Obscure and obsolete VB syntax

Visual Basic 6.0 and Visual Basic for Applications support a great deal of questionable and not-so-useful legacy statements and keywords. This article looks into confusing VB statements and old-style syntax that is rare today.

A language with a long history will eventually have some outdated syntax features. Understanding old code may be hard if it contains undocumented keywords or commands you don't normally use. This is also true for Visual Basic.

Much of VB's old syntax comes from its DOS predecessors. The immediate predecessor of VB was Microsoft QuickBasic, a programming language for DOS. Syntax-wise, QuickBasic was quite similar to VB. For backwards compatibility, VB supports many syntax features that were useful in QuickBasic, even though they are no longer very useful in VB. What is more, the Visual Basic language has been updated between its various versions, rendering a part of the language obsolete.

This article focuses on the oddities of Visual Basic 6.0 and VBA. Reading the article helps you to avoid and eliminate potentially confusing code.

> PA This sign means that Project Analyzer detects the syntax mentioned so that you can rewrite it. Project Analyzer is a Visual Basic and VBA code analyzer that runs code review to help you improve your projects.

Call

```
Call subname(arguments)
```

`Call` is a superfluous keyword to call sub-programs. The keyword doesn't do anything useful. You can remove it. Confusingly, if you do remove it, you will need to also remove the parentheses around the argument list. Otherwise you are likely to end up with an error. Therefore, it's often safer to leave `Call` alone unless you want to introduce new bugs to your program.

A particularly nasty case is `Call` with one parameter: `Call MySub(x)`. If you remove the word `Call`, the code stays the same, right? Wrong. The parentheses in the `Call`-less syntax `MySub (x)` force passing x by value. If `MySub` happens to declare the parameter as `ByRef` and change its value on purpose, the change will no longer be propagated back to the caller. This could break existing code. So, if you do remove `Call`, remember to remove the parentheses too. PA

Date and Time statements

```
Date = expr : Time = expr
Date$ = expr : Time$ = expr
```

You could set the current date and time with `Date = date` and `Time = time`. You could, if you were an Administrator. If not, you will receive a run-time error 70: `Permission denied`.

Why would have a VB program change the system clock? If the PC didn't have a battery-powered clock, it might be a good idea to let the user set the date and time. Today's PCs get the time from an online time server. There is no need to ask the user what date it is.

Interestingly, these needless statements come in two versions: with and without the dollar sign. What's the point?

Declare CDecl

```
Declare Sub|Function name CDecl Lib "…" (…)
```

`CDecl` is an undocumented keyword in Visual Basic. According to online sources, `CDecl` is only implemented in the Mac version of VBA. In all other versions it's a no-op. So what is it for?

In QuickBasic, the `Declare` statement was used to call both Basic and non-Basic procedures. The default was Basic, which meant that procedure arguments were passed from left to right. According to QuickBasic 4.5 help, the optional `CDecl` keyword specified that the declared procedure used C-language argument order instead: arguments were passed from right to left. This means that `CDecl` = pass arguments from right to left.

In Visual Basic (less the Mac VBA version), all `Declare` statements pass their arguments from right to left. Thus, `CDecl` doesn't do anything any more. `CDecl` can be omitted and will be deleted by VB IDE. Confusingly enough, `CDecl` is not the same as `cdecl`. In Windows, there are 2 different calling conventions to consider: `cdecl` and `stdcall`. Event though they are different, both of them use right-to-left passing order. Visual Basic supports only `stdcall`, not `cdecl`. Even if you manage to add `CDecl` to your `Declare` statement, you can't have VB use `cdecl`. Weird.

DefType

```
DefBool | DefByte | DefInt | DefLng | DefLngLng | DefLngPtr | DefCur | DefSng | DefDbl | DefDate
| DefStr | DefObj | DefVar letter1[-letter2]
```

The `DefType` statements declare a data type for variables that would otherwise become Variants. They affect declared variables without a data type specification and also undeclared local variables.

`DefType` was handy in the QuickBasic era. The default datatype was `Single`, and there was no way to require the declaration of variables. You would use type declaration suffixes such as `s$` for `String` and `i%` for `Integer`. With `DefType` you could get rid of the suffixes without changing your habits of not declaring your variables. `DefInt I` would set all i-variables to `Integer`, `DefStr S` would force s-variables to `String` and so forth. This way you could have the variable tell its data type by its initial letter.

In Visual Basic, the default datatype is `Variant`, which is much more flexible if you happen to leave your variables undeclared. Variants are so flexible your code will usually work as intended even if you know nothing about data types. What is more, you can use `Option Explicit` to require explicit declaration of all local variables. `DefType` has thus become less useful. It may be confusing because developers may not expect to find it in today's code. Bugs are likely to get introduced when the data type of an untyped variable is not `Variant` as one would normally expect. PA

Dim Shared

```
Dim Shared varname
```

`Shared` is a keyword previously used with `Dim` statements. It has no effect. `Shared` exists for backwards compatibility with pre-VB languages such as Microsoft QuickBasic and Basic Compiler. In QuickBasic, `Dim Shared` was used to declare module-level variables to be available in procedures, similar to module-level variables in VB. `Shared` has no effect in VB because all module-level variables are always available in the procedures of the module. If you write `Dim Shared` in VB6, the word `Shared` disappears and only `Dim` remains.

