Chapter 13 Miscellaneous Things

"There are three roads to ruin; women, gambling and technicians. The most pleasant is with women, the quickest is with gambling, but the surest is with technicians." (Georges Pompidou, French president. Quoted in "Sunday Telegraph," London, 26 May 1968).

This chapter contains an assortment of useful tips and utilities that do not really belong in any one place but which may be useful in their own right from time to time. Enjoy!

Using the Visual FoxPro debugger

The Visual FoxPro debugger, in the form that was first introduced in Version 5.0, is a very powerful set of tools which can be of enormous value when testing your code. However, it is also quite a complex tool and can take some getting used to.

Features of the debugger windows

Each of the windows available in the debugger has some interesting feature that may not be obvious. Here are some of the things that we have gleaned about them:

Locals Window

- You can change the value of a local variable or property by clicking in the values column of the window and typing in a new value. Such changes will be retained as long as the variable or property remains in scope
- Right-clicking brings up a shortcut menu which allows you to toggle the visibility status, and hence the clarity of the display, of :
    - **Public** Variables declared with the **`PUBLIC`** keyword
    - **Local** Variables declared with the **`LOCAL`** keyword
    - **Standard** All variables in the scope of the procedure named by "Locals for"
    - **Objects** Object references
- You can drill down into an object or array by clicking on the "+" sign in the window
- The Locals Window can be invoked programmatically with **`ACTIVATE WINDOW LOCALS`**

Watch Window

- You can change the value of a local variable or property by clicking in the values column of the window and typing in a new value. Such changes will be retained as long as the variable or property remains in scope
- You can drill down into an object or array by clicking on the "+" sign in the window
- You can drag expressions to and from the Command Window and the 'Watch' entry line

- You can drag expressions from the trace window directly into the 'Watch' entry line
- Shortcut references like *This* and *ThisForm* can be used in Watch expressions
- To see the name of whichever object is under the mouse pointer at any time include the following in the list of Watch expressions: "**`SYS(1272, SYS(1270))`**"
- The watch window can be activated programmatically with **`ACTIVATE WINDOW WATCH`**

Trace Window

- In addition to the 'Resume' (green triangle) and 'Cancel' (Red circle) options, the debugger provides four movement options when tracing code as follows:
  - **Step Into**: Execute current line of code only
  - **Step Over**: Disables trace while executing a subroutine, or call to a method or user-defined function
  - **Step Out**: Resumes execution of the current procedure or method but suspends again when the procedure/method is completed
  - **Run to Cursor**: Resumes execution from the current line and suspends when the line containing the cursor is reached
- Hovering the mouse pointer over a variable or property reference displays a tooltip window showing the current value for that item. You can limit what is evaluated by highlighting text in the trace window. One use for this is verifying that an object is really an object by stripping off a method call. For example, hovering the mouse over "*oObj.Init()*" will not display anything, but if you highlight only "*oObj,*" you will get a tooltip showing "*Object.*" Another use is to get the value of items enclosed in quotes; selecting the "*toObj.Name*" from "**`CASE TYPE("toObj.Name") =`**" will display the value of the name property in the tooltip.
- **Set Next Statement** option: When you are in trace mode, you can jump blocks of code, or re-execute code, simply by moving the mouse cursor to the required row and choosing 'Set Next Statement' from the 'Debug' menu (Keyboard: "**`ALT D + N`**"). Whereas the 'Run to Cursor' option temporarily disables tracing but still runs the code, 'Set Next Statement' moves directly to the specified location without executing any intervening code at all.
- Code can be copied from the trace window and pasted into the command window or to a temporary program and executed independently either as a block or by highlighting lines and executing them directly.
- You can highlight and drag text directly from the trace into the watch window.
- The Trace window can be activated programmatically with **`ACTIVATE WINDOW TRACE.`**

Debugout Window

- Any valid FoxPro expression preceded with the command **`DEBUGOUT`** is evaluated and echoed to the debug output window. All of the following are valid: **`DEBUGOUT "Hello World"`**, **`DEBUGOUT DATE()`**, **`DEBUGOUT 22/7.`**

- `DEBUGOUT` output can be enabled programmatically using `SET DEBUGOUT TO <file> [ADDITIVE]` and disabled with `SET DEBUGOUT TO`.
- `ASSERT` messages, and any events that are being tracked in the event tracker, are also echoed to the DebugOut destination (i.e. to file or window or both).
- To track custom methods or native methods that are not available in the event tracker, include `DEBUGOUT PROGRAM()` statements. (Note: Such statements do not even need to be removed. If the output window is not available because it is not open or because the program is being executed under a run time environment, all DebugOut commands are simply ignored.)
- The DebugOut window can be activated programmatically with `ACTIVATE WINDOW "Debug Output."`

CallStack Window

- Enabling "Show Call Stack Order" places a sequential number alongside each program in the Call Stack window indicating the order in which they are executing. For some reason, control of this function has been named 'Ordinal Position' in the pop-up menu which appears when you right-click in the Call Stack window.
- Enabling the 'Call Stack' Indicator displays a black triangle in the border of both the Call Stack and Trace windows whenever you are viewing a program other than the one that is currently executing. In the Call Stack window this triangle indicates which program you are viewing, while in the Trace window it indicates the line, in that program, which is executing.

Configuring the debugger

The set-up for the debugger is controlled through the Options dialog, where it has its own tab. The settings are all stored (as with all other options controlled by this dialog) in the Windows registry.

Which frame to use?

The debugger can be run in either its own 'frame,' which makes it a separate application, or within the FoxPro 'frame,' in which case it becomes a set of individual windows which are available from the tools menu and which float over the desktop. The main benefits of using the debug frame are that it doesn't take precious screen space away from whatever you are running, all of the debugger windows are contained in a single top level window and are readily accessible and the "Fix" option works. The main drawback to using the debug frame is that if you have code in *LostFocus* or *Deactivate* events, that code will get fired when you switch to and from the debugger and may interfere with whatever you are trying to debug.

Setting up the debugger windows

The debugger's default configuration options can be found in the Visual FoxPro Options dialog (accessible from the Tools pad of the main menu) where they have their own tab. However, the interface for this page is non-standard and it is not immediately obvious that the items available for configuration depend on which window is specified by the option button in the 'Specify Window' section, as follows:

Table 13.1 *Debug window set-up options*

Window	Options
Call Stack	Font and Colors Call Stack Order Call Stack Current Line Indicator Call Stack Indicator
Locals	Font and Colors
Output	Font and Colors Log output to file (specify default file)
Trace	Font and Colors Show line numbers Trace between breakpoints Pause between line execution
Watch	Font and Colors

The significance of 'trace between breaks'

Note that for the Call Stack window to be available in the debugger, you need "Trace between break points" to be set ON but this option is, helpfully, located under the "Trace Window" options. However, unless you are actively inspecting the Call Stack, we strongly recommend that you keep this window closed. It is the slowest debugger window to refresh while stepping through code.

In fact the setting of "Trace between breaks" is the main factor in determining the impact of keeping the debugger open while running code. We prefer to leave it turned off by default, only setting it from the shortcut menu in the debugger when needed.

Close debug windows that you do not need!

The more debug windows you have open, the greater the impact on your code's execution time. It is worth noting that, when running the debugger in "debug frame", you do not actually need to have *any* of the individual debugger windows open, so long as the frame itself is active. If you have defined and enabled any required breakpoints, (including those defined in the Watch Window) the breakpoints will still fire as expected. This means that you can run your code with the absolute minimum of slowdown.

Setting breakpoints

Visual FoxPro allows for four types of breakpoints as shown in the following table. The main difference is that the breakpoints based on a location halt execution before the line of code executes, while those based on expressions take effect after the line of code has executed.

Table 13.2 *Types of breakpoints*

Breakpoint Type	**Takes Effect**
At Location	Immediately before the line of code is executed
At Location If expression is true	Immediately before the line of code is executed if condition is met
When expression is true	Immediately after the line of code that caused expression to become true
When expression is changed	Immediately after the line of code that caused expression to change

There are many ways of setting breakpoints in the debugger. Here are some that we find most useful:

- Right-click on a line of code while editing method code in the form or class designer or in a PRG and choose "Set Breakpoint" (applies to Version 6.0 with service pack 3). This sets a 'break at location' breakpoint.
- In the trace window, you can select any line of code, then right-click - Run To Cursor. This is a quickie breakpoint that's very useful for, for example, stopping after a loop terminates or at some other immediate juncture without setting a permanent breakpoint.
- In the trace window, click on the margin beside a line of code. This sets a 'break at location' breakpoint and displays a red dot in the margin to indicate the breakpoint. Double-click on the same line again, to clear the breakpoint.
- In the watch window click in the margin beside any watched expression. This sets a 'when the expression changes' breakpoint, and displays a red dot in the margin to indicate the breakpoint.
- From the debugger tools menu choose 'Breakpoints' to bring up a dialog where you can set any of the different breakpoint types explicitly. However, as far as we know, this is also, the *only* place in which you can set the **`PASS COUNT`** option for an 'at location' breakpoint. This is a useful one to remember if you want to set a breakpoint inside a loop but not have it executed until a specific number of iterations of the loop have occurred.
- Add an explicit "**`SET STEP ON`**" anywhere in any code. This will unconditionally suspend execution and display the debugger with the trace window open.

