This paper was originally presented at the Southwest Fox conference in Mesa, Arizona in October, 2009. http://www.swfox.net

Quibbles, Quirks, and Quickies

Rick Borup
Information Technology Associates
701 Devonshire Drive, Suite 127
Champaign, IL 61820
Email: rborup@ita-software.com

Things don't always work quite the way we expect them to. The quirky behavior of a function or control can be a common cause of those "What the Heck??" moments we all encounter from time to time. We may be tempted to quibble with VFP's behavior in these situations, at least until we learn how it works and why it behaves the way it does. This session explores some of the quirks and quibbles in Visual FoxPro with the goal of reducing the number of "What the Heck" moments in your development work, and demonstrates ways you can boost your productivity by using some quick and easy techniques in VFP that may often be overlooked or forgotten.

Introduction

This session is based on my own experience and research while developing Visual FoxPro applications over the last 15 or 20 years. The examples and explanations presented here are observational: in other words, they come from observing the behavior of Visual FoxPro and not from any inside knowledge of how VFP works internally. If you are an experienced Visual FoxPro developer, you may have run into some of these things yourself. While not everything in this session will necessarily be new to everyone, my hope is that at least some of it will be new to each of you and useful to all of you.

Execute Code

If you work with Microsoft SQL Server, you know that you can run code in the Query Analyzer by selecting a block of code and pressing F5. I find this to be very useful when writing and debugging SQL scripts as well as for running ad hoc queries.

Did you know that Visual FoxPro provides a similar facility, which is available in both the Command Window and the Code Window? Select a block of code (one or more consecutive lines), right-click on the selection, and choose Execute Selection from the shortcut menu.

There are just a couple of things to be aware of, particularly if you're using this feature to execute a portion of program code from the Code Window. One is that locals are in scope only if they're included in the block of code being executed. The code in Listing 1 runs as expected if you select and execute all 7 lines at once. But if you select and execute lines 1-3, where the local variables are defined and given an initial value, and then select and execute line 4 separately, you get the error "Variable 'LCX' is not found". This is because variable lcx went out scope as soon as the first Execute Selection was finished.

Listing 1: Select a block of consecutive lines, right-click, and choose Execute Selection from the shortcut menu.

```
1 LOCAL lni, lcx
2 lni = 1
3 lcx = "Foo"
4 ?"lcx = " + lcx
5 FOR lni = 1 TO 5
6   ? SPACE(lni - 1) + "Fox rocks!"
7 ENDFOR
```

The same error occurs if the code being executed reference a defined constant and the #DEFINE statement is not included in the code block being executed.

Lines 5-7 in Listing 1 can be executed by themselves, however. Why? Because the `FOR VarName = nInitialValue TO nFinalValue` statement does not require VarName to exist before the statement is executed.

I find this feature to be quite useful when testing and debugging small blocks of code during development. It's also very useful for running demo code during a presentation, and I use it extensively for that purpose in this session.

GetWordCount() and GetWordNum()

If your application needs to import data from a third party, you may find yourself needing to work with delimited text files. This format is often used as kind of a lowest common denominator data format for transferring data between two otherwise incompatible systems.

Visual FoxPro's GetWordCount() and GetWordNum() functions offer a convenient way to work with fields in a delimited text file. If you use VFP's low level file functions to read in each line of the text file as a single string, you can then use GetWordCount() to find out how many "words"—in other word, how many data fields—there are in that string and then use GetWordNum() to parse out each individual field.

The default delimiters for GetWordCount() and GetWordNum() are space, tab, carriage return, and line feed, but Visual FoxPro enables you to define other delimiters if the need arises. In my experience, a comma is frequently used to separate fields in a delimited data file. In one case I even had to work with a spec where the sender used the tilde character (~) as the field delimiter. VFP to the rescue: it was easy to re-program my app to use either of those characters as the delimiter without changing the code used to parse the string.

Listing 2 illustrates the use of GetWordCount() and GetWordNum() to parse out fields from a sample comma-delimited string. For the sake of simplicity I'm creating the string manually instead of reading it in from a flat file, but the result is the same. In the example, field 1 is an ID, field 2 is the last name, field 3 is the first name, and field 4 is the balance.

Listing 2: Use GetWordCount() and GetWordNum() to parse fields from a comma-delimited string.

```
lcDelimiter = ","
lcString = "ALFKI,Anders,Maria,567.89"
lnWordCount = GETWORDCOUNT( lcString, lcDelimiter)
?"Word count = " + TRANSFORM( lnWordCount)
?"ID         = " + GETWORDNUM( lcString, 1, lcDelimiter)
?"LAST NAME  = " + GETWORDNUM( lcString, 2, lcDelimiter)
?"FIRST NAME = " + GETWORDNUM( lcString, 3, lcDelimiter)
?"BALANCE    = " + GETWORDNUM( lcString, 4, lcDelimiter)
```

In this example everything comes out as expected:

```
Word count = 4
ID         = ALFKI
LAST NAME  = Anders
FIRST NAME = Maria
BALANCE    = 567.89
```

This approach is quite easy to implement, but it turns out problems arise when there are empty fields resulting in consecutive delimiters in the string. Let's suppose that for some record the first name is unknown. The sender of the file might very well format that line in their file as follows:

```
ALFKI,Anders,,567.89
```

Unfortunately, GetWordCount() and GetWordNum() don't handle the consecutive delimiters the way you would expect them to. Instead of recognizing that there are still four fields in the string, with the third one being blank, the code in Listing 2 (using the new value for lcString) returns the following result:

```
Word count = 3
ID         = ALFKI
LAST NAME  = Anders
FIRST NAME = 567.89
BALANCE    =
```

Obviously this presents some major problems. If you are using GetWordCount() to validate the expected number of fields in the record, validation fails because it returns 3 instead of 4. If you proceed with GetWordNum() to parse out the fields, the first name becomes 567.89 and the balance is empty! Clearly, this is unworkable.

The solution is to insert a space between the consecutive delimiters wherever there is an empty field. With a space present between the delimiters, GetWordCount() returns the correct value and GetWordNum() parses them out correctly.

Visual FoxPro's excellent and extensive string handling functions make it easy to do that. One way is to use a STRTRAN( ) function to replace ",," (two consecutive commas) with ", ," (comma space comma) wherever they occur in the string. Keep in mind that there might be more than two consecutive delimiters, though, which would happen if there are two or more consecutive field with an empty values, so a single STRTRAN() may not be enough. You can implement a general solution by looping though the string to detect and replace consecutive delimiters until no more exist, as shown in Listing 3.

Listing 3: Use STRTRAN( ) in a FOR loop to insert a space between all consecutive delimiters.

```
lcDelimiter = ","
lcString = "ALFKI,,,567.89"   && Note two consecutive empty fields.
DO WHILE lcDelimiter + lcDelimiter $ lcString
   lcString = STRTRAN( lcString, ;
                       lcDelimiter + lcDelimiter, ;
                       lcDelimiter + SPACE(1) + lcDelimiter)
ENDDO
lnWordCount = GETWORDCOUNT( lcString, lcDelimiter)  && Result is 4
?"Word count = " + TRANSFORM( lnWordCount)
?"ID         = " + GETWORDNUM( lcString, 1, lcDelimiter)
?"LAST NAME  = " + GETWORDNUM( lcString, 2, lcDelimiter)
?"FIRST NAME = " + GETWORDNUM( lcString, 3, lcDelimiter)
?"BALANCE    = " + GETWORDNUM( lcString, 4, lcDelimiter)
```

This gives the expected result: GetWordCount( ) says there are 4 fields and GetWordNum( ) parses fields 2 and 3 as empty while extracting the balance as field 4 where it belongs.


```
Word count = 4
ID         = ALFKI
LAST NAME  =
FIRST NAME =
BALANCE    = 567.89
```

Importing Dates

In a flat file, all fields are essentially character strings. The simplest way to import this kind of data is to create a cursor with a character data type for each field and use APPEND FROM <the file> to pull the raw data into VFP. Then you can do data type transformation as needed on any fields containing numeric values, dates, and so on.

In some cases, though, you can save a step by importing directly into a non-character data type field, such as numeric or date. This can work for numeric data if the incoming string is all numerals without a currency symbol or separators (i.e., without the dollar sign and commas, for American currency). It can also work with dates, but there are a couple of quirks due to the different ways dates can be stored as strings.

In a flat file, dates are frequently stored with a separator between the month, day, and year. Common formats in the United States are mm/dd/yyyy and mm-dd-yyyy. Sometimes dates are stored without separators, frequently as yyyymmdd. Of course there are other possibilities, but in my experience these are the three most common.

If you append a date string into a column of data type date in a VFP cursor or table, results vary depending on the format of the date string and the type of file from which it's being imported. The two types of files I frequently work with are SDF, which has a fixed record length and fixed field widths, and Delimited, in which the fields are separated by commas or some other delimiter.

Assume an incoming flat file with two columns where the first is a date and the other is a character string. A row from an SDF file might look like this:

```
09/08/2009Foo – or -
09-08-2009Foo – or -
20090809Foo
```

An equivalent row from a comma-delimited file could look like this:

```
09/08/2009,Foo – or -
09-08-2009,Foo – or -
20090809,Foo
```

If we create a VFP table as follows:

```
CREATE TABLE myTable ( dDate1 D, cField1 C(10))
```

we can then use APPEND FROM to read the flat file into the table. The results are shown in Table 1.

Table 1: Dates do not always get imported correctly.

Date Format	File Type	dDate1	cFieldl
09/08/2009	SDF	09/08/0020	09Foo
09-08-2009	SDF	09/08/1920	09Foo
20090809	SDF	09/08/2009	Foo
09/08/2009	Delimited	09/08/2009	Foo
09-08-2009	Delimited	09/08/2009	Foo
20090809	Delimited	09/08/2009	Foo

Notice that VFP handles all three date formats correctly when importing from a delimited file, but only the yyyymmdd date format came across correctly when importing from an SDF file. The solution, of course, is to import the date into a C(10) or C(8) field, whichever is appropriate, and then convert the string to a date as a second step.

Using Transform() with precision

The Transform( ) statement formats an expression as a string. It uses a format code to determine how the resulting string is formatted. This makes Transform( ) very powerful and quite a time saver in situations where you might otherwise have to write a chunk of code to accomplish the same thing. However, Transform( ) can be a little tricky and may not always give the results you at first expect.

Like many other developers, I commonly use Transform( ) to format numeric values for presentation on a report, and in some cases for printing a currency value on a check. Different customers have different preferences for how numeric and currency values should appear, and sometimes there are constraints on the size of the resulting string due to fields being very close together on a report or a pre-printed check having a fixed-width box in which the amount must fit. Knowing how to use Transform( ) effectively can help you work within these constraints.

