## Column: Visual Basic for Dataheads
## Figures: NONE
## File for Companion Disk: None

A Command and Function Summary – Part II

Whil Hentzen

In part one of this series, you got a glimpse of the language through a summary of functions. This month, I'm going to cover the logic structures and commonly used commands in Visual Basic, and compares them to their counterparts in VFP.

I know I claimed in my first article that there were only 19 keywords in all of Visual Basic, but, well frankly, I was lying. There are actually over 70. And while that number pales to the 500+ commands and functions in VFP, it's still too many to simply list in an alphabetical manner and expect you to grok them in a single sitting. How to organize them?

I asked my family (a bunch of VB candidates if I ever saw them) for suggestions, and they came up with a number of interesting suggestions. One family member thought perhaps by length of word, another suggested listing them the order that they were introduced to VB, and a third suggestion was summing up the ASCII codes that make up each word, and using the eventual values as a sorting order. The fourth suggestion was "Winnie the Pooh" but the offerer of the suggestion is only three years old. Each of these suggestions was excellent (I have to say that else I'll end up sleeping on the couch until VB43 is released), but it struck me that you use commands to do things when programming, and thus it makes sense to divide up the command into functional groups. So that's what I'll do here.

Logic Structures

You can do the same things with Visual Basic as you can with VFP in terms of controlling flow within a program module. The syntax is just a little different.

Making a decision

The statement for making a decision is, as it is with VFP, IF. However, VB's IF is much more flexible, as shown in this code fragment:

```
IF <condition1> THEN
<stuff to do>
ELSEIF <condition2> THEN
<stuff to do>
ELSEIF <condition3> THEN
<stuff to do>
ELSE
<stuff to do>
END IF
```

You'll notice that a "THEN" keyword is required after the IF <condition> expression, and the closing expression is two words, not just one as in VFP. The coolest thing about this is that you can nest multiple IF conditions using the ELSEIF <condition> expression instead of having to build multiple IF/ELSE/ENDIF constructs as you do in VFP. Also take note that each condition can be different!

Choosing between multiple choices

The SELECT CASE construct is VB's version of DO CASE in VFP. It, too, is a bit more powerful, as shown in the following code fragment:

```
SELECT CASE sFileExtension
 <stuff to do>
CASE "BAT"
<stuff to do>
CASE "SYS"
<stuff to do>
CASE "DBF", "CDX", "FPT"
<stuff to do>
CASE "MAA" to "MZZ"
<stuff to do>
CASE ELSE
<stuff to do>
END SELECT
```

This construct takes as input a string expression, sFileExtension, that contains the extension of a file, and processes it according to what extension it is. Thus, .BAT files get one treatment, .SYS files get a different treatment, and so on. This is different from VFP in that you can use multiple, inconsitent conditional expressions following each CASE statement in VFP, and, thus, you have to use the entire conditional expression Note that you can include multiple values following VB's CASE statement, as in the third CASE statement. The code following that piece will be executed for DBF, CDX and FPT files. In VFP, you'd have to write an expression like so:

```
CASE inlist(cFileExtension, "DBF", "CDX", "FPT")
```

Note that you can also have VB span a series of values, as in the fourth CASE where every extension between MAA and MZZ will be treated the same. CASE ELSE works the same as OTEHRWISE in VFP.

Looping

Somebody in Redmond went bonkers when it came time to assemble looping constructs in VB, because there are five of them. I'll describe the first four and then explain why you shouldn't use the fifth.

The FOR NEXT construct is the basic looping construct, as shown in this code fragment:

```
FOR nIndex = 1 to 10 STEP 2
  <some code to run>
  IF <some bad condition occurred> THEN
    EXIT FOR
  END IF
  <more code could be here>
NEXT
```

You'll see that NEXT is used in place of END FOR in VFP, although many of you are probably using the (sort of new) NEXT keyword in VFP instead of END FOR. (If you're not, you should, because it will cut down on the confusion generated when switching between languages.) You'll also see that the escape route out of a FOR loop is EXIT. Big surprise.

If your index expression is an element in a collection of some sort, you can use FOR EACH much easier.

```
FOR EACH <element> IN <group>
  <some code to run>
  IF <some condition found the right element> THEN
    EXIT FOR
  END IF
  <more code could be here>
NEXT
```

So far, so good. The next two logic structures are pretty similar: DO WHILE and DO UNTIL. In fact, you can probably noodle them out yourself, right? Not so fast!

```
DO WHILE <condition>
  <some code to run>
LOOP

DO UNTIL <condition>
  <some code to run>
LOOP
```

The trick to both of these is that while the above syntax looks comfortable to you, you're most likely not going to see it in experienced VB'ers code. That's because you can also put the WHILE and UNTIL expressions after the LOOP keyword, like so:

```
DO
  <some code to run>
LOOP WHILE <condition>

DO
  <some code to run>
LOOP UNTIL <condition>
```

Obviously, doing so changes how the loop is processed – much like incrementing a counter at the beginning or the end of a loop. You can use the EXIT DO statement to get out of DO JAIL early, like so:

```
DO WHILE <condition>
  <some code to run>
  IF <some bad condition occurred> THEN
    EXIT DO
  END IF
LOOP
```

The WHILE/WEND construct is a cheap, poorly imitated version of DO WHILE, because you can't use EXIT DO to escape early. If that's not important to you, then you can use WHILE/WEND instead. Keep in mind, however, that if your requirements change and you have to be able to terminate processing early, you'll have to recode your WHILE/WEND to DO WHILEs. Why not save yourself the inevitable hassle now?