When running the debugger in its own frame, an additional dialog shows all currently defined breakpoints and allows you to selectively enable/disable them. Note that the 'Clear All' button not only disables all breakpoints, it also clears them from history.

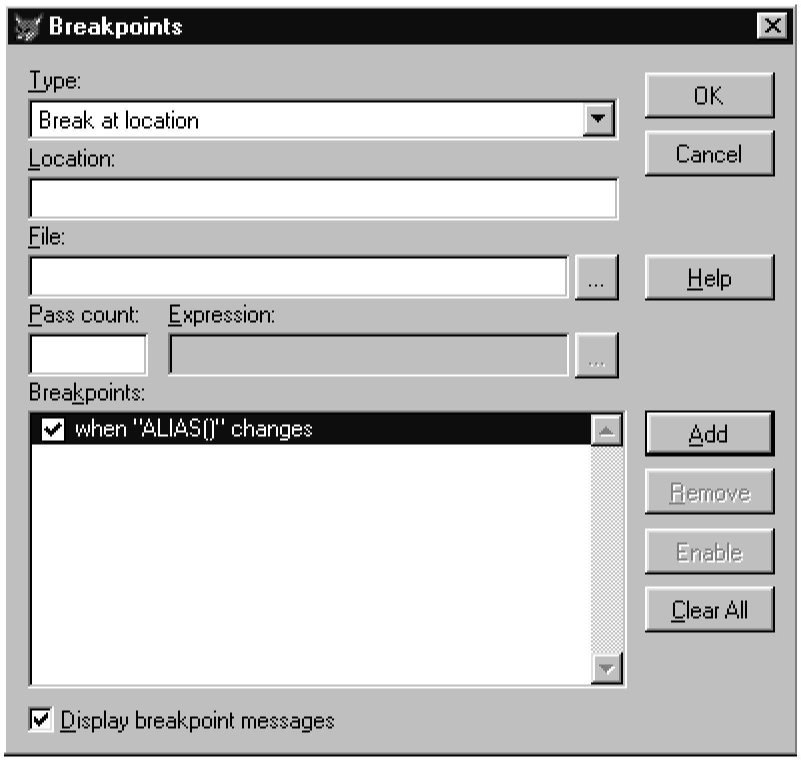

Figure 13.1 *BreakPoint dialog*

Despite the Help File stating that "**CTRL+B**" will display this dialog in either Debug or FoxPro frame, this dialog does not seem to be available when running the debugger in FoxPro Frame.

Useful breakpoint expressions

Often the most difficult decision is how to get Visual FoxPro to break at precisely the point that you want, without having to add explicit code to whatever you are testing. Here are a couple of suggestions we have found useful:

Use a "time based" breakpoint

Quite often we want to examine the state of our objects immediately after some user interaction - like immediately after a message box or other modal dialog. Enter a watch expression like this:

```
MINUTE( DATETIME() )
```

and set a breakpoint on it. Then run your test code and wait at the appropriate dialog until the minute changes before exiting from the dialog. The breakpoint will immediately halt execution and allow you to examine the state of the system. The same expression can also be used to halt large (or endless!) loops.

Use the program function

To set a breakpoint that will fire whenever a specified program or method is executed use an expression like:

```
"checkmethod" $ LOWER( PROGRAM() )
```

This will switch from .F. to .T. whenever "*checkmethod*" starts executing and by setting a 'when expression changes' breakpoint you can halt execution right at the start of the code block. This can, and should, be made very specific - especially when you want to break on a call to a native VFP method which may exist in many objects. For example, the following breakpoint will only interrupt when the *Click* method of an object named '*cmdnext*' begins to execute:

```
"cmdnext.click" $ LOWER( PROGRAM() )
```

while the following line will break in when any method of the same '*cmdnext*' object executes:

```
"cmdnext" $ LOWER( PROGRAM() )
```

Limiting breakpoints to changes

One of the problems with using a 'when expression changes' breakpoint is that Visual FoxPro will regard the watched expression as changed when whatever is being monitored comes into, or goes out of, scope. To avoid this and limit breaks to those occasions on which the value has *really* changed, use an **`IIF()`** function to return a constant from the watch expression unless the value has really changed. For example, the following expression will interrupt program execution whenever a variable named '*lnCnt*' is in scope and changes to or from a value of "*3*," but otherwise will be totally ignored:

```
IIF(TYPE("lnCnt")="N" AND lncnt = 3, .T., .F.)
```

Similarly the return value from a **`SET("DATASESSION")`** test can be used to ensure that when debugging with multiple datasessions open, only the correct one is being tracked by the debugger. Thus setting a breakpoint on a watch expression like this:

```
IIF( SET("DATASESSION") # 1, ALIAS(), "WRONG DS")
```

will break only when the currently selected **`ALIAS()`** changes in a data session **except** the Visual FoxPro default datasession (i.e. DataSession = 1).

How do I ensure that my custom methods fire when I expect them to?

One of the biggest problems that we all encounter when working with objects is that it is not always obvious when events fire. The purpose of the Event Tracker, which is built into the debugger, is to allow you to follow the sequence in which Visual FoxPro's native events fire. By enabling Event Tracking you can direct Visual FoxPro to echo the name of events and methods as they fire, to either the debug output window or to a text file. However, you cannot track all of Visual FoxPro's native events and methods and there is no provision at all for tracking custom methods. So if you need to track exactly what is being executed, you must either use the coverage logger or include code to create a log file into your classes.

Fortunately, the introduction of the *STRTOFILE()* function in Version 6.0 makes this very simple indeed. A single line of code is all that is needed, as follows:

```
STRTOFILE( PROGRAM() + CHR(10) + CHR(13), '<log file name>', .T. )
```

This will output the name of the currently executing method or procedure to the specified file. The final parameter ensures that the text is added to the target file if it already exists, otherwise *STRTOFILE()* would simply overwrite the log file.

We have used this technique to monitor exactly what code in a compiled application actually gets executed during 'user testing.' In order to do this selectively, we wrap the line of code that writes the log file in a test for a system variable (read from the application's *INI* file at start-up). Thereafter simply changing the setting in the INI file allows us to turn run time logging on and off.

Writing code for ease of debugging and maintenance

There have been many pages of good advice written to try convince programmers that they should adopt good defensive practices, use proper, standardized naming conventions and plan their code before writing it. Alas, all too often we still see things that make us wonder whether some programmers are even remotely bothered about being able to test, debug and even maintain their code. Here are some of our thoughts, for what they are worth, on writing better code.

Use a variable/property naming convention

There has been much debate in the FoxPro community over the years about such topics as whether the "m." prefix is a good thing to use or whether the standard Microsoft convention should be adopted for naming variables in all situations. We don't think that it matters too much what the details of your naming convention actually are, so long as you are consistent in its application. Why so?

The reason is that Visual FoxPro is a weakly typed language. In other words, it does not enforce data typing for its properties or variables. This means that without specific testing, you cannot be sure what type of data a variable actually holds because a variable derives its type from its data and changes to reflect whatever data it is given. This is, paradoxically, both one of Visual FoxPro's strengths, and also one of its great weaknesses.

It is a strength because it allows us to define a variable anywhere that we need one (without having to formally declare it) simply by assigning a value to a reference. This makes the programming language very flexible and easy to use.

It is a weakness for two reasons. First it means that there need not be a single place in a method, procedure or function where all the variables used are declared. This can lead to the same variable name being re-declared at different places within the code. Another consequence is that minor typographic errors cause errors at run time because what is actually a misspelled variable name is accepted by the compiler as a new variable declaration. Second, it means that even when a variable has been explicitly declared, named and initialized with a specific data type, the mere act of assigning data of an inappropriate type will not generate an error. The variable simply changes to accommodate the data.

To address the first issue, we strongly recommend you make a habit of declaring any variables that you create in a specific part of the program (normally right at the very start). There is no easy way to avoid the second problem except to ensure that when data is assigned to a variable, that variable's name correctly indicates the data type. This may result in more variables being created and used but, in our opinion, this is a small price to pay for clarity and ease of maintenance.

While using a naming convention will not actually prevent errors, we find it tends to make it easier to keep things conceptually separate and so minimizes the chance of introducing error. Whether you extend your naming convention to include Objects, Fields and Files is really up to you, but these things are generally less immediately significant. We feel it is better to ensure that there is proper documentation for a database and its tables, and for exposed properties and methods of classes.

Keep procedures and methods as short as possible

This may sound obvious, but less code means fewer bugs. More importantly it is easier to both manage the code and understand the logic when dealing with clearly focussed methods or procedures. As a general rule a method should handle one, and only one, specific piece of functionality. (Adopting this principle also makes it easier to name methods!) The greater the degree by which a method is overloaded, the more difficult it is to maintain and the greater the risk of something going wrong.

Use "return" statements in method and procedure code

Like most Visual FoxPro programmers we tend to be a little sloppy in our use of Return statements, especially when we are not actually returning a value explicitly. Most of the time this is not a problem. The FoxPro compiler is smart enough to accept that when it runs out of code in one place, it should return to the bit of code that started it off and it will implement an 'implicit' **`RETURN`** for us. However, this can cause problems when debugging code, especially in situations where the last line to be executed is either a function call or a call to another method. By adding the explicit **`RETURN`** statement, you have an opportunity to stop within the calling method when tracing code in the debugger.