The syntax of the Transform( ) statement is

```
TRANSFORM( eExpression, [cFormatCodes])
```

where eExpression is the expression to be formatted and the optional cFormatCodes are used to determine the format of the resulting string. In the simplest case you can transform a numeric value into a string with using any format codes just by writing

```
TRANSFORM( 123456.78)
```

which returns the string 123456.78 with a length of 9 characters. Similarly,


```
TRANSFORM( 1234567.89)
```

returns the string 1234567.89 with a length of 10 characters.

Most of the time, when working with numbers over 1000, you'll want to format them with a 1000's separator and possibly a currency symbol. For this you can use a format code, which is passed to the Transform( ) statement as the second parameter. Note that the format code is a string, so remember to enclose it in double quotes, single quotes, or square brackets. The statement

```
TRANSFORM( 123456.78, "999,999.99")
```

returns the string 123,456.78. In this example, the second parameter "999,999.99" is the format code. It has a length of 10 characters, and therefore so does the resulting string.

If a number is negative, the minus sign occupies one of the positions in the return string. The format code "999,999.99" is therefore sufficient for values in the range of -99,999.99 to +999,999.99. If you pass a number smaller than -99,999.99, such as -100,000.00, Transform() returns a string of asterisks.

```
TRANSFORM( -100000.00, "999,999.99")   && Result is ***,***.**
```

In a real application you'll almost certainly be passing a column or variable name instead of a literal value and you'll want the format code to be able to accommodate the largest anticipated numeric value. For example, in an application that deals with numbers below one billion, you might use a format code of "999,999,999.99". When smaller numeric values are passed to Transform( ) using this format code, the return string has leading spaces. The statement

```
TRANSFORM( 123456.78, "999,999,999.99")
```

returns the 14-character string ___123,456.89 where the underscore characters represent the leading spaces.

The Transform() statement allows for the use of several format codes beginning with the @ symbol. These format codes determine in part how the result string is formatted. For example, the @$ format code adds the currency symbol to the resulting string.

However, if you use only the @$ format code the result does not have any 1000's separators.

```
TRANSFORM( 123456.78, "@$")   && Result is $123456.78 with three leading spaces
```

For more control over the result, the @[x] format codes can be used in conjunction with another string that supplies additional formatting information. In this case the second

string is referred to as the format mask. In the above example, you can restore the 1000's separator by adding a format mask of "9999,999.99" after the @$ format code, like this:

```
TRANSFORM( 123456.78, "@$ 9999,999.99") && Result is $123,456.78
```

Note that the space between the format mask and the format code is required. Otherwise, the format mask is ignored.

Those of you with a sharp eye probably noticed the extra "9" in the format mask compared to the earlier example of "999,999.99". The reason this is required is that the currency symbol occupies one position in the result string. If you use a format mask of "999,999.99", values up to 99,999.99 will include the currency symbol but values of 100,000.00 and up will not.

```
TRANSFORM( 99999.99, "@$ 999,999.99") && Result is $99,999.99

TRANSFORM( 100000.00, "@$ 999,999.99") && Result is 100,000.00 (no currency symbol)
```

The @R format code tells the Transform() statement to read and apply the format mask that follows. Again, a space is required between the format code and the format mask. This offers another way to format numeric values, as in the following examples.

```
TRANSFORM( 123456.78, "@R $999,999,999.99") && Result is $    123,456.78, length 15
with four embedded spaces

TRANSFORM( 123456.78, "@R $$$$,$$$,$$9.99") && Result is     $123,456.78, length 15
with four leading spaces
```

A lot of the data I work with includes social security numbers (SSN), and I frequently use Transform( ) when I need to present an SSN in the common format of xxx-xx-xxxx. If the SSN is stored as a numeric value, it can be formatted using a format code of 999-99-9999.

```
TRANSFORM( 123456789, "999-99-9999") && Result is 123-45-6789
```

However, storing an SSN as a numeric value is bad database design. The rule of thumb for any stored value, the way I was always taught, is that if you are not going to add to or subtract from it—that is, if you're never going to use it in an arithmetic expression—then it should be stored as a string. Therefore, the correct way to store an SSN is as a 9-character string. (Don't even get me started on why you should never store the formatting characters in the database—and yes, I've seen it done!)

Using Transform() on an SSN stored as a string presents a minor challenge, because you cannot use the simple "999-99-9999" format code. If you do, the result is not what you want.

```
TRANSFORM( "123456789", "999-99-9999") && Result is 123-56-89
```

Note that the hyphens have replaced the '4' and the '7' instead of being inserted into the string as expected.

The solution is to use the @R format code in conjunction with the desired format mask. The following gives the expected result.

```
TRANSFORM( "123456789", "@R 999-99-9999") && Result is 123-45-6789
```

Numeric values in fields

When you store a numeric value in a memory variable, that variable can accommodate any value within the range allowed by Visual FoxPro.[1] The same is not true when you store a numeric value in a numeric field in a table, because a field in a table has a defined width that limits its maximum value.[2]

The question is, given a numeric field of some defined size, what is the maximum value it can hold? The answer may seem obvious but could surprise you.

As an example, create a table with a numeric field of size N(10,2). Remembering that the decimal position occupies one position, it seems obvious the maximum value that can be stored in that field is 9,999,999.99. Or is it?

```
CREATE TABLE myTable ( nField1 N(10,2))&& Max value 9,999,999.99 (?)
```

Certainly the field can accommodate a value of 9,999,999.99 (nine million, nine hundred ninety nine thousand, nine hundred ninety nine dollars and ninety nine cents), so that's not in doubt. This suggests that any number of ten million or greater would not fit in the field. But what actually happens if you attempt to store a value greater of ten million or greater? Does it cause numeric overflow, or a program error, or something else?

The answer is that it depends on the number of significant digits in the value, and to understand what does happen you need to know that Visual FoxPro may truncate, round, or convert values to scientific notation in order to force the value to fit in the field. This is not hidden behavior; it is documented in the REPLACE Command topic in the Visual FoxPro Help file.

As a first example, try to replace the value in nField1 with a number greater than ten million, such as 12,345,678.90.[3] Note that while this number is indeed greater than ten

---

[1] The range for numeric data is - .9999999999E+19 to .9999999999E+20.

[2] It would be more correct to refer to this as a column instead of a field, but the VFP Help file uses the term field and so will I.

[3] Long values consisting of all 9's are a bit difficult to read, so for some of these examples I use a value with different numerals such as 1234567.89. Again, what's important is the number of significant digits, so using values like these doesn't materially affect the results, it just makes them easier to read.

million, it contains only nine significant digits because the trailing zero is not significant for our purposes.

In order to see the results of each REPLACE statement, BROWSE the table to see how Visual FoxPro displays the value. If you're skeptical, you can see the actual byte values stored in the field by opening the table in a hex editor, such as the one that comes with Visual FoxPro.[4]

```
REPLACE nField1 WITH 12345678.90    && 12,345,678.90
BROWSE && Field contains 12345678.9
```

When viewed in the hex editor you can see this is exactly what's stored in the table, including one byte for the decimal point.

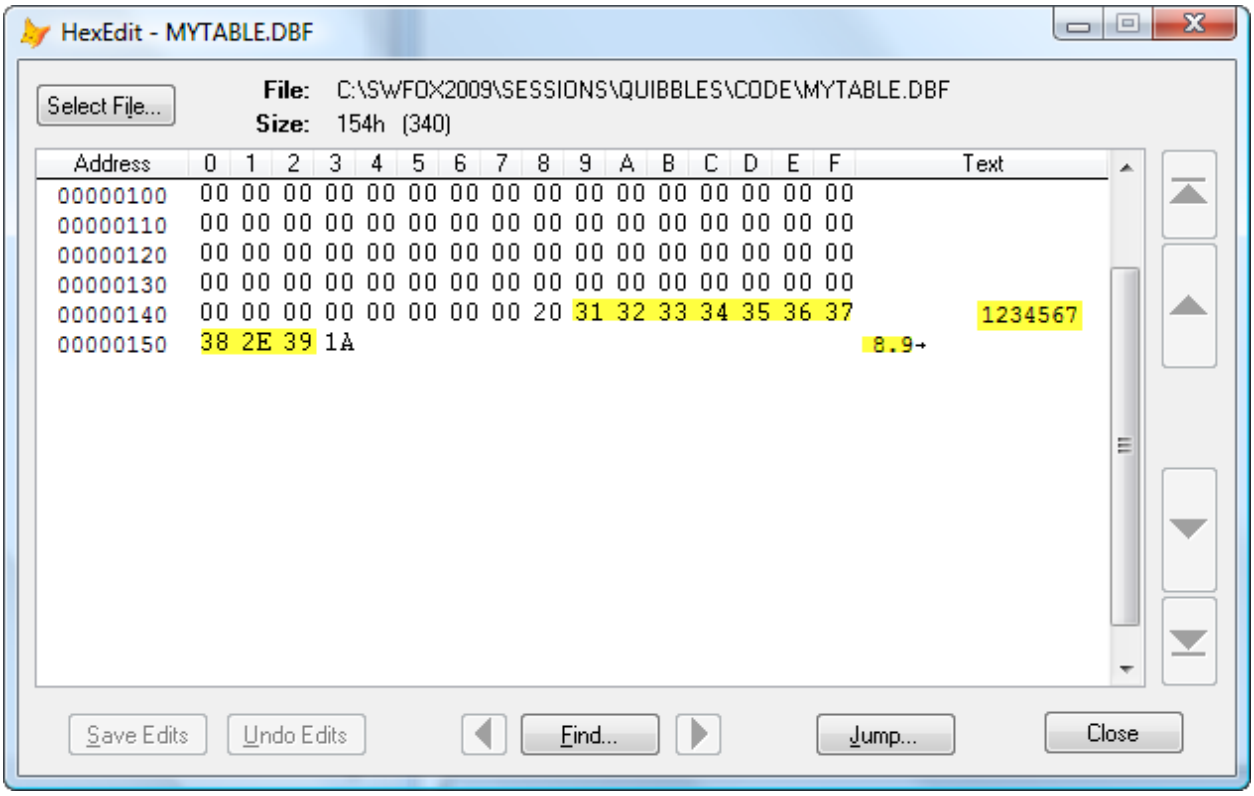


Figure 1: The hex editor shows what's actually stored in the table.

Rather than tediously going through each of the rest of the examples one by one in this paper, the results are summarized in the following table. Note what happens as the number of significant digits reaches and finally exceeds the maximum value the field can hold.