Instead of trying to describe, in detail, each VB statement, I've just listed the rest of them, so that you can get a quick idea of what functionality is available to you natively, and, thus, what you'll have to write yourself or live without.

File Handling

Close	Concludes IO to a file opened using OPEN
Get	Reads data from an open disk file into a variable
Input #	Reads data from an open sequential file and assigns the data to variables
Let	Assigns the value of an expression to a variable or property
Line	Input # Reads a single line from an open sequential file and assigns it to a string variable
Lock, Unlock	Controls access by other processes to all or part of a file opened using OPEN
Lset	Left aligns a string with a string variable, copies a variable of one user-defined type to another variable of a different user-defined type
Mid	Replaces a specified number of characters in a variant (string) variable with characters from another string
Open	Enables IO to a file
Print #	Writes display-formatted data to a sequential file
Put	Writes data from a variable to a disk file
Reset	Closes all disk files opened using OPEN
Rset	Right aligns a string within a string variable
Seek	Sets the position for the next read/write operation within a file opened using OPEN
Width #	Assigns an output line width to a file opened using OPEN
Write #	Writes data to a sequential file

Program Control

Call	Transfers control to a Sub procedure, Function procedure or a DLL
Function	Declares the name, arguments and code that form the body of a function procedure
GoSub	Branches to and returns from a subroutine within a single procedure
GoTo	Branches unconditionally to a line within a procedure
On GoSub, On GoTo	Branches to a line depending on the value of an expression
Stop	Suspends execution of a program
Sub	Declares the name, arguments, and code that form the body of a sub procedure
With	Executes a series of statements on a single object or user-defined type

File System

ChDir	Changes the current directory or folder
ChDrive	Changes the current drive
FileCopy	Copies a file
Kill	Deletes files from a disk
MkDir	Creates a new directory
Name	Renames a directory or file
RmDir	Deletes a directory
SavePicture	Saves a graphic from the picture or image property of a control to a file
SetAttr	Sets attributes information for a file

Variables

Const	Declares constants for use in place of literal values
Declare	Declares references to external procedures in a DLL – used at module level
Deftype	Set default data type for variables, arguments passed to procedures, and return type for Functiona nd Property Get procedures – used at module level
Dim	Declares variables and allocates storage space
Enum	Declares a type for an enumeration
Erase	Reinitializes the elements of a fixed size array and releases dynamic array storage space
Option Base	Declares the default lower bound for array subscripts – used at the module level

Option Compare	Declare the default comparison method to use when string data is compared – used at the module level
Option Explicit	Forces explicit declaration of all variables in that module – used at the module level
Option Private	Prevents a module’s contents from being referenced outside its project
Private	Declares private variables and allocates storage space – used at the module level
Property Get	Declares the name, argument and code that form the body of a property procedure which gets the value of a property
Property Let	Declares the name, argument and code that form the body of a property procedure which assigns the value to a property
Property Set	Declares the name, argument and code that form the body of a property procedure which sets a reference to an object
Public	Declares public variables and allocates storage space – used at the module level
Randomize	Initializes the random-number generator
ReDim	Reallocates storage space for dynamic array variables
Static	Declares variables and allocates storage space – used at the procedure level
Type	Defines a user-defined data type containing one or more elements – used at the module level

Registry

DeleteSetting	Deletes a section or key setting from an application’s entry in the Registry
SaveSetting	Saves or creates an application entry in the application’s entry in the Registry

Classes

Implements	Specifies an interface or class that will be implemented in the class module in which it appears
Load	Loads a form or control into memory
Set	Assigns an object reference to a variable or property
Unload	Unloads a form or control from memory

N.E.C.

Date	Sets the system date
Error	Simulates an error
Event	Declares a user-defined event
On Error	Enables an error-handling routine and specifies the location of the routine within a procedure
Resume	Resumes execution after an error-handling routine is finished
Rem	Identifies a comment
RaiseEvent	Fires an event
SendKeys	Sends one or more keystrokes to the active window as if typed at the keyboard
Time	Sets the system time

Conclusion

If you look through these commands, you’ll see that they seem to stop well short of some of the functionality you’re used to in VFP. That’s because much of the language is a throwback to BASIC before it became “Visual Basic” and thus still contains legacy references to opening and closing files and such. Most of the work you’ll do in VB now will be done visually – connecting to data will be done by dropping data-aware controls on a form, not through a series of commands like “USE” and “APPEND FROM” like VFP still does.

And one last thing: remember that you can’t abbreviate to the first four characters as you can in VFP! Next month, I’ll tackle the second most interesting part of application development – building user interfaces through forms and controls – and in subsequent months, I’ll address the really fun part – data.

Whil Hentzen is editor of FoxTalk. whil@hentzenwerke.com.

## make this a sidebar

Stupid VB (Windows) Tricks

Differences between VB and VFP, and inconsistencies in the VB environment that can "gotcha" if you're not careful or observant. This month, there’s a trap in the way different operating systems handle dates.

Stupid VB trick #5:

The DATE statement sets the current system date. That seems reasonable. However, it appears that the emphasis that Microsoft puts on the term “cross platform” is well warranted, as there is a difference in permissible date specifications depending on whether the operating system is Win 9x or Windows NT. 9x can handle system dates between 1/1/1980 and 12/31/2099. However, NT can only handle dates between 1/1/1980 and 12/31/2079. Hmmm, obviously a conspiracy. Do you think maybe this means that Microsoft is going to kill NT, and they're just stringing us along for 80 years? Hmmmm?