Use asserts to help catch errors at development time

Asserts are extremely valuable to the developer because they allow us to handle problems differently at development and at run time without actually having to change any code. This is because Visual FoxPro will only execute a line of code that begins with an '**`ASSERT`**' statement if the code is running in development mode and **`SET ASSERTS`** is **`ON`**. In any other situation the line is treated as if it were a comment and does not interfere with program execution.

One common use of **`ASSERT`** is to warn developers (and testers) when something has not behaved as expected. The objective here is to provide additional information when testing in development. The following code snippet shows how this might be done:

```
luSomeVar = SomeProcessResult()
IF VARTYPE( luSomeVar ) # "C"
  ASSERT .F. MESSAGE 'SomeProcessResult has failed to return a Character
String'
  RETURN .F.
ENDIF
```

The error message will only ever be displayed when **SET ASSERTS** is on and the process fails to return a character string. In all other situations, if the process fails to return a character string, the code will simply return a logical **.F.** and continue its normal execution.

The second common use of Asserts is to check programming logic. This is slightly different from the first example in that the test, in this case, will only be performed when **ASSERTS** are enabled and so at run time the test will not even be attempted:

```
ASSERT PCOUNT() # 3 MESSAGE "Expected 3 parameters, received " +
PADL(PCOUNT(),2)
```

Keep processing and logic separate

One of the easiest ways of simplifying your code is to ensure that you do not mix up processing and logic. As an example of what we mean, consider the following code that was taken, as shown here, from an actual application:

```
IF cheknam(alltrim(table2.legal_name ))== cheknam(alltrim(;
thisform.pageframe1.page1.text1.value)) AND NOT cheknam(alltrim(;
table2.legal_name)) == cheknam(alltrim(curval('legal_name','BUYER'))
```

This may look very cool - but how on earth is one supposed to interpret and debug something like this? In fact, if you check this line of code carefully you will discover that there is actually a missing parenthesis! How *should* the statements have been written? Well maybe something like this would have been a little clearer:

```
*** Do all ChekNam() functions first
lcSceName = ChekNam( ALLTRIM( table2.legal_name ))
lcInpName = ChekNam( ALLTRIM( ThisForm.PageFrame1.Page1.Text1.value ))
lcLegName = ChekNam( ALLTRIM( CURVAL( 'legal_name', 'BUYER')))

*** Now do the test!
IF lcSceName == lcInpName AND NOT lcSceName == lcLegName
```

Why do we think this is better? There are three reasons:

- The calls to the *ChekNam()* function are handled separately. We can therefore check, (in the debugger even if we did not want to place checking code in the program) that the return values are really what we expect.
- We can actually reduce the number of calls that we make to the *ChekNam* function. The test requires that the value that we called *lcSceName* be used twice, so the original code has to make two calls to the function, passing the same value each time.

- We now have separated the program logic from the processing carried out by the *ChekNam()* function. This means that even without knowing what the *ChekNam()* function does, we can at least see what the **`IF`** statement is actually testing and can validate the results in the debugger.

Working with datasessions

We have not covered the use of datasessions specifically in any other part of the book - mainly because we could not decide whether the topic was related to data or to forms. Here are a few tips and tricks for using datasessions.

How do I share datasessions between forms?

It is actually very easy to get a form (or report) to use the private datasession of the object that called it. Simply set the child object's *DataSession* property to "1 - Default Data Session". This works because of the way Visual FoxPro interprets the term '*Default*' in this context and it isn't what you might reasonably expect!

When you first start Visual FoxPro and open the "DataSession Window", you will notice that the current datasession is "*Default(1)*". Setting a form's datasession property to "*1 - Default Data Session*" would seem likely to ensure that the form uses the same DataSession number 1 that VFP refers to as Default. Alas this is not necessarily true! In this context the term 'Default' really means that the form does **NOT** create a private datasession for itself but just uses whatever datasession is current *when it is instantiated*. This is why a form whose datasession property is left at "*1 -Default Data Session*" shares the data session of the form that launches it.

If a form uses its own dataenvironment to AutoOpen/AutoClose tables, only those tables that are actually opened by the child form will be closed. In other words, if the child form requires a table which is already open in the parent form's datasession, Visual FoxPro is smart enough to recognize that fact and will not re-open such tables when the child form is instantiated, or close them when the child form is released.

One thing to note is that if you allow a form to share in the private datasession of another, when you release the child form the Visual FoxPro Current DataSession name will change to '*Unknown(n)*', where 'n' is the datasession number of the form which originally created the datasession. However, this does not appear to cause Visual FoxPro any problem although it can be a little disconcerting when you first notice it happen. The reason appears to be that although Visual FoxPro is capable of re-naming the datasession to the current owner when the child form is instantiated, it does not know how to re-name it when that form is released and so simply leaves it as '*Unknown*'.

How do I change datasessions?

Despite the information to the contrary in the Help File, a form's *DataSessionID* property is actually read-write at run time. You can, therefore, force a form to run in a specific datasession by setting its *DataSessionID* property directly in code. However, if you have any bound controls on your form, changing the form's data session after they have been bound will cause you serious problems. The degree of severity will depend on the control in question. A grid will simply lose its *RecordSource* and go irrevocably blank. A list box whose *RowSource* is taken from a table will cause an '*Unable to Access Selected Table'* error and will disappear, while a combo box will simply go blank. Interestingly, re-connecting the form to the correct datasession will restore things to normal in the case of both list and combo boxes, though not for grids. The moral of this story is that if you need to change a form's datasession, do it in the form's LOAD before any controls have been instantiated, and do not allow the form's dataenvironment to AutoOpen tables.

There may, however, be occasions in which you will need to manipulate an object's *DataSessionID* property. For example, toolbars are often required to switch themselves into the datasession of the currently active form. (Our "managed" toolbar class in Chapter 10 has such code and this is not a problem because such generic toolbars do not have data bound controls.)

For objects that do not have a *DataSessionID* property, you must use the **`SET DATASESSION`** command to change the global datasession. This may be required for a global object (e.g. an Application Object) that is created in the Visual FoxPro default data session, but which may need access to tables opened by a form in a private datasession. Providing that you first save the current datasession and restore it immediately afterwards, this should not cause problems, even when other objects with data bound controls are present. The following code snippet shows how:

```
*** Save Current DS
lnDSID = SET( "DATASESSION" )
*** Change Datasession
SET DATASESSION TO <new session number>
*** Do whatever is needed and then revert
SET DATASESSION TO (lnDSId)
```

How do I get a list of all active datasessions? *(Example: getallds.prg)*

There is no native way to get a list of all active datasessions programmatically, but since we would normally only be interested in the datasessions being used by forms, we can use the *_Screen.Forms* collection to determine which are active. The following function does precisely this and populates an array (which is passed by reference) with the datasession number and the owning object name. The function returns the number of active datasessions that it finds:

```
**********************************************************************
* Program....: GetAllDS.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Populate array with all open datasessions
* ...........: Pass target array by reference to this function
* ...........:     DIMENSION aDSList[1]
* ...........:     lnNumSess = GetAllDs( @aDSList )
**********************************************************************
LPARAMETERS taSessions
EXTERNAL ARRAY taSessions
LOCAL lnCurDatasession, lnSessions
*** Initialize Counter
lnSessions = 0
*** Loop through Forms Collection
FOR EACH oForm IN _SCREEN.FORMS
  *** Have we got this session already?
  IF ASCAN( taSessions, oForm.DatasessionID) = 0
    *** If not, add it to the array
    lnSessions = lnSessions + 1
    DIMENSION taSessions[lnSessions,2]
    taSessions[lnSessions,1] = oForm.DatasessionID
    taSessions[lnSessions,2] = oForm.Name
  ENDIF
NEXT
*** Return the number of sessions
RETURN lnSessions
```

Miscellaneous items

The remainder of this chapter is a collection of things that we haven't specifically included elsewhere. There is no particular link between them, but they are worth mentioning, even if only as a reminder.

What is the event sequence when a form is instantiated or destroyed?

As is often the case with Visual FoxPro, the answer is that 'it depends.' In this case it depends upon whether the form is being instantiated from a *SCX* file using '`DO FORM`' or from a VCX file using `CREATEOBJECT()` or `NEWOBJECT()`. Instantiating the form as a class is the simplest, and here is the sequence of events:

```
FORM.LOAD
  INIT for each control
FORM.INIT
FORM.SHOW
FORM.ACTIVATE
FORM.REFRESH
  WHEN for 1st control in Taborder
  GOTFOCUS for 1st Control in Taborder (if it has one)
```

The basic start-up sequence is, therefore, given by the acronym "*L.I.S.A.R.*". By default, the individual controls on the form are instantiated in the order in which they were added to the class in the designer. However by using the '*Bring to Front*' and '*Send to Back*' options, you can alter the sequence in which controls are instantiated. (Although it really shouldn't matter in what order things are instantiated. Creating classes that rely on controls being instantiated in a particular order is not, in our opinion, good design!)

Releasing a form class is essentially the reverse of the initialization process. The *Release* method provides the means to programmatically initiate the process, while the *QueryUnload* method is called when a user clicks on the 'window close' button in a form. Neither calls the other unless you specifically add code to make them do so, but both call the form's *Destroy* method:

```
FORM.RELEASE or FORM.QUERYUNLOAD
FORM.DESTROY
  DESTROY for each control (in reverse order)
FORM.UNLOAD
```

This means that if you require code to be executed irrespective of whether the user closes a form by clicking on a command button (which calls *Release*) or the 'window close' button (which calls *QueryUnload*), then that code must be placed in the form's *Destroy* method.