---

[4] HexEdit.app can be found in the Visual FoxPro Tools\Hexedit folder, and can be launched with `DO HOME(1) + "Tools\Hexedit\Hexedit.app"` in the Command Window.

Table 2: A numeric field of size N(10,2) can hold values much larger than you might think.

Value to store	Result (value actually stored)	Comment
12,345,678.90	12345678.9	Trailing zero dropped
12,345,678.99	12345679.0	Rounded
123,456,789	123456789	No decimal positions
123,456,789.99	123456790	Rounded
999,999,999	999999999	Much bigger than ten million
999,999,999.99	1000000000	Rounded to ten billion
9,999,999,999.99	9999900000 (sic)	Numeric overflow, data was lost
10,000,000,000	1.000E+10	Converted to scientific notation

The point of these examples is to illustrate that although the maximum value a numeric field can hold is determined in part by the size of the field, Visual FoxPro can truncate, round, or convert the value to scientific notation in order to force a much larger value to fit.

Behavior of null values

Null values are a fact of life when working with fields in a database. They can occur explicitly when stored as the actual value in a field, frequently as the default value, or they can occur in a cursor as the result of a query. Either way, null values may require special handling in your code, so it's important to understand how Visual FoxPro treats null values in various types of expressions.

Null values in logical expressions

The rule of thumb is that a null value represents an "I don't know" condition, and therefore the result of any operation involving a null is also "I don't know". When it comes to logical expressions, however, that's not always true. The following table shows the result of compound logical expressions involving a null value.

Table 3: The behavior of null values in compound logical expressions is documented in the VFP Help file.

Compound logical expression	Result
.T. AND NULL	NULL
.F. AND NULL	False
.T. OR NULL	True
.F. OR NULL	NULL

While the behavior of null values in logical expressions is well documented in the VFP Help file, it may be counter-intuitive because the result is not always NULL even if one of the components in the expression is NULL.

Null values in character expressions

Consider a database table that stores people's names. Such a table may have individual fields for the person's first name, middle name or middle initial, and last name. Because the middle name or initial is frequently omitted or not known, it's common to use a default value of NULL for the middle name field in the table.

```
CREATE TABLE myTable ;
   (  cFName C(10) NULL, ;
      cMName C(10) NULL, ;
      cLName C(20) NULL, ;
      nBalance N(10,2) NULL ;
   )
INSERT INTO myTable VALUES ( "Homer", "J", "Simpson", 100.00)
```

Now consider an expression to format a name for printing. A simple example, omitting the traditional period after the middle initial, might look this:

```
ALLTRIM( cFName) + " " + ALLTRIM( cMName) + " " + ALLTRIM( cLName)
```

If there is non-null data in all three fields, the formatted string is "Homer J Simpson" as expected. But what happens if Homer didn't supply his middle initial on his job application? Even though the first name and last name are still present, the middle name is now null and as a result, entire formatted name string is null! Not the result you want to see printed on a report or a paycheck.

By the way, this has nothing to do with using ALLTRIM(). The expression

```
cFName + " " + cMName + " " + cLName
```

also returns null if cMName is null.

The solution of course is to use the NVL() function to substitute the empty string for a null value. The expression

```
ALLTRIM( cFName) + " " + NVL( ALLTRIM( cMName), "") + " " + ALLTRIM( cLName)
```

returns "Homer Simpson", a much more acceptable result. A more sophisticated example could take out the extraneous space, too.

Null values in numeric expressions

In the sample table above, the balance field contains a value of 100.00. To add a $10.00 late charge, a program use code like that shown below, storing the result in a local memory variable named lnNewBalance.

```
lnNewBalance = nBalance + 10.00 && If nBalance is 100.00,
?lnNewBalance                   && the result is 110.00.
```

This yields the expected result of 110.00. But if the balance field is null, the numeric expression does not evaluate to 10.00 as you might expect. Instead, it evaluates to null.

```
lnNewBalance = nBalance + 10.00 && If nBalance is null
?lnNewBalance                   && the result is .NULL.
```

You can use NVL() to substitute a value of zero for a null balance, in which case the expression would evaluate to 10.00.

FoxPro is case insensitive… or is it?

I admit up front that this is somewhat of a misleading question. It's certainly true that Visual FoxPro is a case insensitive language with respect to field and variable names. For example, you can define a variable as lcVar and refer to it later as lcvar with no problems. This is either a good or a bad thing, depending on your point of view, but it is second nature for most Visual FoxPro developers not to care about case.

However, case can definitely make a difference in string comparisons. As I demonstrate in the session presentation, there are times when a string comparison like "FOO" = "foo" is true and times when it is false.[5] If you habitually use UPPER() or LOWER() to handle the variable side(s) of a string comparison you won't run into trouble, but if you're accustomed to working in an environment where "FOO" = "foo" is true but ends up being false at runtime, this can bite you big time.

The ASCAN() function offers another example where case might not make a difference even though it seems that it should. ASCAN() scans an array and returns the number of matches found, or zero if a match is not found.

You can pass a bit flag parameter to ASCAN(), whose value is used to refine the search. Turning on the one-bit in this flag is makes the search case insensitive. This implies that if the one-bit is off or the parameter is omitted then the search should be case sensitive. However, it is possible to construct a situation where ASCAN() is case insensitive even if the one-bit in the flag is off or the parameter is omitted.[6]

One place where this made a difference in my own work was using ASCAN() to search an array created by the ADBOJECTS() function. I used ADBOJECTS() to create an array of tables in a database, then used ASCAN() to find out if a particular table existed. The values in the array returned by ADBOJECTS() are all upper case, as shown in the code below. Therefore an ASCAN() that looks for a match on a lower case or mixed case table name should return false. In my situation, however, due to the way my development environment

[5] Because this paper is being distributed to registered attendees in advance of the conference, I won't reveal the "secret" at this time. You'll have to attend the session to find out!

[6] See footnote 5.

was configured, it returned true. This served me well during design and development but caused me a lot of grief when it began returning false at runtime.[7]

Here's the code, modified just a bit for illustration.

Listing X: The values in the array created by ADBOBJECTS() are upper case, so a case sensitive ASCAN should fail.

```
=ADBOBJECTS( laTables, "TABLE")  && Create array of table names.
IF TYPE( "laTables[1]") = "C"
   IF ASCAN( laTables, "tblFoo") > 0 && If tblFoo exists,
      DROP TABLE tblFoo              && drop it.
   ENDIF
ENDIF
CREATE TABLE tblFoo ;
   (  nIdentity I     AUTOINC, ;
      cField1      C(10) ;
   )
```

If ASCAN() is case sensitive it should return a non-zero value because "TBLFOO" != "tblFoo". This indicates the table already exists in the database, in which case the table is dropped before the new one of the same name is created. However, if ASCAN() is case insensitive it should return a zero value indicating the table does not exist. In this case the table is not dropped and the CREATE TABLE command, which is executed regardless, fails because a table of the same name does in fact already exist in the database.

_TALLY

As explained in the Visual FoxPro Help file, the system variable _TALLY contains the number of records processed by the most recent table command. The value of _TALLY is set after running VFP table commands such as SUM, REPLACE, and DELETE, but it is also set to the number of rows returned from a SQL SELECT statement. This is quite useful because often you may want to take one action if no rows were returned from a SQL SELECT, but take another action if one or more rows were returned.

To illustrate, a sample table is created with two columns and ten rows, as shown below. The value in the first column is the same in all rows, so a query for any other value in the first column will return no rows. The second column has a numeric value we can use in an aggregate function such as SUM.

```
CREATE TABLE myTable ( cID c(10), nValue I)
FOR lni = 1 TO 10
   INSERT INTO myTable ( cID, nValue) VALUES ( "foo", lni)
ENDFOR
```

When used without any aggregate functions in the SQL SELECT statement, the value of _TALLY is the same in VFP 9.0 as in earlier versions.

[7] See footnote 6.

```
SELECT * FROM myTable WHERE cID = "foo"
?_tally && Value is 10
SELECT * FROM myTable WHERE cID = "bar"
?_tally && Value is 0
```

A database engine change in Visual FoxPro 9.0 affects the cursor returned from a SQL SELECT statement that uses an aggregate function, such as SUM, when the query returns no rows.

In previous versions of VFP, the resulting cursor contains no rows and therefore _TALLY is zero. So in earlier versions of VFP it was common to write code like this:

```
* With SET ENGINEBEHAVIOR 70 or 80
SELECT SUM( nValue) ;
   FROM myTable ;
   WHERE cID = "bar" ;
   INTO CURSOR csrResult
IF _tally = 0     && True, so
   DO Procedure_A && Procedure_A is performed.
ELSE
   DO Procedure_B
ENDIF
```

However, in VFP 9.0 the resulting cursor contains one row with a value of NULL for the sum. Therefore, the value of _TALLY is 1 and Procedure_B is performed instead.

```
* With SET ENGINEBEHAVIOR 90
SELECT SUM( nValue) ;
   FROM myTable ;
   WHERE cID = "bar" ;
   INTO CURSOR csrResult
IF _tally = 0     && False, so
   DO Procedure_A
ELSE
   DO Procedure_B && Procedure_B is performed instead!
ENDIF
```

By simply changing to ENGINEBEHVIOR 90, the same code gives a different result, creating an instant problem for your application.

Because of this and other changes that “broke” existing code in applications written before VFP 9.0, many developers simply stuck with the older engine behavior. I’ve done this myself in applications where there was no incentive (financial or otherwise) to migrate to the newer engine behavior in VFP 9.0.

If you do want to update your application to use the VFP 9.0 engine behavior, the way to make the above code work like it did before is to add a GROUP BY clause to group on the cID column.

```
SELECT SUM( nValue) ;
   FROM myTable ;
```

```
   WHERE cID = "bar" ;
   INTO CURSOR csrResult ;
   GROUP BY cID
IF _tally = 0     && True, so
   DO Procedure_A && Procedure_A is performed, as before.
ELSE
   DO Procedure_B
ENDIF
```

If you migrate an older application to ENGINEBEHAVIOR 90, you will likely need to make other changes to the code as well. Read the What's New in Visual FoxPro topic in the Help file and pay particular attention to the changes affecting the behavior of SQL statements.

What's in a name?

The concept of a "work area" has been a fundamental when working with tables ever since the earliest days of FoxPro. Every table you open in Visual FoxPro is opened in its own work area.

Work areas can be referenced by a number from 1 to n, but are more commonly referenced by an alias. An alias is simply a name used to refer to a work area. Alias names must begin with a letter or an underscore. By convention, alias names are generally the same as the name of the table itself, but this is not required.

