve.com.

Entrapment

Les Pinter

Why aren't the police allowed to leave $100 bills on the sidewalk, then arrest people who pick them up? It's called *entrapment*. It brings out the worst in human nature. Yet some of us do it all the time.

You're a programmer. You love to code and you'd like to be an independent consultant. One day you hear that someone wants some code written. So you take a deep breath, strike a deal, and give notice. Two weeks later, you're an independent consultant. You arrive, get the specs, and rub your hands gleefully. You're going to be a hero!

The end of the first week comes, and you type out an invoice and leave it on the client's desk. On Monday, you're back at work. There's no check on your desk, but who gets immediate payment? You start the second week.

You've never done such good work. The freedom, the exhilaration of the new challenge, brings out the best in you. By the end of the second week, you've amazed yourself. You leave the source code and a second invoice on the client's desk and go fishing for the weekend.

The third week, you're so excited about the progress you're making that you don't even remember to bill. So at the end of the fourth week, you print up two invoices and leave them on your client's desk. Maybe he's forgotten. You decide to speak up. "Here are my invoices for the last two weeks," you offer helpfully. He doesn't even look up. "I'll get to them. See you Monday."

By the middle of the fifth week you're getting nervous. The bills are coming due, and your assurances to your wife that "there's money in the pipeline" are beginning to sound a little hollow. You decide to have a little chat with your client.

"You call yourself a programmer?" he shouts. "I expected you to be done by now. Why are you so slow?" You're astonished. The original estimate you gave him was for four months, and you're already more than half done. He's added dozens of new specs. And you can program circles around other programmers. What's going on?

By the end of the week, you've had one ugly confrontation after another, each accusation more contrived than the previous one. Finally, on Friday, your client informs you that your work is not worth paying for. You realize that you don't even have any proof that you ever worked there. *And he's got your source code.*

What you did wrong was to leave the cost of five weeks of work on the table for someone to steal. If you'd gotten paid at the end of each week, you wouldn't have created such a tempting target. But the pile of cash got so high that it was irresistible!

Fool me once, shame on you; fool me twice, shame on me.

Les Pinter is a software developer in Silicon Valley. His first novel, The Valley , will be out in the fall. 415-344-3969, .

Tables and Labels

Compiled by Barbara Peisch

Use GENDBC to Create Tables at a Remote Site

Moving a VFP database to a different directory may require changing the pointers between the tables and the .DBC. If you have live data that you need to preserve, Andy Neil's method using low-level file functions may be necessary. But if you're moving empty tables (for example, installing an application on a client's LAN), you can simply use GENDBC.PRG (in the VFP\TOOLS\GENDBC directory) to generate the definition for the database. You can then use search and replace to change the paths, and run the resulting program to create the database in the new directory. One caution, though: If you're running under VFP 3.0 and have any RI code stored in the .DBC, two commands generated by GENDBC -- APPEND PROCEDURES FROM... and COMPILE DATABASE... -- are not supported when running under the runtime library. In order to work around this for a runtime environment, substitute the following code for these two commands:

```
CLOSE DATABASE
USE <database name>
LOCATE FOR Objectname = "StoredProceduresSource"
IF FOUND()
   APPEND MEMO Code FROM <RI procedure file> OVERWRITE
ENDIF
LOCATE FOR Objectname = "StoredProceduresObject"
IF FOUND()
   APPEND MEMO Code FROM <RI Procedure file> OVERWRITE
ENDIF
USE
OPEN DATABASE <DBC name>
```

If you are using version 5.0, this is the code generated, so you don't have to change it.

Bill Budney

Make Sure Your Labels Look the Same on All Platforms

In Windows 95, along with the big interface changes we've come to know and love, Microsoft also made some subtle changes. One change is whether or not labels are bold. In Windows 3.1x and Windows NT 3.x, the labels on forms were bold. In Windows 95 and NT 4.0, the labels are not bold. If you've had a chance to use VFP 5.0, you've no doubt discovered that the default for labels in VFP is "no bold," just the opposite of version 3.0, and maybe different from the default of your operating system. If you prefer that your applications use bold (or no bold) labels consistently, regardless of platform, it's easy to add code to your label subclass that checks the operating system version and then sets the FontBold flag. But that's not enough. If you have labels that are right-aligned they'll move out of alignment. Here's how to keep your labels aligned, whether you want them aligned left or right.

I've added a property to my base label class called lLabelRightAlignment. It accepts a logical value. Set its default to the value you most commonly use. For me, that's .F., as most of my labels are left-aligned.

In the label's Init() method, add the following code:

```
IF NOT " 3." $ OS()
   * Windows 95 & NT 4 return a "4" from OS()
   IF This.lLabelRightAlignment
      LOCAL nRight, lAutoSize
      * Calculate the current actual Right
      * Coordinate.
      nRight = This.Width + This.Left
      * Store the Current AutoSize value
      lAutoSize = This.AutoSize
      * This code requires AutoSize Set to .T. to
      * work correctly.
      This.AutoSize = .T.
      * Change the FontBold to match the current
      * Windows Standard.
      This.FontBold = .F.
      This.Left = nRight - This.Width
      This.AutoSize = lAutoSize
   ELSE
      * Change the FontBold to match the current
      * Windows Standard. No Recalculation is
      * required to correct for repositioning.
      This.FontBold = .F.
   ENDIF
ELSE
   IF NOT This.FontBold
      * If Font is already bolded, don't do anything.
      IF This.lLabelRightAlignment
         LOCAL nRight, lAutoSize
         * Store the Current AutoSize value
         lAutoSize = This.AutoSize
         * This code requires AutoSize Set to .T.
         * to work correctly.
         This.AutoSize = .T.
         * Change the FontBold to match the current
         * Windows Standard.
         This.FontBold = .T.
         This.Left = nRight - This.Width
         This.AutoSize = lAutoSize
      ELSE
         This.FontBold = .T.
      ENDIF
   ENDIF
ENDIF
```