In the case of a form created from a SCX, the basic sequence of form events is the same, but the presence of the native DataEnvironment complicates the issue. Notice that the dataenvironment *OpenTables* method is the first method called (it fires *BeforeOpenTables*) – even before the form's *Load* method is called. After the form *Load* method has completed successfully, the cursors are initialized. This ensures that when the form's controls are instantiated, those that are bound to data will be able to do so properly:

```
DATAENVIRONMENT.OPENTABLES
DATAENVIRONMENT.BEFOREOPENTABLES
FORM.LOAD
  INIT for each cursor in the DataEnvironment
DATAENVIRONMENT.INIT
  INIT for each control in the form
FORM.INIT
FORM.SHOW
FORM.ACTIVATE
FORM.REFRESH
  WHEN for 1st control in Taborder
  GOTFOCUS for 1st Control in Taborder (if it has one)
```

The release process is, as far as the form itself is concerned, identical to that above. Notice that the DataEnvironment is not actually destroyed until *after* the form has been unloaded from memory:

```
FORM.RELEASE or FORM.QUERYUNLOAD
FORM.DESTROY
  DESTROY for each control (in reverse order)
FORM.UNLOAD
DATAENVIRONMENT.AFTERCLOSETABLES
DATAENVIRONMENT.DESTROY
  DESTROY for each Cursor in the DE (in reverse order)
```

How do I get a reference to a form's parent form?

There are a couple of ways of doing this, the most obvious being simply to have the parent form pass a reference to itself when calling the child form - thus:

```
DO FORM <child form> WITH This
```

However this does require that the child form's *Init* method must be set up to receive the object reference as a parameter and then store it to a form property. Interestingly the object pointed to by Visual FoxPro's *_Screen.ActiveForm* property does not change when a new form is instantiated until that form's *Init* method has completed successfully. (This makes sense when you remember that returning a logical **.F.** from either the *Load* or *Init* methods will prevent the new form from instantiating.)

Therefore to get a reference to the calling form, there is no need to pass anything at all. Simply store *_Screen.ActiveForm* to a property in either the *Load* or *Init* method of the child form. Normally we would place this sort of code in the *Load* method (to leave the *Init* free for handling parameters) like this:

```
IF TYPE("_Screen.ActiveForm.Name") # "U"
  ThisForm.oCalledBy = _Screen.ActiveForm
ENDIF
```

Note that we cannot reliably use the **VARTYPE()** function here because it will fail if there is no active form.

How do I get a list of all objects on a form?

The Visual FoxPro Form object has a collection named "*Controls*" and an associated counter property ("*ControlCount*") holds a reference to every object on the form. However, this collection only holds references to objects that are *DIRECTLY* contained by the form. So, for example, it will not include objects that are on a page, inside a pageframe on the form. All it will have is the reference to the pageframe. To get a list of all objects, we will therefore need to 'drill down' into each container that is encountered.

Fortunately every Visual FoxPro container class has a collection (and associated counter) which holds references to the objects that the container owns. Unfortunately, these collections are not all named the same as the following table indicates:

Table 13.3 *Container class collections*

Base Class	Collection	Counter Property
_Screen	Forms	FormCount
_Screen	Controls	ControlCount
Formset	Forms	FormCount
Form	Controls	ControlCount
Toolbar	Controls	ControlCount
PageFrame	Pages	PageCount
Page	Controls	ControlCount
Grid	Columns	ColumnCount
Column	Controls	ControlCount
Container	Controls	ControlCount
Custom	Controls	ControlCount
Control	Controls	ControlCount

This variation in naming makes writing a routine that will drill down through a form a little more difficult because we must test the baseclass of each container that we encounter in order to ascertain what its collection property is called. The sample form *LogCtrls.Scx* has a recursive custom method (*GetControls*) and a custom array property (*aAllControls*) which is used as the collection for all controls and is populated by the custom *AddToCollection* method. Finally, a custom *RegisterControls* method is used to initialize the array property and to start the drill down process by calling the *GetControls* method with a reference to the form itself. Here is the code for the *RegisterControls* method:

```
*** LogCtrls::RegisterControls Method
*** Initialises Form Collection and Calls the recursive GetControls()
WITH ThisForm
    *** Clear the current list (if any)
    DIMENSION .aAllControls[1,3]
    .aAllControls = ""
    *** Start the Drill-Down with the Form Object itself
    .GetControls( This )
ENDWITH
```

The *GetControls* method is a little more complex. The first thing it does is to create a local reference to the object passed to it as a parameter (note that it will use the Form itself if nothing is passed). Next it calls the *AddToCollection* method to add the object to the collection and then stores the object's base class into another local variable for use later:

```
*** LogCtrls::GetControls Method
*** Drills down through form and populates the custom collection array
LPARAMETERS toStartObj
LOCAL loRef, lnCnt, lnControls, loObj

*** Get ref to parent or to form, by default
loRef = IIF( TYPE('toStartObj')='O', toStartObj, THISFORM )
*** Add This object to the collection
ThisForm.AddToCollection( loRef )

*** Get the Base Class of the current object
```

```
lcClass = LOWER(ALLTRIM(loRef.BaseClass))
```

Next we need to determine what sort of object we are dealing with in the current iteration. First we check to see whether we are dealing with one of the classes which use something other than a 'controls' collection. If so we call the *GetControls* method recursively while looping through that object's collection:

```
*** Now Process the current object
DO CASE
  CASE lcClass = 'pageframe'
    FOR lnCnt = 1 TO loRef.PageCount
      *** Call this method for each page
      THISFORM.GetControls( loRef.Pages[lnCnt] )
    NEXT

  CASE lcClass = 'grid'
    FOR lnCnt = 1 TO loRef.ColumnCount
      *** Call this method for each column
      THISFORM.GetControls( loRef.Columns[lnCnt] )
    NEXT

  CASE lcClass = 'formset'
    FOR lnCnt = 1 TO loRef.FormCount
      *** Call this method for each form
      THISFORM.GetControls( loRef.Forms[lnCnt] )
    NEXT
```

Any other container class will be using a "Controls" collection so we can process them all in a single case statement. (Notice that we check, using an exact comparison, for the '*page*' base class. This is to avoid falling foul of Visual FoxPro's slightly idiosyncratic string comparison which would return `.T.` if we simply included '*page*' in the list, but the object was a '*pageframe*'.)

If the current object is a container, this code loops through its *controls* collection. Again we check the base class of each object that we encounter and, if it is another container, call the *GetControls* method recursively passing a reference to the object. However, if it is not a container, a reference to it is passed to the *AddToCollection* method so that it can be logged:

```
*** OK, is it an object which has a controls collection?
  CASE INLIST( lcClass, 'form', 'container', 'column', 'custom', 'control' ) ;
             OR lcClass == 'page'
    *** If so, loop through its collection
    FOR lnCnt = 1 TO loRef.ControlCount
      *** Get a reference to the current object
      loObj = loRef.Controls[lnCnt]
      IF INLIST( loObj.BaseClass, 'Container', 'Pageframe', 'Grid', 'Custom',
'Control' )
        *** Call this method recursively if it is a contained container
        ThisForm.GetControls( loObj )
      ELSE
         *** Just add the object to the collection
         ThisForm.AddToCollection( loObj )
      ENDIF
    NEXT
```

If the current object does not trigger any of the conditions in this case statement, it is not a container and has already been logged so we can safely ignore it and exit from this level of recursion:

```
  OTHERWISE
    *** Nothing more to do at this level
ENDCASE

*** Just return
RETURN
```

The only other code is that which actually adds an object to the form's 'aAllControls' collection. This is very simple indeed because it receives a direct reference to the object that it has to log as a parameter from the *GetControls* method:

```
LPARAMETERS toObj
LOCAL loRef
IF VARTYPE( toObj ) # "O"
    *** If not an object, just return
    RETURN
ELSE
    *** Get Local Reference
    loRef = toObj
ENDIF
```

After checking that the parameter is indeed an object, the next task is to determine how many items have already been logged:

```
WITH ThisForm
    *** Get number of rows already in collection
    lnControls = ALEN( .aAllControls, 1 )
  *** If 1 row - is it populated?
    IF lnControls = 1
        lnControls = IIF( EMPTY( .aAllControls[1,1]), 0, 1 )
    ENDIF
    *** Increment the counter
    lnControls = lnControls + 1
```

A new row is then added to the collection and the elements of the collection populated:

```
    *** Add a row to the array
    DIMENSION .aAllControls[ lnControls, 3]
    *** Populate the new row
    .aAllControls[ lnControls, 1] = loRef.Name        && Object Name
    .aAllControls[ lnControls, 2] = loRef             && Object Reference
    .aAllControls[ lnControls, 3] = SYS(1272, loRef ) && Object Hierarchy
ENDWITH
*** Just Return
```

```
RETURN
```

There are many situations in which it is useful to be able to loop through all controls on a form. While this code is specifically written to populate a collection array, with the exception of the two calls to the *AddToCollection* method, the code is completely generic and could be used anytime it's necessary to drill down through a form. Furthermore, since the *GetControls* method is called with an object reference, it does not have to start with the form. It can just start with whatever object reference it is passed.

How can I set focus to the first control in the tab order?