The main reason for using an alias is so your code can refer to a work area by a meaningful name instead of by a meaningless number. Alias names should always be made meaningful regardless of whether or not they are the same as the table that's open in the work area they refer to.

The USE statement provides for an optional alias name to be assigned to the work area when opening a table. It's common practice to explicitly use the table name as the alias when opening a table, or to omit the alias and let Visual FoxPro assign one. Unless the table name does not begin with a letter or underscore, and as long as it does not conflict with another alias already in use, the alias assigned by Visual FoxPro is the same as the physical table name.

Sometimes it's difficult to tell whether you're referencing a table name or an alias. The SQL INSERT statement provides a good example of this. Here is the syntax of the SQL INSERT statement, taken directly from the Visual FoxPro Help file.

```
INSERT INTO dbf_name [(FieldName1 [, FieldName2, ...])]
   VALUES (eExpression1 [, eExpression2, ...])
```

Consider the following code:

```
CREATE TABLE table1 ( cField1 C(10))
INSERT INTO table1 ( cField1) VALUES ( "Bob")
```

In the INSERT statement in this code, the question is: Does `table1` refer to a physical table name or to an alias? The answer is: It depends. What it depends on is whether or not there is an open work area with an alias of 'table1' at the time the INSERT statement is executed.

Remembering that the alias does not have to be the same as the table name, we can construct an example that's intentionally misleading but serves to illustrate the point. The following code creates two tables and opens each one with what would ordinarily be the other one's alias.

```
CREATE TABLE table1 ( cField1 C(10))
CREATE TABLE table2 ( cField1 C(10))
CLOSE TABLES ALL
USE table1 IN 0 ALIAS table2
USE table2 IN 0 ALIAS table1
```

It's clear that the work area whose alias is 'table2' refers to table1.dbf, while the work area whose alias is 'table1' refers to table2.dbf. What isn't so clear is what happens when you write the following INSERT statements:

```
INSERT INTO table1 ( cField1) VALUES ( "Bob")
INSERT INTO table2 ( cField1) VALUES ( "Carol")
```

Do `table1` and `table2` refer to aliases or table names? The syntax says they are "dbf_names", but in which table do Bob and Carol actually end up?

In this example, Bob ends up in table2.dbf and Carol ends up in table1.dbf. The way to understand this is to interpret the first statement as meaning "insert the value 'Bob' into the table that is open in the work area whose alias is 'table1'", and conversely for the second statement.

To force VFP to interpret "dbf_name" as a physical table name instead of an alias, include the .dbf file name extension.

```
INSERT INTO table1.dbf ( cField1) VALUES ( "Ted")
INSERT INTO table2.dbf ( cField1) VALUES ( "Alice")
```

In this way, you can guarantee that Ted ends up in table1 and Alice ends up in table2.dbf regardless of any alias names in use at the time. The net of all this is that Carol and Ted wind up together in table1.dbf while Bob and Alice end up together in table2.dbf, which is kind of what happened in the movie.

Breakpoints list in the debugger

The Breakpoints window enables you to view and manage the breakpoints you want to use to help debug your application. Among other information, the Breakpoints window includes a list of the breakpoints currently in effect. You can click on a breakpoint in the list to select it and then remove, enable, or disable it.

When the number of breakpoints is greater than the number of items that can be displayed in the list at one time, the list can be scrolled down to see the rest of the items. Figure 2 shows that several breakpoints have been set in main.prg, with the breakpoint at line 38 being the last one visible without scrolling the list. There are also breakpoints at lines 39-44, but you have to scroll down to see them.

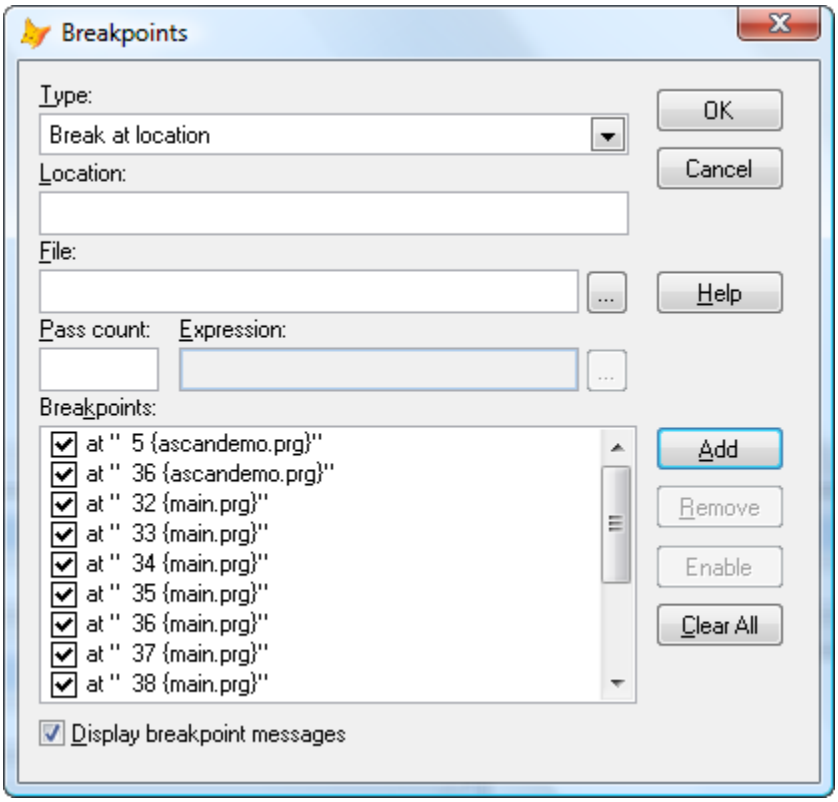


Figure 2: The Breakpoints window contains a scrollable list of the breakpoints currently in effect.

Suppose you want to select the breakpoint at line 44, maybe to disable or remove it. The natural thing to do is to scroll down until that breakpoint is visible, the click on it to select it. However, there is a quirk in the behavior of the list that can cause the wrong item to be selected.

When the Breakpoints window opens, the focus is not originally on the list. Without clicking on the list, grab the scrollbar with the mouse and drag it down until the breakpoint at line 44 is visible. Then click on that breakpoint to select it. Figure 3 shows how the Breakpoints list looks immediately prior to clicking on the line 44 breakpoint. Notice the position of the scrollbar, which is at its bottommost position.

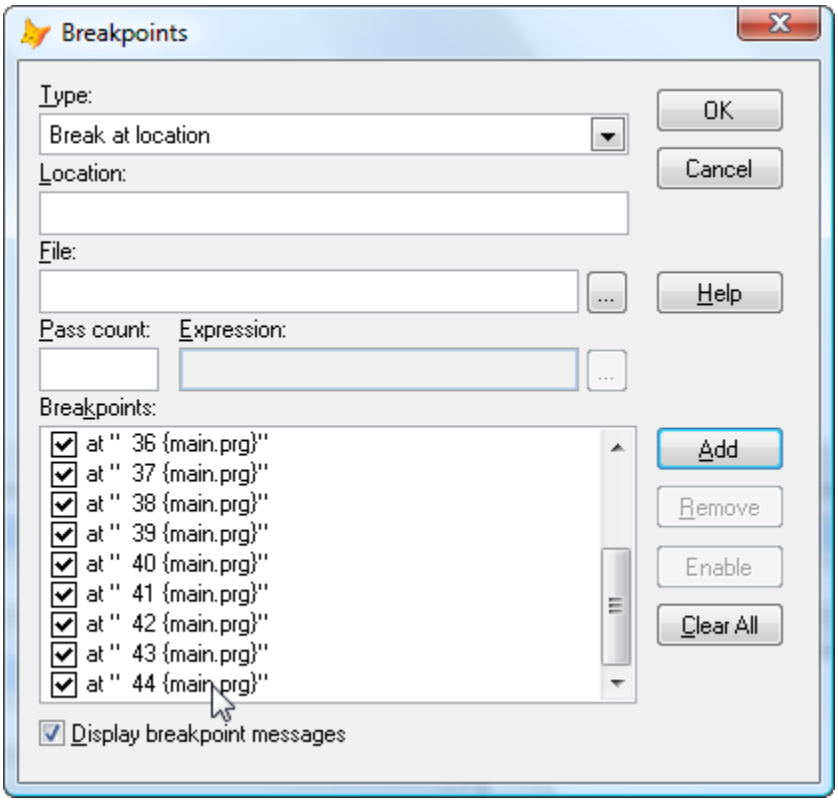


Figure 3: With the focus still on one of the other controls, scroll down and click on the breakpoint at line 44 to select it.

What happens is that instead of selecting the breakpoint you clicked on , Visual FoxPro selects the breakpoint that was originally at the bottom of the list before you scrolled down. In other words, instead of selecting the breakpoint at line 44, which is the one you clicked on, the breakpoint at line 38 is selected.

This is illustrated in in Figure 4. Note that the breakpoint in line 38 has been selected, and the scrollbar has reverted to its original topmost position.

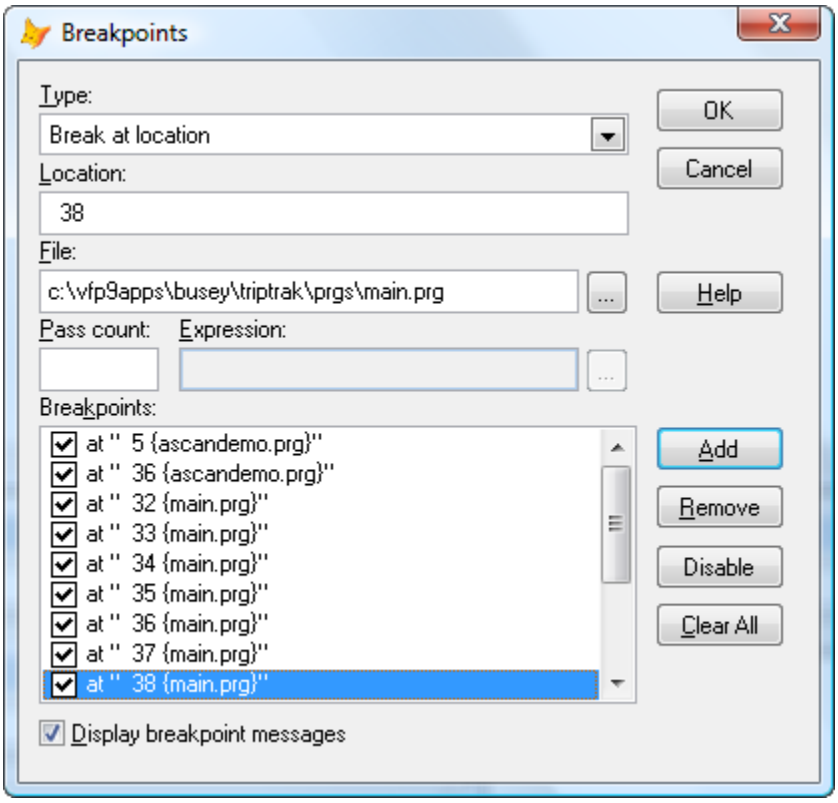


Figure 4: The breakpoint at line 38, which was originally the last item visible in the list, is selected even though you clicked on the breakpoint at line 44 further down the list.