One answer to this question is that you could use code similar to that given in the preceding section to drill down through a container (i.e. the form, or a page in a form) to find the contained object that has its *TabIndex* property set to 1. Then simply set focus to that object and exit.

Alternatively when using forms with multiple pages, rather than repeatedly executing this drill down code, you might prefer to build a special collection (when the form is instantiated) to record for each page a reference to the object which is first in the Tab Order. Then all that would be needed would be to scan that collection for the required page and set focus to the specified object.

How do I return a value from a modal form?

Elsewhere, we have already discussed some techniques for passing parameters between objects of varying types. What we have not covered specifically anywhere else are the various techniques for getting values back from a modal form. The concept of "*returning a value*" is only meaningful when the form returning the value is modal because the implicit requirement is that some process must be interrupted, or suspended, until the required value is returned. Only by using a modal form can you ensure that:

- The process cannot continue until the value is available
- The value is returned to the correct place in the calling code.

There are essentially three ways of getting a value back from a modal form, but one of them works only for forms which are run as SCX files, one works with forms that are instantiated from VCX files and one works irrespective of how the form is created. Remember that, although we are talking about returning '*a value*', this 'value' could be a parameter object containing multiple items (see *Chapter 2 "Functions and Procedures"* for more information on creating and using parameter objects).

Returning a value from a form

The actual mechanism for returning a value from a form is quite straightforward. You simply place a **`RETURN <value>`** command as the last line of code in the form's *UNLOAD* method. This is the last form method to be executed before a form is released and so it is a perfectly logical place for the return statement. However, there is one little catch. If the value that you wish to return is coming from a control on the form, by the time the form's Unload method runs all controls have been released, so the values that they held will no longer be available. To get around this problem you must ensure that any control values you wish to return are saved to form properties no later than in the Form's *Destroy* method. (See the section earlier in this chapter for details of the event sequence when a form is instantiated or destroyed.)

Hiding a modal form

One way of getting access to values that are contained within a modal form is simply to hide the form instead of releasing it. When a modal form is hidden, whichever form was active immediately prior to the modal form being instantiated becomes the active form once again. In other words, the form that *called* the modal form is re-activated. Providing that you have, within the calling form, a valid reference to the modal form you can access any exposed properties of the form, or of its contained controls. This approach will work irrespective of the way in which the form is instantiated.

The following code snippet shows how this might be done for a form instantiated directly from a class:

```
*** Instantiate a modal form
oFm = NEWOBJECT( 'modalform', 'formclasses' )
*** Show the form and ensure that it is modal
oFm.Show(1)
*** When form is 'released' it is actually hidden!
*** Access the Modal form's properties directly
ThisForm.SomeProperty = oFm.ModalFormProperty
*** Release the modal form when done
oFm.Release()
```

While for a form that is created from an SCX file, the following code is equivalent:

```
*** Instantiate the modal form
DO FORM modalform NAME oFm LINKED
*** When the modal form is "released" it is actually hidden!
*** The NAME 'oFm' can now be used to access it directly:
ThisForm.SomeProperty = oFm.ModalFormProperty
*** Release the modal form when done by releasing the "linked name"
RELEASE oFm
```

Using do form <name> to <variable>

For modal forms that have been created as SCX files and which are run using the **`DO FORM`** command, there is a specific syntax you use to save a value which is returned from the form, as follows:

```
DO FORM modalform TO luRetVal
```

When the modal form is released, whatever was returned from the form's *Unload* method will be saved to the variable 'luRetVal'. Note that this variable does not need to have been previously declared and will be created as needed. However, if the called form does not contain a **`RETURN`** statement in its *Unload* method, the variable will **`NOT`** be created. So, in our opinion, it is much safer to always explicitly declare the return variable, and initialize it, rather than just relying on there being a return statement.

Returning a value from a form instantiated directly from a class

The problem in this situation is that there is no way to return both the object reference *AND* a return value from either the **`CreateObject`** or **`NewObject`** functions. Since both must return a reference to the new object we have to find another way of getting a value back. The solution is to pass a parameter object to the form that can then be returned by the modal form when it is released.

The form class must be set up to receive, and store to a property, the parameter object that is passed to it. (Normally we would also have the class *Init* method call its *Show* method directly to make the form visible immediately on instantiation.). This object must be populated with the relevant properties while the form is active and returned to the calling method (or procedure) from the modal form's *Unload* method. The code to instantiate the modal form would look like this:

```
*** Create the Parameter object
oParamObj  = CREATEOBJECT( 'parameterobject' )
*** Instantiate the modal form
oModalForm = CREATEOBJECT( 'modalformclass', oParamObj )
*** Check the returned object properties
IF oParamObj.FormWasOK
  *** Do whatever
ELSE
  *** Do something else
ENDIF
```

How do I change the mouse pointer in a form while a process is running?

As usual, the basic answer is very simple. All visual controls have a *MousePointer* property that determines the shape of the mouse cursor when the cursor is positioned over a control. However, because each control has its own setting for controlling the mouse pointer there is no single property or method to control the *MousePointer* property for all controls on a form at once.

The standard way to change all the values of a property for all objects on a form is to use the form's *SETALL* method. The following code sets the mouse pointer for all controls on a form to the 'hourglass':

```
WITH ThisForm
  *** Set the form's own mouse pointer
  .MousePointer = 11
  *** Now all contained objects
  .SetAll( 'MousePointer', 11 )
ENDWITH
```

To restore to the default setting, simply repeat this code with a value of 0 instead of 11. However, this does rely on all controls on the form using the same *MousePointer* property setting at all times. If you already have different *MousePointer* property settings for different classes of control, the only alternative is to loop through all controls on a form and save each control's current *MousePointer* property and set it explicitly to the required value. (You can use code similar to that shown in the "*How do I get a list of all objects on a form?*" section of this chapter.). To restore the mouse pointer you would simply have to repeat the process and read the saved value. We feel this is a fairly unusual scenario and would normally expect to find all controls using their 'default' (*MousePointer = 0*) setting.

So far, so good! Unfortunately there is an exception to everything that we have said above when a grid is involved. While a grid does have a *MousePointer* property, we are not quite sure why. In Version 6.0, anyway, it does not seem to behave the same way as other controls and only affects the display when the mouse is over an area of the grid which does not contain any data. No matter what the grid's *MousePointer* property is set to, moving the mouse over the populated portion of the grid always displays the "I" beam cursor.

The best solution that we can come up with is to add a transparent *SHAPE* object to cover the grid (except for the scrollbars). The result is that it is actually the shape object's *MousePointer* that the user sees when they move their mouse pointer over the grid. Of course if the grid is not Read-Only, we must provide some mechanism for detecting a click on the shape and transferring it to the relevant portion of the grid. Fortunately some new functions in Version 6.0 can help us out here. The sample code includes a form ("*ChgMPtr.scx*") which shows how this works. Here is the code from the Click method of the shape that overlays the grid:

```
*** We need to know the name of the grid
WITH ThisForm.grdVatrates

  *** Use AMOUSEOBJ() to get details of the mouse position
  LOCAL ARRAY laList[1]
  AMOUSEOBJ( laList, 1 )

  *** X and Y co-ordinates are in Rows 3 and 4
  lnX = laList[3]
  lnY = laList[4]

  *** Initialise some variables
  lnGObj = 0
  lnGrow = 0
  lnGCol = 0

  *** Use GRIDHITTEST() to get Grid Row/Col under the mouse
  llStat = .GridHitTest(lnX, lnY, @lnGobj, @lnGRow, @lnGCol)

  *** Send Shape behind the grid
  This.ZORDER(1)

  *** Activate the correct cell in the grid
  .ActivateCell( lnGRow, lnGCol)
  .SetFocus()

ENDWITH
```

This code determines where in the shape the click occurred and translates that position into the corresponding row and column of the grid. We then drop the shape behind the grid and activate the correct cell. The only trick is that we need to restore the shape to its 'On Top' position when the grid loses focus, but the Grid has no *LostFocus* method! So we have to add code to the grid's *Valid* method instead, and call the grid's own *ZOrder* method to send the grid 'To Bottom' thereby restoring the shape object to its original position over the grid.

How can I create a 'global' property for my application?

In Visual FoxPro, properties are scoped to objects so it is not really possible for a property to be "global" in the same way that a memory variable can. The best we can do is to define the property as belonging to an object whose scope is global. Given this approach we have a couple of options.

First we could simply create an object of the required class and associate its reference with a Public variable, like this:

```
RELEASE goGlobalObject
PUBLIC goGlobalObject
goGlobalObject = NEWOBJECT( <class>, <classlibrary> )
```

Anything in the application now has access to '*goGlobalObject*' and hence to all of its properties and methods. This is how an 'Application Object' (sometimes known as a 'g*od*' object) is usually created. Of course if the object is created in the startup program of the application, there is no need to explicitly declare its reference as 'Public', providing that it is not explicitly declared as 'Local'. Any private variable created in the startup program is effectively 'public' to the application anyway.

With the introduction into the language (in Version 6.0) of the *AddProperty* method, an intriguing alternative to creating a global object was opened up. The Visual FoxPro "*_Screen*" object is actually a ready-made global object and since it has an *AddProperty* method of its own, we can simply add any properties that we require globally directly to the screen object as follows:

```
_Screen.AddProperty( 'cCurrentUser', '' )
```

But wait, we hear you cry, what happened if you run your application with the screen turned off? Actually it makes no difference at all. The *_Screen* object is still available even if you do not show it and you can still access its properties and methods – including any custom properties or those of added objects – at any time.