This is a relatively minor quirk but it can cause quite a bit of confusion and lost time if you don't notice that the wrong breakpoint has been selected and then proceed to enable, disable, or remove it in preparation for running your code through its tests again.

The work-around is simply to click in the list itself before scrolling down, or to scroll down again after the incorrect item has been selected and then re-select the desired breakpoint further down the list. Incidentally, the behavior is the same regardless of whether you open the Breakpoints window from Tools | Breakpoints on the Debugger menu or from Tools | Breakpoints on the VFP main menu.

Code references tool

The Code References tool searches the specified file types in a project or folder and locates all references to a specified string. A project-level search examines all files of the specified types[8] that are part of the project and lists the references found.

In order to use the Code References tool to search all the files within an entire project, the project must be open in the project manager. The quirk is that if you rename a project file

[8] The default file types are *.scx, *.vcx, *.prg,*.frx, *.lbx, *.mnx, *.dbc, *.qpr, and *.h.

from within the project manager (right-click | Rename), the renamed file is not included in future searches until you close and reopen the Project Manager.

For example, a Code References search for the variable name `llNameUsed` in the project I'm using as an example turns up four references, all of them in a program file named makename.prg. If I change the name of that program file to createname.prg and search again without closing and reopening the Project Manager, no references to `llNameUsed` are found.

Forms and shortcut menus

This one is more of a "how to" than a quickie, but it does illustrate a quick and easy way to construct a call to a method on a form from a shortcut menu invoked from that form.

The objective is to be able to right-click on a form, or on a control on a form, choose an item from the shortcut menu, and have the menu code execute a method on the form without tightly coupling the menu to the form.

One approach would be to have the code in the menu use either the object name of the form, assuming it has one, or _screen.ActiveForm to identify the object whose method is to be called. For example, if you `DO FORM myForm NAME oMyForm`, then a menu could call `oMyForm.DoSomething()` to execute the DoSomething() method on the calling form.

This solution is sub-optimal because it depends on coding the runtime object name of the form into the menu code at design time. Using _screen.ActiveForm is even worse because there's no good way to control what form might actually be the active form at runtime.

The solution I offer here is simply to pass an object reference to the form, and optionally to an object on the form, to the menu at runtime. Code in the menu can then be written to refer to the object and control name references it received, with no dependency on their actual names at runtime.

As an example, consider a simple form with a button as shown in Figure 5. Right-clicking on the form or the button brings up a shortcut menu from which an action can be chosen. Choosing the Do Something item on the menu calls the DoSomething() method on the form, which simply displays a message box saying that the method was called. The Change Color item on the menu changes the background color of the object that was clicked, which culd be either the form or the command button.

Do implement this, the RightClick() method on the form passes an object reference to the form to the menu.

```
DO myPopupMenu.mpr WITH thisform, null
```

The RightClick() method code on the command button is the same except it passes an object reference to itself as well as to the containing form.

```
DO myPopupMenu.mpr WITH thisform, this
```

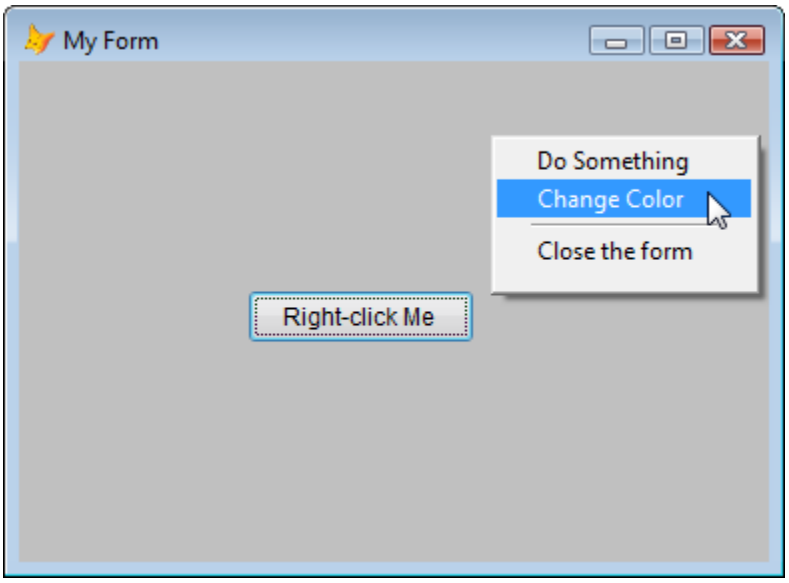


Figure 5: A simple form with one control. Right-clicking on the form or on the command button brings up a shortcut menu with choices that call back to methods on the form.

To key to making this work is to use the Setup code in the shortcut menu to accept two parameters, one for the form and an optional second one for the control.

```
PARAMETERS oCaller, oControl
* Use PARAMETERS instead of LPARAMETERS so the parameters
* can be seen by the rest of the menu code.
```

Once the menu has an object reference to the calling form and control, the rest is simple. The code in Do Something procedure in the menu is simply

```
oCaller.DoSomething()
```

The code in the Change Color menu procedure is only slightly more involved:

```
IF ISNULL( oControl)   && We're working with the form itself.
   oCaller.BackColor = GETCOLOR()
ELSE                   && We're working with a control on the form.
   oControl.BackColor = GETCOLOR()
ENDIF
```

A general solution to the design objective is thus accomplished by providing the menu with an object reference instead of relying on a specific object name.

Setting tab order

The tab order of controls on a form determines the order in which the controls receive focus when pressing the tab key or otherwise moving through the controls in sequence.

Visual FoxPro provides a very nice visual method for assigning tab order to controls on a form. With a form open in the Form Designer, clicking the Set Tab Order button on the Form Designer toolbar applies a numbered overlay to each control. You can then renumber the controls from 1 to n by clicking on the numbered overlays in the desired sequence.

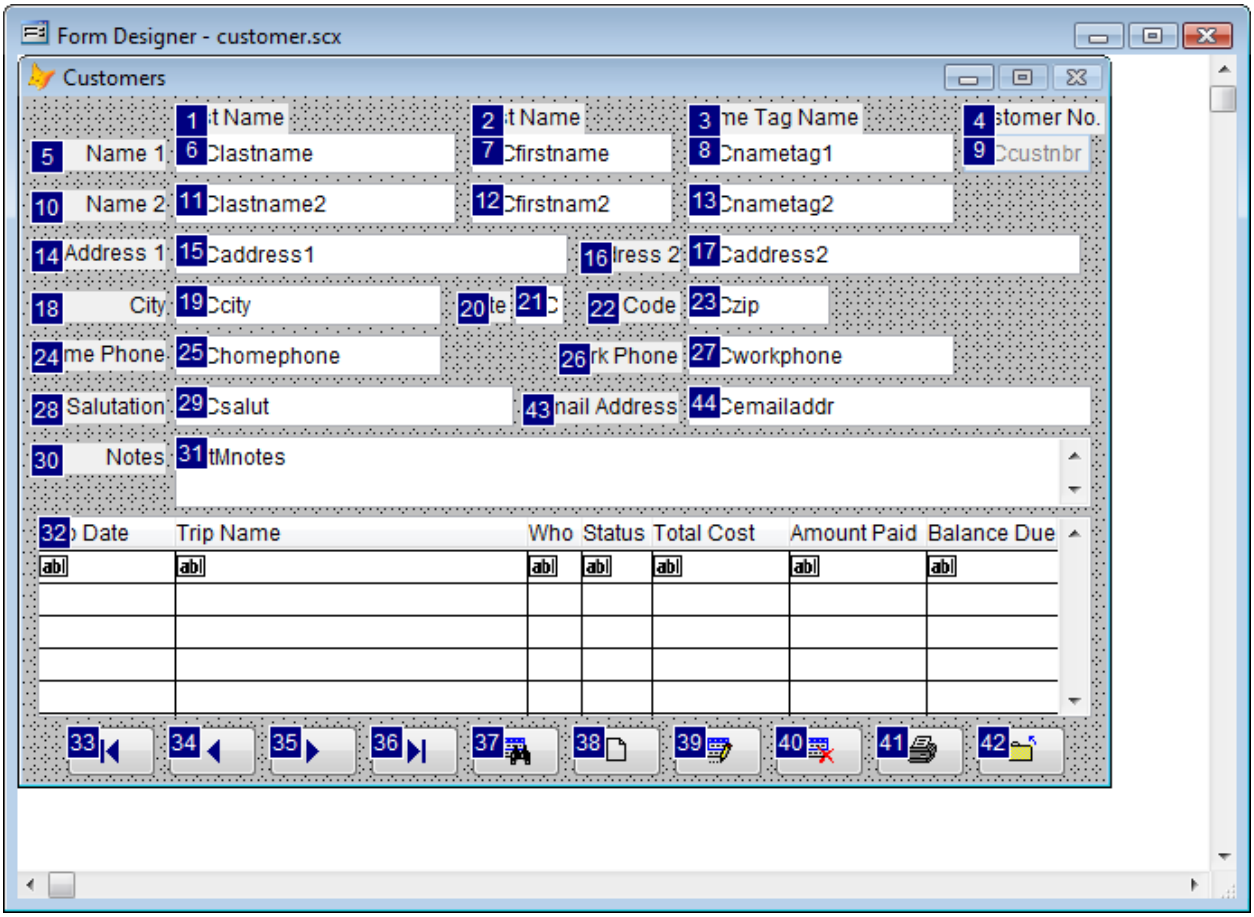


Figure 6: The tab order of controls on a form can be set visually by clicking on the numbered overlays for each control in the desired sequence.

While this is easy to do, it can be tedious when there are a lot of controls on the form. Also, since you have to start with 1 and proceed up from there, the visual method may not be best when all you have to do is to reorder a couple of controls near the end of the sequence.

In Figure 6, notice that the controls for the email address are out of sequence. For proper sequencing when doing data entry with the keyboard, the textbox for the email address should immediately follow the one for the salutation. (The tab order of the labels can be ignored since labels never receive the focus.)

To change the tab order visually, you would need to start with the name field, which is already number 1, and click through each of the other controls in turn, ending with the rightmost button in the lower right of the form. In a situation like this it may be easier to use the list method of setting tab order, a feature than may be often overlooked.

With the form open in the Form Designer, and the visual tab order overlays turned off, select View | Tab Order | Assign by List from the main menu. This brings up an ordered list of the controls on the form. The tab order of a control can be changed simply by clicking and dragging the name of the control up or down in the list.

To set the tab order of the email address textbox to be immediately after the salutation, locate the txtcEmailAddr control at the bottom of the list and drag it up until it is directly underneath the txtcSalut control. The tab order of all the other controls beneath it is automatically changed to reflect the new sequence.

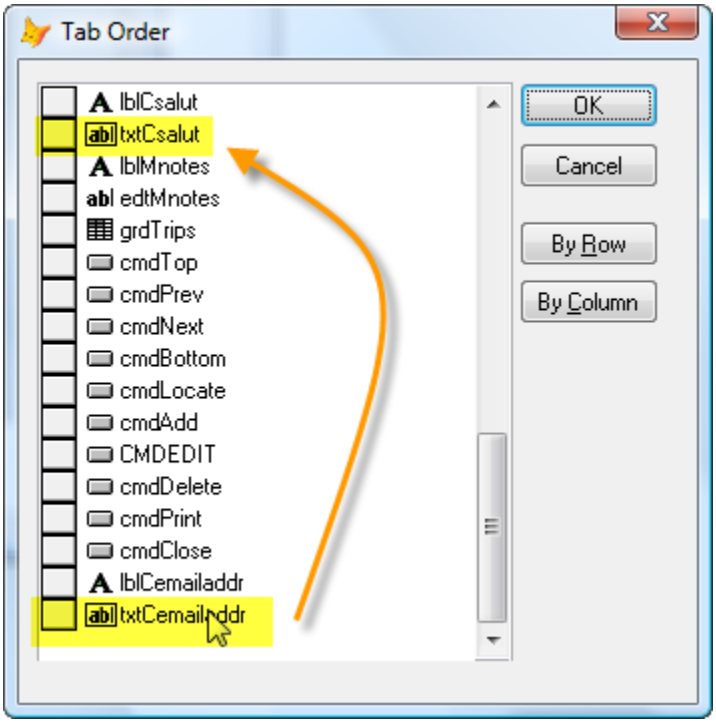


Figure 7: You can change the tab order of controls by rearranging the order of the items in the Tab Order list.

As with so many things in Visual FoxPro, there's more than one way to rearrange the tab order of controls on a form. It's nice to have a choice and be able to use whichever one works best for the task at hand.

ICASE

The ICASE (Immediate CASE) function was introduced in VFP 9.0. It offers developers a convenient and highly readable way to write code that returns a result from a list of conditions, which might otherwise typically be coded as a series of nested IIFs.

Consider the following code, which returns the phonetic name for the first four letters of the alphabet.

```
lcType = IIF( lcInput = "A", "Alpha", ;
              IIF( lcInput = "B", "Bravo", ;
                   IIF( lcInput = "C", "Charlie", ;
                        IIF( lcInput = "D", "Delta", "Delta"))))
```

The example has been truncated for the sake of illustration, but even with only four conditions it's easy to see how cumbersome this approach would become with a much longer list of conditions.

The new ICASE statement makes this type of code much easier to write.

```
lcType = ICASE( lcInput = "A", "Alpha", ;
                lcInput = "B", "Bravo", ;
                lcInput = "C", "Charlie", ;
                lcInput = "D", "Delta", ;
                "Other")
```

Again, imagine the list of conditions extended to the entire alphabet and think about how much easier it is to write, read, and debug this code than it would be with a list of 26 nested IIFs. Notice too that an "otherwise" condition is easily added to the end of the list, and you only need one right parenthesis to end the statement.

ICASE is a great addition to the language and deserves a place in your set of frequently used functions.

References and definitions

Visual FoxPro offers some convenient ways to look up references and definitions for things in your code. As with many conveniences, these are located on the shortcut menu available while working in the Code Window.

Look up a reference

To easily look up a reference to something in the code you are editing, select the string you're interested in, right-click on the selection and choose Look Up Reference from the shortcut menu.

For example, in the following segment of code let's assume we're interested in knowing where else "formlist" is referenced.

```
IF NOT USED('formlist')
   USE FORMLIST.DBF IN 0 ALIAS FORMLIST
ENDIF
```

Select the string FORMLIST (on either line), right-click on the selection and choose Look Up Reference from the shortcut menu. This brings up the Look Up Reference dialog with the search term already entered for you.

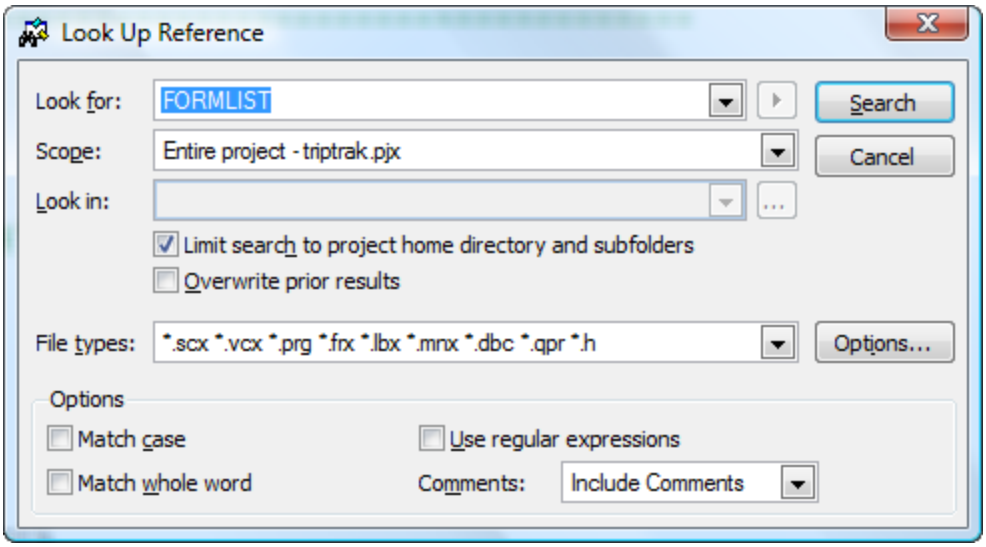


Figure 8: Use the Look Up Reference item on the Code Window shortcut menu to open the Look Up Reference dialog with your search term already entered for you.

Clicking the Search button then opens and runs the Code References tool and displays the results of the search, just as if you had begun by opening the Code References tool and initiating the search that way.

Find a definition

Along the same lines as looking up references, there are times when you may want to quickly find out where something such as a defined constant or a method name is defined. Visual FoxPro gives you an easy way to do this using View Definitions item on the Code Window shortcut menu.

In the code segment below, note that the word “TRUE” is used as the return value.

```
DEFINE BAR lnNewBar OF Window PROMPT "&lcBarName."
ON SELECTION BAR lnNewBar OF Window &lcObjName..Show()
RETURN TRUE
```

It’s fairly obvious from the syntax that TRUE is probably a defined constant with a logical value of .T., but in other cases it might not be at all obvious, particularly if this is not your own code. To find out where TRUE is defined, select that string in the Code Window, right-click on the selection and choose View Definitions from the shortcut menu. VFP searches

the project for the definition and displays the result. In this case the only place TRUE is defined is in a header file named ITA.h, so when the search is finished Visual FoxPro opens ITA.h in the Code Window and highlights the search term for you.

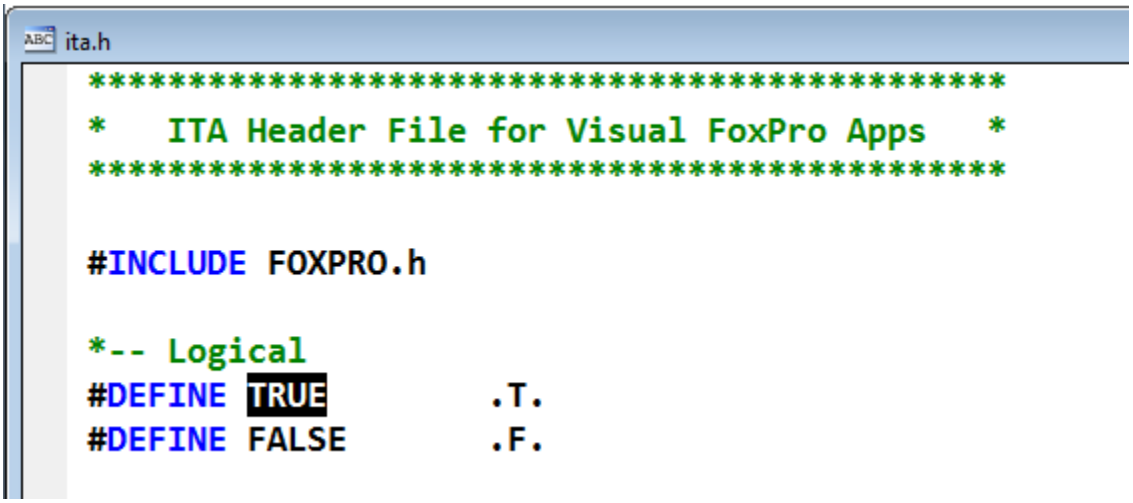


Figure 9: View Definitions found where TRUE is defined in the file ITA.h.

You can use the View Definition feature to search for method, function, and procedure names, too. If the name you are searching for is defined in only one place in the project, the file containing the definition is automatically opened in the Code Window. If the name you are searching for is defined in two or more places, the View Definition feature opens a Go To Definition window. This window lists the files in which a definition was found. This is illustrated in Figure 10, where two definitions were found for the method name CreateMenuPad.