How can I 'browse' an array? *(Example ArToCurs.prg)*

One of Visual FoxPro's minor irritants is that there is no good way to actually see the contents of an array. There are, of course, several ways of getting at an array. You can use the debugger to expand and drill down through an array or you can list it to screen, or to a text file, but none of them are entirely satisfactory. After all, we can create an array from a cursor using SQL by simply issuing a command like this:

```
SELECT <fields> FROM <table> INTO ARRAY <name>
```

What we cannot do is the opposite – turn an array back into a cursor so that we can browse it, or view it in a form or whatever else we may need at the time. The *ArToCurs* function does the job for us by taking a reference to an array, and optionally a cursor name, and building a cursor from the contents of the array.

There is a caveat for the code as presented here. This function will not handle arrays containing object references. It would not be difficult to amend the code so that it did (maybe by getting the *name* of the object), but this function was not intended for that purpose so it is not written that way. The return value is the number of rows in the cursor that was created. Here is the code:

```
**********************************************************************
* Program....: ArToCurs.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Accepts an array and converts it into a cursor
**********************************************************************
LPARAMETERS taSceArray, tcCursorName
LOCAL ARRAY laStru[1]
LOCAL lnRows, lnCols, lnCnt, lcColNum, lnColSize, lcInstr
EXTERNAL ARRAY taSceArray

*** Check that we have an array as a parameter
*** NB Cannot use VarType() here in case array does NOT exist!
IF TYPE( "taSceArray[1]" ) = "U"
  ASSERT .F. MESSAGE "Must pass a valid Array to ArToCurs()"
  RETURN
ENDIF

*** Default Cursor Name to "arraycur" if nothing passed
lcCursor = IIF( VARTYPE( tcCursorName ) = "C" AND ! EMPTY( tcCursorName ), ;
                ALLTRIM( LOWER( tcCursorName )), "arraycur" )

*** Determine the size of the array
lnRows = ALEN(taSceArray,1)
lnCols = MAX( ALEN(taSceArray,2), 1 )
DIMENSION laStru(lnCols, 4)

*** Create the structure array
lcInstr = ""
FOR lnCnt = 1 TO lnCols
  *** Name Columns with the Data Type + Zero Padded number
  lcColNum = PADL( lnCnt, 5, "0" )
  laStru[ lnCnt, 1 ] = VARTYPE( taSceArray[ 1, lnCnt] ) + lcColNum
  laStru[ lnCnt, 2 ] = "C"                        && Data Type
  *** Determine Maximum Column width needed
  lnColSize = 1
  FOR lnRowCnt = 1 TO lnRows
    lnColSize = MAX( lnColSize, LEN( TRANSFORM( taSceArray[lnRowCnt, lnCnt] )))
  NEXT
  laStru[ lnCnt, 3 ] = lnColSize                  && Col Width
  laStru[ lnCnt, 4 ] = 0                          && No Decimals
  *** Add the field to the Insert String
  IF ! EMPTY( lcInstr )
    lcInStr = lcInstr + ","
  ENDIF
  IF lnCols > 1
    lcInStr = lcInstr+"TRANSFORM(taSceArray[lnCnt,"+ALLTRIM(STR(lnCnt))+"])"
  ELSE
   lcInStr = lcInstr+"TRANSFORM(taSceArray[lnCnt"+"])"
  ENDIF
NEXT

*** Create the cursor from the structure array
CREATE CURSOR (lcCursor) FROM ARRAY laStru
```

```
*** Populate the cursor
FOR lnCnt = 1 TO lnRows
  INSERT INTO (lcCursor) VALUES ( &lcInStr )
NEXT
GO TOP IN (lcCursor)

*** Return Number of records
RETURN RECCOUNT( lcCursor )
```

Windows API Calls

There are an awful lot of these and they are, alas, not well documented for Visual FoxPro users. Here are some Visual FoxPro functions which use API calls that we have found useful. Although presented here as stand-alone functions, normally we would collect these sort of functions into either a procedure file, or as methods of a class. The benefit of using a visual class (e.g. a '*custom*' class) is that when these functions are needed, an object based on the class can simply be added directly to the form that needs it.

One of the biggest problems for most Visual FoxPro developers when beginning to work with the Windows API is that the functions rely heavily on defined constants. But it is not always easy to determine where these constants actually come from, or even what they are. Gary DeWitt has gleaned, and very generously made available to everyone, over 4000 Windows constants as Visual FoxPro "**#DEFINE**" statements. A copy of his file ('*windows.h*') is included with the sample code for this chapter.

How do I find the file associated with a file type? *(Example: findexec.prg)*

This little function is useful because it tells you *where* the executable file associated with a given file extension is located. For any file extension that has a specific Windows association, the return value consists of the full path and file name to the executable program. This can then be manipulated using the native Visual FoxPro functions (**JUSTPATH(), JUSTFNAME()** etc) to extract whatever information you really need.

Note that the API function that we are using here (*FindExecutable()*) can accept a specific path and file name, but the file must exist. So, to be absolutely certain, we have opted to create a temporary file, with the required extension, in the current working directory and use that file to determine the result. However, this means that passing any of the Windows executable extensions (i.e. '**COM**', '**BAT**' or '**EXE**'), which are not associated to specific applications, will simply return the location of this temporary file. This is not a problem, since the whole purpose of the function is to determine where the executable program which runs a non-executable file type is located:

```
**********************************************************************
* Program....: FindExec.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Returns the full path and file name of the windows exe
* ...........: which is associated with the specified function
**********************************************************************
LPARAMETERS tcExt
LOCAL lcRetVal, lcFileExt, lcFileName, lnFileHandle, lcDirectory, lcResBuff
STORE "" TO lcRetVal, lcFileExt, lcFileName, lcDirectory
*** Check that an extension is passed
IF VARTYPE( tcExt ) # "C" OR EMPTY( tcExt )
```

```
  ERROR "9000: Must pass a file extension to FindExec()"
  RETURN lcRetVal
ELSE
  lcFileExt = UPPER( ALLTRIM( tcExt ))
ENDIF

*** This function MUST have a file of the requisite type
*** So create one right here (just temporary)!
lcFileName = "DUMMY." + lcFileExt
lnFileHandle = FCREATE( lcFileName )
IF lnFileHandle < 1
  *** Cannot create file
  ERROR "9000: Unable to create file.  FindExec() must stop" ;
      + CHR(13) + "Check that you have the necessary rights for file creation"
  RETURN lcRetVal
ENDIF
FCLOSE( lnFileHandle )

*** Create the return value buffer and declare the API Function
lcResBuff = SPACE(128)
DECLARE INTEGER FindExecutable IN SHELL32 ;
         STRING @cFileName, ;
         STRING @cDirectory, ;
         STRING @cBuffer

*** Now call it with filename and directory
lnRetVal = FindExecutable( @lcFileName, @lcDirectory, @lcResBuff)

*** Check the return value
lcMsgTxt = ""
DO CASE
  CASE lnRetVal = 0
    lcMsgTxt = "Out of memory or resources"
  CASE lnRetVal = 2
    lcMsgTxt = "Specified file not found"
  CASE lnRetVal = 3
    lcMsgTxt = "Specified path not found"
  CASE lnRetVal = 11
    lcMsgTxt = "Invalid EXE format"
  CASE lnRetVal = 31
    lcMsgTxt = "No association for file type " + lcFileExt
  OTHERWISE
    *** We got something back
    *** String is Null-terminated in the result buffer so:
    lcRetVal = LEFT(lcResBuff, AT(CHR(0), lcResBuff) - 1)
ENDCASE

*** Delete the dummy file we created.
DELETE FILE (lcFileName)

*** Display Results and return
IF ! EMPTY( lcMsgTxt )
  MESSAGEBOX( lcMsgTxt, 16, "FindExec Failed" )
ENDIF
```

```
RETURN lcRetVal
```

How can I open a file using Windows file associations? *(Example: runfile.prg)*

This is remarkably simple using the *ShellExecute()* function. This function either opens or prints the specified file (which can either be an executable file or a document). Here is the code:

```
**********************************************************************
* Program....: RunFile.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Open or Print a named file/document using windows association
**********************************************************************
LPARAMETERS tcDocName, tlPrint
LOCAL lnRetVal, lnShow, lcAction
*** Check Parameters
IF VARTYPE( tcDocName ) # "C" OR EMPTY(tcDocName)
    WAIT WINDOW "Must Pass a valid document name and extension" NOWAIT
    RETURN
ENDIF
*** Must have an Extension too
IF EMPTY( JUSTEXT( tcDocName ))
  WAIT WINDOW "Must Pass a valid document name and extension" NOWAIT
  RETURN
ENDIF

*** Check action, if tlPrint = .T., the "Print" otherwise "Open"
lcAction = IIF( tlPrint, "Print", "Open" )
lnShow   = IIF( tlPrint, 0, 5 )

*** Declare API function
DECLARE INTEGER ShellExecute IN Shell32.dll ;
  LONG HWnd, ;
  STRING cAction, ;
  STRING cFileName, ;
  STRING cParameters, ;
  STRING cPath, ;
  INTEGER nShowWindow

*** Now execute it
lnRetVal = ShellExecute( 0, lcAction, tcDocName, "", "", lnShow)

RETURN
```

Note that we have set this function up to accept a fully qualified path and file name as a single parameter. In fact you could use it equally well by splitting the parameter into file name and path and passing them separately as the *cFileName* and *cPath* parameters.