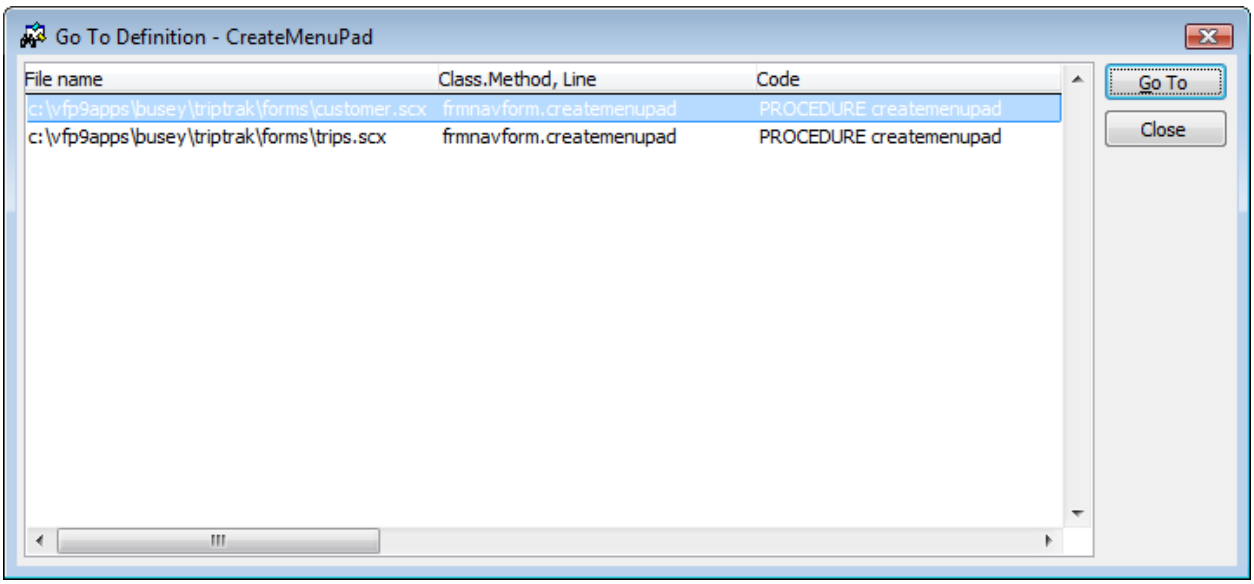


Figure 10: If two or more definitions are found, the View Definition feature opens a window listing the files.

As an additional convenience, you can select a file name from the list and click the Go To button. VFP opens the selected file in the Code Window and displays the location where the definition was found.

Inserting a reference to an object

When writing code you frequently need to insert a reference to an object. However, you may not instantly remember how you spelled the name of the desired object—was it txtFName or txtFirstName, for example? Moreover, if the object you want to reference is deep in a containership hierarchy it might be tedious to figure out exactly what that hierarchy is, much less to remember the names of all the higher level objects.

Consider a form with a grid. A text box in a column in the grid is at least four levels deep in the form's containership hierarchy, possibly even more if for example the grid is on a page within a pageframe. Manually typing a fully qualified reference to that text box would be tedious and potentially error-prone.

The Insert Object Reference tool, which is available via the Object List item on the Code Window shortcut menu, shows you a list of all the objects in the current designer and makes it easy to insert a fully qualified reference to an object into your code. This tool is available when working in the Form Designer or the Class Designer.

For example, with the Code Window open to a method on a form, choose Object List from the shortcut menu and VFP presents you with a treeview-style list of all the objects in that form. Figure 11 shows the list of objects in a form named Customer.

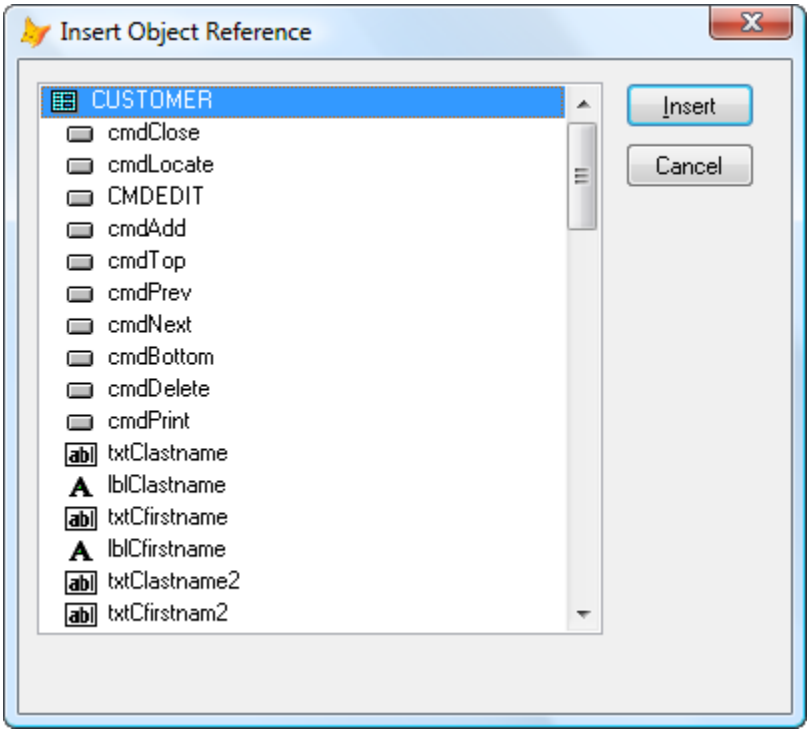


Figure 11: The Object List tool shows a hierarchical list of all the objects in a form.

To insert a fully qualified reference to any object, simply select the desired object in the list and click the Insert button. For example, to insert a reference to the txtCustNbr text box in the grid in the Customer form, select the txtCustNbr object name in the list and click Insert, as shown in Figure 12.

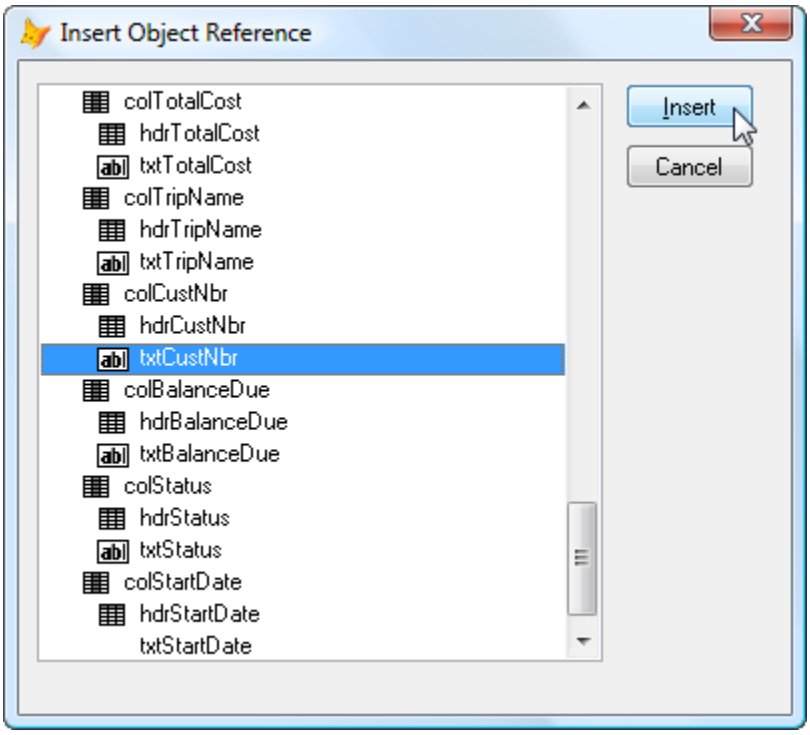


Figure 12: Select an object name in the list and click the Insert button to insert a fully qualified reference to that object into your code.

Visual FoxPro inserts a fully qualified reference to the txtCustNbr textbox object at the insertion point in the method currently open in the Code Window. The result looks like this:

```
ThisForm.grdTrips.colCustNbr.txtCustNbr
```

The Insert Object Reference tool thus offers a quick way to view the names of all the objects in a form or class, and makes entering a reference to any of those objects much easier and more reliable than typing it in by hand.

TEXT… ENDTEXT

Constructing a SQL statement in code for use with SQLEXEC can be a laborious task, and the challenge of getting it right increases with the length of the SQL statement. Debugging

the code that creates a long SQL statement can be difficult because the code itself may not look a great deal like the SQL statement that results from running that code.

One common way of constructing a SQL statement is to break it into manageable segments and write code to continuously append each segment to an aggregate string. Given the task of constructing a SQL INSERT statement for SQLEXEC to run against a Microsoft SQL Server database, and given that the values to be inserted are all strings stored in local memory variables in a VFP program, a simple example might look like this:

```
lcSQL = [INSERT INTO myTable ]
lcSQL = lcSQL + [( cAcctNbr, cSSN, cLastName, cFirstName, cGender, mNotes )]
lcSQL = lcSQL + [ VALUES ]
lcSQL = lcSQL + [( '] + lcAcctNbr + [', '] + lcSSN + [', '] + lcLastName
lcSQL = lcSQL + [', '] + lcFirstName + [', '] + lcGender
lcSQL = lcSQL + [', '] + lcNotes + [')]
?lcSQL
```

Assuming values as shown below for the local memory variables, the resulting string is what you would expect:

```
INSERT INTO myTable ( cAcctNbr, cSSN, cLastName, cFirstName, cGender, mNotes )
 VALUES  ( '12345',  '123456789',  'Doe',  'Jane',  'F',  'This is a note.' )
```

While this works fine, the code that creates the SQL INSERT statement does not look a great deal like the resulting statement itself. This can make it more difficult to spot syntax or coding errors, and the difficulty grows as the SQL statement you are building becomes longer and more complex.

Visual FoxPro's TEXT... ENDTEXT command can be used to great effect in constructing SQL statements as a string and storing it in a memory variable for use with SQLEXEC. By using TEXTMERGE to insert the values into the string, and by using a PRETEXT value to strip out unwanted characters, you can write code like this:

```
TEXT TO lcSQL TEXTMERGE NOSHOW PRETEXT 15
   INSERT INTO myTable
   ( cAcctNbr,
     cSSN,
     cLastName,
     cFirstName,
     cGender,
     mNotes
   )
   VALUES
   ( <<['] + lcAcctNbr + ']>>,
     <<['] + lcSSN + ']>>,
     <<['] + lcLastName + ']>>,
     <<['] + lcFirstName + ']>>,
     <<['] + lcGender + ']>>,
     <<['] + lcNotes + ']>>
   )
ENDTEXT
```

The resulting value of lcSQL is the same as the string created above, but notice how much more readable this code is than the first way. In my experience, using this approach has made it much easier to catch coding and syntax errors in long SQL statements at design time, rather than having to find out there is a problem during testing or (worse) at runtime in a live environment.