The *nShowWindow* value is set to 5 (i.e. '*Show*' the application) when opening a document, and to 0 (hide the application) when printing a document. (The full range of values can be found in the *WinUser.h* file under the "*ShowWindow() Commands*" heading.)

How can I get the user's Windows log-in name? *(Example: getlogin.prg)*

There is actually a purely VFP way of getting this information by using the *SYS(0)* function which returns the machine name and current user log-in name. The return value is a single string and uses the "#" symbol to separate the machine name from the user name. So one possible solution is to use:

```
lcCurrentUser = ALLTRIM( SUBSTR( SYS(0), AT('#', SYS(0))+1))
```

However there is also a *GetUserName()* function which will return the same information and which can be wrapped as a simple, user defined function as follows:

```
****************************************************
* Program....: GetLogIn.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Get Windows Log-In Name
****************************************************
LOCAL lcUserName, lcRetVal

*** Declare API Function
DECLARE GetUserName IN Win32Api ;
    STRING @cString, ;
    INTEGER @nBuffer

*** Initialise the buffers
lcUserName = SPACE(50)

*** Get the Login Name
GetUserName( @lcUserName, LEN(lcUserName) )

*** String is Null-terminated in the result buffer so:
lcRetVal = LEFT(lcUserName, AT(CHR(0), lcUserName) - 1)

*** Return Login ID
RETURN lcRetVal
```

How can I get directory information? *(Example: windir.prg)*

The Windows API contains several functions that can be used to get directory information. Some are also available from within Visual FoxPro (e.g. changing the current directory) while others are not (e.g. finding the *Windows* or *System* directories). The following program collects several of these functions together and uses a numeric index to determine the action required as follows:

Table 13.4 *API directory functions*

Index	Action
1	Returns the full path to the Windows System Directory.
2	Returns the full path to the Windows Home Directory.
3	Returns the full path to the current working directory. (Equivalent to "CD" in Visual FoxPro).
4	Accepts additional parameter which is either a relative path, or a fully qualified path, and makes that the current working directory. Returns either the fully qualified current directory or an error message if the specified directory does not exist.
5	Accepts additional parameter which is either a relative path, or a fully qualified path, and creates the directory. Returns either 'Created' or 'Failed'.
6	Accepts additional parameter which is either a relative path, or a fully qualified path, and deletes the directory. Returns either 'Removed' or 'Failed'.

It is worth noting that these functions will handle either UNC or conventional drive identifiers with equal facility that may make them useful in some situations. Here is the code:

```
**********************************************************************
* Program....: WinDir.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Windows API Directory Functions
* ...........: Calling Options
* ...........: 1  -> Return Windows System Directory
* ...........: 2  -> Return Windows Directory
* ...........: 3  -> Return Current Working Directory
* ...........: 4, <path> -> Set Working Directory (Accepts Relative Path)
* ...........: 5, <path> -> Create Named Directory (Accepts Relative Path)
* ...........: 6, <path> -> Remove Named Directory (Accepts Relative Path)
**********************************************************************
LPARAMETERS tnWhich, tcDirName
LOCAL lcSysDir, lnBuffer, lnDirLen, lcRetVal
*** Initialize the buffers
lcSysDir = REPLICATE(CHR(0),255)
lnBuffer = 255
*** Execute Appropriate call
Do CASE
    CASE tnWhich = 1
        *** Windows system directory
        DECLARE INTEGER GetSystemDirectory IN Win32API ;
            STRING @cBuffer, ;
            INTEGER nSize
        *** Call the function
        lnDirLen = GetSystemDirectory( @lcSysDir, lnBuffer )
        lcRetVal = LEFT( lcSysDir, lnDirLen )

    CASE tnWhich = 2
        *** Windows system directory
        DECLARE INTEGER GetWindowsDirectory IN Win32API ;
            STRING @cBuffer, ;
            INTEGER nSize
        *** Call the function
        lnDirLen = GetWindowsDirectory( @lcSysDir, lnBuffer )
        lcRetVal = LEFT( lcSysDir, lnDirLen )
    CASE tnWhich = 3
        *** Current working directory
        DECLARE INTEGER GetCurrentDirectory IN Win32API ;
            INTEGER nSize, ;
            STRING @cBuffer
        *** Call the function
        lnDirLen = GetCurrentDirectory( lnBuffer, @lcSysDir )
        lcRetVal = LEFT( lcSysDir, lnDirLen )

    CASE tnWhich = 4
        *** Set Default Directory
        DECLARE INTEGER SetCurrentDirectory IN WIN32API ;
            STRING cNewDir
        *** Call the function, return name if OK, empty string if not
        lcRetVal = IIF( SetCurrentDirectory( tcDirName) = 1, tcDirName, ;
                                              "Directory does not exist")

    CASE tnWhich = 5
        *** Create Directory
        DECLARE INTEGER CreateDirectory IN WIN32API ;
            STRING cNewDir, ;
            STRING cAttrib
        *** Call the function
        lnSuccess = CreateDirectory ( tcDirName, "")
        lcRetVal = IIF( lnSuccess = 1, "Created", "Failed" )

    CASE tnWhich = 6
        *** Remove Directory
        DECLARE INTEGER RemoveDirectory IN WIN32API ;
            STRING cKillDir
        *** Call the function
        lnSuccess = RemoveDirectory( tcDirName)
        lcRetVal = IIF( lnSuccess = 1, "Removed", "Failed" )

    OTHERWISE
        *** Unknown Parameter
        lcRetVal = ""
ENDCASE
*** Return the directory location
RETURN lcRetVal
```

Note that the "remove directory" option will only operate if the target directory is empty of all files.

How can I get the number of colors available? *(Example: wincols.prg)*

The number of colors can be calculated from the number of color bits that are allocated for each pixel. This is one (of many) values that can be obtained using the *GetDeviceCaps()* function. However, in order to determine where to get its values, this function needs to be passed the context id (which is obtained from the *GetDC()* function) to get which we need the Windows "*WHnd*" handle. This can be obtained either by using the FoxTools library like this:

```
SET LIBRARY TO FoxTools.fll ADDITIVE
```

```
lnHWND = _WHTOHWND( _WMainWind() )
```

or, as here, the API function *GetActiveWindow()* can be used to return the handle to the main FoxPro window:

```
**********************************************************************
* Program....: WinCols.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Returns the number of colors available
**********************************************************************
LOCAL lnHWND, lnBitsPixel, lnDeviceContext

*** Declare API Functions
DECLARE INTEGER GetActiveWindow IN WIN32API

DECLARE INTEGER GetDC IN Win32Api ;
               INTEGER nWHnd

DECLARE INTEGER GetDeviceCaps IN Win32Api ;
               INTEGER nDeviceContext, ;
               INTEGER nValueToGet

*** Get the Windows Handle for the Active Window
lnHWND = GetActiveWindow()

*** First get the device context for the current window
lnDeviceContext = GetDC( lnHWND )

*** Then get number of color bits per pixel
lnBitsPixel = GetDeviceCaps( lnDeviceContext, 12)

*** Return Result
RETURN (2 ^ lnBitsPixel)
```

How do I get the values for Windows color settings? *(Example: getwcol.prg)*

There are two parts involved in this process. Firs we need to retrieve the color setting for the required Windows item from Windows itself. (There is an API function that will do this.) Then we convert that value into the red, green and blue components that we can use within Visual FoxPro. (Although, if all you are doing is setting a color property, then Visual FoxPro can use the Windows Color Number directly and the conversion is not necessary.) Here is the function:

```
**********************************************************************
* Program....: GetWCol.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Returns the Red, Green, and Blue color values for a
* ...........: given numbered Windows Object Color
**********************************************************************
LPARAMETERS tnObjectNumber
*** Check Parameter
IF VARTYPE( tnObjectNumber ) # "N" OR ! BETWEEN( tnObjectNumber, 0, 28)
    WAIT WINDOW "Must pass windows color number between 0 and 28" NOWAIT
    RETURN
```

```
ENDIF

*** Get the required color setting
DECLARE INTEGER GetSysColor IN Win32API ;
               INTEGER nObject
lnWinCol = GetSysColor(tnObjectNumber)

*** Convert to RGB Values
lnSq256 = 256 ^ 2
lnRedGrn = MOD( lnWinCol, lnSq256 )

*** Now get the individual components
lcBlue   = TRANSFORM( INT( lnWinCol/lnSq256 ) )
lcGreen  = TRANSFORM( INT( lnRedGrn/256) )
lcRed    = TRANSFORM( MOD( lnRedGrn,256) )

*** Return RGB string
RETURN (lcRed + "," + lcGreen + "," + lcBlue)
```

This function requires a numeric constant that identifies a Windows element. These are all defined in the "*Windows.h*" file included with the sample code for this chapter, but some of the key ones are listed here for convenience.