The value of the PRETEXT flag is the key to making this work. PRETEXT is a one-byte binary flag whose bit position values tell the TEXT... ENDTEXT command to eliminate certain characters from the string. A value of 15 (all bits on, or 0xF) tells the TEXT... ENDTEXT command to eliminate spaces, tabs, carriage returns, and line feeds from the resulting string. This means that what's between TEXT and ENDTEXT is treated as all one line regardless of how you format it in your code. This is what allows you to write the statement on multiple lines the same way you might write the it in actual code, making it much easier to read and understand.

There are two things to watch out for when using this approach. One is to remember NOT to put a semi-colon at the end of each line with the TEXT and ENDTEXT boundaries, which you might be tempted to do out of habit since it looks so much like actual VFP code.

The other thing is to know that by taking out carriage returns and line feeds you may affect the contents of a memo field if the memo itself contains those characters. For example, if the mNotes field is a memo field and lcMemo is a multi-line string with line breaks, the embedded carriage return and line feed characters are passed to the TEXT... ENDTEXT command in lcMemo but then eliminated due to the PRETEXT setting. The result is one long line, which may be a problem if the format of the memo is important.

Even with those two caveats, using TEXT... ENDTEXT and PRETEXT to construct long SQL statements in code is a useful technique due to the greater readability of the code.

Report format expressions

Have you ever built a rather complicated report with lots of numeric fields, using a format expression for each one, only to find out that at runtime one or more of the numbers don't print as intended? Maybe a currency symbol is missing where one is desired, or the format expression is too small and the values are displayed as all asterisks, or some other problem along those same lines. In this situation you probably realize that not only should you fix the format in question, but you'd better check all the rest of the format expressions to be sure there aren't other problems waiting to happen.

As good as the VFP Report Designer is, it's still tedious work to sight verify the properties of several fields because you have to select, open, and view the properties sheet for each field individually. The task becomes a repetitive sequence of right-click, select Properties, click on the Format tab, inspect the format expression, click Cancel if no change is needed or modify the expression and click OK if a change is needed. Repeat, and repeat again.

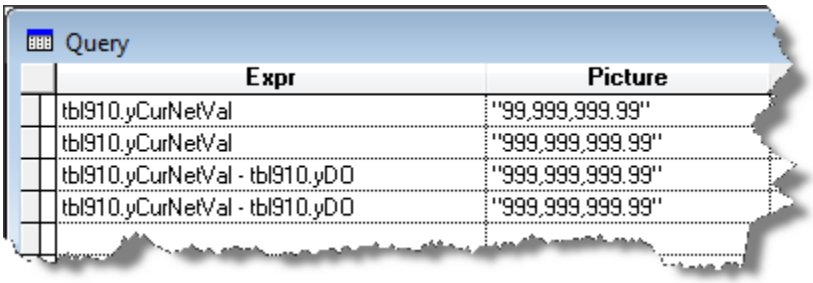
A simple piece of code can make it much easier to quickly inspect the format expressions for every field on a report. Because a Visual FoxPro .frx report file is actually a Visual FoxPro table file, you can write a query against it.

The following query selects the report expression field and its format for all rows in a Visual FoxPro report file.

```
SELECT PADR( ALLTRIM( expr), 30) as expr, ;
       PADR( ALLTRIM( picture), 30) as picture ;
       FROM theReport.frx
```

This creates a cursor you can visually scan to look for mistakes or formats that may need changing. It's particularly useful when all of the fields are supposed to be the same format, say 999,999,999.99 (nine positions to the left of the decimal), but a couple of them may have mistakenly been formatted with only eight positions to the left of the decimal. This kind of thing would be time consuming to find by examining the properties sheet of each field on the report individually, but the exception sticks out like a sore thumb in the query result.

Figure 13 shows a portion of the result of this query. Notice how easy it is to see that the format expression in the first row is different than the one for the other rows.


Query

	Expr	Picture
	tbl910.yCurNetVal	"99,999,999.99"
	tbl910.yCurNetVal	"999,999,999.99"
	tbl910.yCurNetVal - tbl910.yDO	"999,999,999.99"
	tbl910.yCurNetVal - tbl910.yDO	"999,999,999.99"

Figure 13: Using a query to create a cursor of report expressions and their formats makes it easy to compare and see which ones, if any, are different.

Naturally, you cannot make any changes or corrections to the format expressions directly in the query, but the value displayed in the expr column enables you to find the corresponding field in the Report Designer so you can make the change there.

VFP Options

The last "quickie" I want to show you is how to display and/or capture all of your Visual FoxPro option settings. These are the options you set in the Tools | Options dialog off the Visual FoxPro main menu. Although you can review and edit these setting in the Options dialog in the Visual FoxPro IDE, you might want to be able to print them off for review or documentation purposes.

The first method, which is built into Visual FoxPro, enables you to export all of the VFP Options to the output window in the debugger. To do this, open the VFP debugger, then go back into the VFP IDE, select Tools | Options from the main menu, and hold down the Shift key while clicking the OK button on the Options dialog. VFP writes all the options to the debugger output window, from which you can save them as a file.

The second method accomplishes essentially the same thing but enables you to do it programmatically. The VFP options are stored in the registry, so we can write code to read the registry and pull out the keys and values we want. The registry class from the VFP Foundation Classes makes it easy to enumerate keys and values in the registry.

The following code shows how to do this. It enumerates the values store in the HKCU\Software\Microsoft\VisualFoxPro\9.0\Options key, writes them to a file named EnumVFPOptions.txt and opens that file for viewing in the VFP Code Window.

```
#DEFINE HKEY_CURRENT_USER  -2147483647  && BITSET(0,31)+1
DIMENSION laValues[1]
LOCAL loRegistry, lcKey, lnFileHandle
loRegistry = NEWOBJECT( "registry", HOME(1) + "ffc\registry.vcx")
lcKey = "Software\Microsoft\VisualFoxPro\9.0\Options"
loRegistry.OpenKey( lcKey, HKEY_CURRENT_USER, .F.)
loRegistry.EnumKeyValues( @laValues)
lnFileHandle = FCREATE( "EnumVFPOptions.txt")
IF lnFileHandle = -1
   WAIT WINDOW "Unable to create file" TIMEOUT 3
ELSE
   FOR lni = 1 TO ALEN( laValues, 1)
      FPUTS( lnFileHandle, laValues[lni,1] + " = " + laValues[lni,2])
   ENDFOR
   FCLOSE( lnFileHandle)
ENDIF
MODIFY FILE "EnumVFPOptions.txt"
```

A third thing you might want to do is to export the VFP options from the registry into a .reg file. A .reg file, of course, can be imported back into the registry, so one situation where this can be particularly useful is when you're migrating to a new machine and want to reproduce all your VFP options on the new machine without having to set them up again manually via the IDE.

To save your VFP options in a .reg file, open the registry editor and navigate to the HKCU\Software\Microsoft\VisualFoxPro\9.0\Options key. Right click on the key, choose Export, and enter a file name to save the key and all its values, as illustrated in Figure 14.

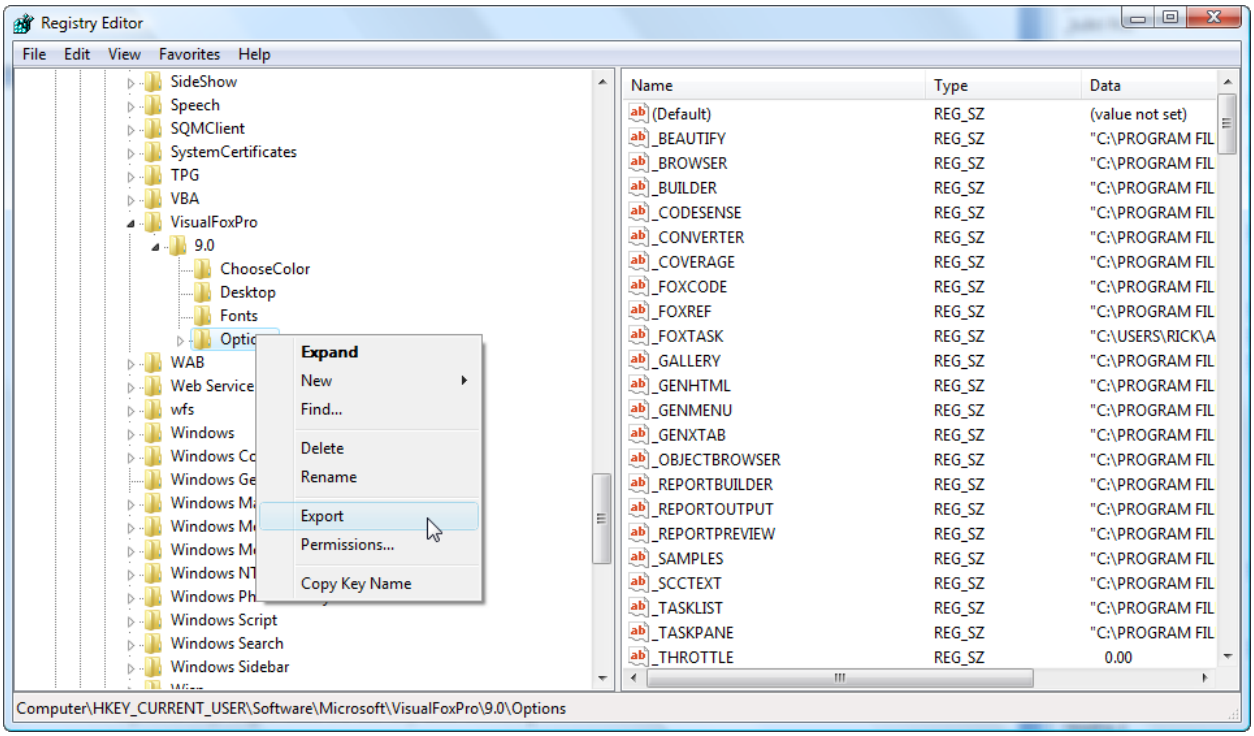


Figure 14: The VFP options are stored in the registry, from which they can be exported to a file using the registry editor.

As always, the standard caveats apply whenever you are working with the registry: be careful, make a backup first if you're unsure what you're doing, don't change or delete anything you don't understand, etc.

Summary

The intent of this paper has been to cover some of the quirks in Visual FoxPro that may cause frustration until you know how things work. The items presented in this paper come from my own experiences, so the list is by no means exhaustive and you may very likely have discovered other quirks on your own. If so, I encourage you to share them with the rest of the VFP community via user groups, your blog, twitter, or by posting on one of the Visual FoxPro forums. We all learn from each other, which is part of what makes the VFP community so special.

Visual FoxPro is a rich development environment. There is often more than one way to do something, and in many cases VFP provides quick and easy ways to perform common tasks. I hope you find the "quickies" presented in this paper useful in your own development work.