Table 13.5 *Constants for Windows element colors*

Constant	Windows Element Color
1	Background (Windows Desktop)
2	Title Bar (Active Window)
3	Title Bar (Inactive Window)
5	Window Background
9	Title Bar Caption Text (Active Window)
19	Title Bar Caption Text (Inactive Window)
13	Highlighted Item Background
14	Highlighted Item Text
17	Command Button
18	Command Button Text

How do I change the cursor? *(Example: ChgCursor.prg)*

You can easily customize your cursors using the Windows API. For example, you can replace the standard static hourglass with an animated hourglass that rotates, just by issuing this function call:

```
ChgCursor( FULLPATH( 'HourGlas.ani' ), 32514 )
```

Note that the *ChgCursor()* function takes two parameters. The first is file name that is to be used for the cursor – in the example above this is one of the Windows standard "animated" cursor files. The second parameter is a constant which defines which cursor type (i.e. I-beam, hand, hourglass) will be replaced with the new file. These constants can be found in Windows.h and a list of them is also included in ChgCursor.prg itself.

One good use for this function is to change the standard I-beam cursor to an arrow when you want to use a grid that is either read-only, or looks like a list box, using a single line of code (as opposed to the methodology we outlined earlier in this chapter):

```
ChgCursor( FULLPATH( 'Arrow_m.cur' ), 32513 )
```

Just be aware that if you do, ***all*** of your I-beam cursor will be replaced by the arrow. This includes any text boxes that may be on the form so make sure that the change only applies when the mouse is over the grid. Here is the program we use to change the cursor:

```
**********************************************************************
* Program....: ChgCursor.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Changes the specified cursor to the specified .cur or .ani file
**********************************************************************
LPARAMETERS tcCursorFile, tnCursorType
LOCAL lcNewCursor

ASSERT VARTYPE( tcCursorFile ) = 'C' ;
  MESSAGE 'Must Pass a File Name to ChgCursor.Prg'
ASSERT VARTYPE( tnCursorType ) = 'N' ;
  MESSAGE 'Must Pass a Numeric Cursor Type to ChgCursor.Prg'

IF INLIST( JUSTEXT( tcCursorFile ), 'CUR', 'ANI' )
  IF FILE( tcCursorFile )
    DECLARE INTEGER LoadCursorFromFile in Win32Api String
    DECLARE SetSystemCursor in Win32Api Integer, Integer

    lcNewCursor = LoadCursorFromFile( tcCursorFile )
    SetSystemCursor( lcNewCursor, tnCursorType )
  ENDIF
ENDIF
```

How do I customize my beeps? *(Example: MsgBeep.Prg)*

If you look at system sounds in the Windows control panel, you will notice that different sounds are defined for different types of events. It is a very simple matter to use these settings to play sounds that are consistent with other Windows application when you display a message box in Visual FoxPro. It is even more convenient because the **MESSAGEBOX** icon constants are identical to those used to identify the associated sounds in the Windows API. To play the sound associated with critical stop set up in the Windows control panel, all you need to do is this:

```
MsgBeep( 16 )
```

The program used to wrap this API function is very simple indeed:

```
**********************************************************************
* Program....: MsgBeep.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Play the specified system sound as a beep
*              Note that the beep constants correspond to the MESSAGEBOX Icon
constants
```

```
**********************************************************************
LPARAMETERS tnBeep

DECLARE INTEGER MessageBeep IN Win32API INTEGER
MessageBeep( tnBeep )
```

How do I find out if a specific application is running? *(Example: IsRunning.prg)*

As you read through the Window API Help, you may run into the `FindWindow` function and think that you can use this to find out if a particular application is running. Unfortunately, this function requires you to know the exact caption displayed in the application window's title bar. Sometimes this is impossible. For example, when a document is being edited in Microsoft Word, the Word window's caption contains the name of the document being edited. Because it is generally not possible for your Visual FoxPro application to know these details, we cannot use the *FindWindow()* function to determine whether Word is running. To solve this problem, our *IsRunning()* function makes use of the Windows API `GetWindowText` function.

IsRunning() is an overloaded function. If called with no parameters, it returns a parameter object that contains an array property. This array contains the names of all the running applications. If a partial string such as 'Microsoft Word' is passed, it returns a logical true if the application is running. If you wanted to expand upon its functionality, you could modify it to return a two-dimensional array and populate the second column of the array with the application's window handle:

```
**********************************************************************
* Program....: IsRunning.prg
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: When passed a string (i.e., 'Microsoft Word'), return .T.
* ...........: if the application is running. When invoked with no parameters,
*............: returns a parameter object whose array lists all running
*............: applications
*************************************************************************
FUNCTION IsRunning
LPARAMETERS luApplication

LOCAL luRetVal, lnFoxHwnd, lnWindow, lnWhich, lcText, ;
  lnLen, laApps[1], lnAppCnt

*** Declare necessary Windows API functions

DECLARE INTEGER GetActiveWindow IN Win32Api

DECLARE INTEGER GetWindow IN Win32Api ;
  INTEGER lnWindow, ;
  INTEGER lnWhich

DECLARE INTEGER GetWindowText IN Win32Api ;
  INTEGER lnWindow, ;
  STRING   @lcText, ;
  INTEGER lnLen

DECLARE INTEGER IsWindowVisible IN Win32Api ;
  INTEGER lnWindow
```

```
lnAppCnt = 0

*** Get the HWND (handle) to the main FoxPro window
lnFoxHwnd = GetActiveWindow()
IF lnFoxHwnd = 0
  MESSAGEBOX( 'Invalid return value from GetActiveWindow', 16, 'Fatal Error' )
  RETURN
ENDIF

***  Loop through all the running applications
lnWindow = GetWindow( lnFoxHwnd, 0 )
DO WHILE lnWindow # 0

  *** Make sure we do not have the Visual Foxpro window
  IF lnWindow # lnFoxHwnd
    IF GetWindow( lnWindow, 4 ) = 0 AND IsWindowVisible( lnWindow ) # 0
      lcText = SPACE( 254 )
      lnLen  = GetWindowText( lnWindow, @lcText, LEN( lcText ) )

      *** If the function was passed an Application Name, check for a match
      *** Otherwise, Add this to the array
      IF lnLen > 0
        IF VARTYPE( luApplication ) = 'L'
          lnAppCnt = lnAppCnt + 1
          DIMENSION laApps[ lnAppCnt ]
          laApps[ lnAppCnt ] = LEFT( lcText, lnLen )
        ELSE
          IF UPPER( ALLTRIM( luApplication ) ) $ UPPER( ALLTRIM( lcText ) )
            RETURN .T.
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDIF

  *** See if there is another running application
  lnWindow = GetWindow( lnWindow, 2 )
ENDDO

*** Either we haven't found a match for the passed application name
*** or we are returning an array of all running applications
IF VARTYPE( luApplication ) = 'L'
  SET CLASSLIB TO Ch13 ADDITIVE
  luRetVal = CREATEOBJECT( 'xParameters', @laApps )
ELSE
  luRetVal = .F.
ENDIF

RETURN luRetVal
```

To see *IsRunning()* in action, just type **`DO DemoIsRunning`** in the command window (after you have downloaded and unzipped the sample code, of course!) to view a cursor which lists all running applications.

Using the DECLARE command

You will have realized from the preceding sections that the key to accessing the Windows API is the Visual FoxPro **DECLARE** command. This command registers a function which is defined in a Windows *Dynamic Linked Library* (*.DLL* file) and makes it available to Visual FoxPro as if it actually were a native function. You can think of DLLs as the Windows equivalent of the familiar Visual FoxPro procedure files but, unlike a procedure file, the functions contained in a DLL must be registered individually before they can be accessed.

The basic syntax and usage of DECLARE is explained reasonably clearly in the online Help files, and even more clearly in the "*Hacker's Guide to Visual FoxPro 6.0,*" but there are a few points that are worth emphasizing when working with API functions:

- The function name is actually case sensitive! This is most unusual in Visual FoxPro and is therefore worth mentioning here. If you receive an error that states "Cannot find entry point…." then almost certainly you have the case for the function name wrong..
- Not only is the function name case sensitive but in some cases the actual function name may not be the same as that which is stated. (This is because there may be different functions for different character sets. Thus the API "*MessageBox*" function is actually two functions – "*MessageBoxA*" which works with single-byte character sets and "*MessageBoxW*" for unicode character sets. However, you do not normally need to worry about this since, if Visual FoxPro cannot find the function specified, it will append an "A" and try again.)
- There is no such file as '*WIN32API.DLL*" despite the fact that you will often see functions being declared in this library. It is actually a 'shortcut' that instructs Visual FoxPro to search a pre-defined list of files including: *Kernel32.dll*, *Gdi32.dll*, *User32.dll*, *Mpr.dll*, and *Advapi32.dll*.
- When declaring a function, you can specify a local alias for that function by including the 'AS' keyword in the declaration. This can be useful because while the actual function name is case sensitive, the local alias (being known only to Visual FoxPro) is not.

How do I check what API functions are loaded?

The native **DISPLAY STATUS** report includes, at the very end, a list of all declared DLL functions together with the actual file in which the function is located as illustrated:

```
Declared DLLs:
GetActiveWindow     C:\WINDOWS\SYSTEM\USER32.DLL
GetSystemDirectory  C:\WINDOWS\SYSTEM\KERNEL32.DLL
GetWindow           C:\WINDOWS\SYSTEM\USER32.DLL
GetWindowText       C:\WINDOWS\SYSTEM\USER32.DLL
IsWindowVisible     C:\WINDOWS\SYSTEM\USER32.DLL
```

How do I release an API function?

Unfortunately, there is no way to release a single API function once it has been declared. Issuing either a CLEAR DLLS or a CLEAR ALL will release all declared functions but in this case at least, it really is "all or nothing!"