Eqv

```
x Eqv y
```

The `Eqv` operator makes a bitwise equivalence comparison of two integers. `Eqv` is not an "is equivalent to" operator but rather a kind of "`Not Xor`" operator. `x Eqv y` is the same as `Not (x Xor y)`.

It's important to realize that `Eqv` is not an alternative to the equals operator (`=`). While `2 = 3` returns `False`, `2 Eqv 3` returns a non-zero result and is `True`. `Eqv` will return zero only when no bits of the operands are equal.

Erl

The `Erl` function is used in error handlers. It returns the line number where an error occurred. If there is no line number on the line that failed, `Erl` returns the preceding line number. `Erl` doesn't work with line labels. It doesn't work with large line numbers either, the current maximum being 65535. If the line number is any larger, `Erl` will return an unexpected number instead. A major disadvantage of `Erl` is that it requires the source to be line numbered, which is rarely the case with VB, unless you use a tool such as VB Watch that deliberately adds the line numbers for error handling.

Floating point numbers with D exponent

Floating point numbers, which are usually written like `1E+6`, can also be written in an alternative syntax: `1D+6`. Both represent a million: 6 zeros after 1.

The legacy D-syntax originally declared a `Double` value, while the E-syntax was for `Single` values. According to VB 1.0 help, the largest E-value was `3.402823E38`, but with D you could go up to `1.797693134862315D308`.

In VB6 and VBA, there is no difference any more. They will automatically replace D with E. You can still have D-values in strings, though. Thus, `Val("1D+6") = 1000000`. This can be confusing if you don't know the explanation.

Global

`Global` is a synonym for `Public`. In the good old days, `Global` was the only VB keyword that could modify the scope of a variable or a constant. Keywords `Public`, `Private` and `Friend` didn't exist.

VB6 and VBA still allow you to use `Global` to define variables and constants. But, if you try it on a `Sub`, `Function`, `Property`, `Enum` or `Type`, VB will (normally) delete the word and behave as if you had written `Public`.

GoSub, Return

```
GoSub line
line: … : Return
```

`GoSub` is a way to call a sub-routine within the procedure itself. `GoSub` will jump to the given line, execute the statements following it up to a `Return` statement. A `Return` will resume from the statement next to `GoSub`.

This was the only way to have sub-routines before Subs and Functions were available. That is, a long time before VB was invented. You wouldn't normally use `GoSub` in VB. Subs and Functions are much more structured.

There is still one special use for `GoSub` in VB. That's in a procedure with a lot of local variables. If you want to access local variables in another `Sub` or `Function`, you need to either pass them as parameters or move variable definitions to the module level. This may not always be desirable. By implementing the sub-routine with `GoSub` and `Return`, you avoid passing the local data, which may come handy. This can easily result in an unnecessarily complex routine, though. So it's best to leave `GoSub` unused if you don't really, really need it.
PA

If..GoTo

```
If condition GoTo line
```

Did you know you can write an `If` statement without `Then`? Yes, it's true, but for single-line `If` statements only. `If..GoTo` is the equivalent of `If..Then GoTo`. VB6 and VBA will add the `Then` keyword automatically, but technically `If..GoTo` is still acceptable.

If..Then 10 Else 20

```
If condition Then linenum [Else linenum]
```

Did you know the `GoTo` statement is optional after `Then` and `Else`? This is true in single-line `If` statements. You can omit `GoTo` and simply add the target line number. This won't work with line labels, though. This was originally a documented feature of the VB language. VB6 and VBA, however, will add the omitted `GoTo`, forcing you to use `GoTo` and not the shorter `GoTo`-less format.

Imp

```
x Imp y
```

`Imp` is the logical-implication operator: $x \rightarrow y$. Few people understand it correctly. The rationale behind `Imp` is hard to grasp unless you're involved with logic. It's best to rewrite it. `x Imp y` is the same as `(Not x) Or y`. The latter one is much more clear—unless all developers working on the code remember logic operators from their logic course.

Int

```
Int(number)
```

The `Int` function returns the integer part of a decimal number? Wrong. It doesn't. It returns the largest integer less than or equal to the number. `Int(3.1) = 3`, but `Int(-3.1) = -4`. This is a source of errors and confusion.

The `Fix` function will always return the integer part. `Fix(-3.1) = -3`. When negative numbers are expected, `Fix` is more natural than `Int`.

Let

```
[Let] variable = value
```

The `Let` keyword is superfluous. There is no reason to use it. Let doesn't add anything to your program, nor does it make it easier to read or understand. You can always omit it. VB does allow you to write it, but it should automatically delete it, really. PA

Line numbers

Line numbers are truly outdated Basic. A long, long time ago, all Basic lines were numbered. This is no longer the case. Line numbers tend to appear only as targets of `GoTo`, `GoSub`, `On Error GoTo` and `Resume`. Instead of line numbers, you can use the much more descriptive line labels. Line numbers are rare now.

The only useful use for line numbers is error handling. In an error handler, the Erl function returns the line number where the error occurred (as long as the number is 65535 or less). This functionality is not available in any other way.

In VB6 and VBA, the minimum line number is 0 and the undocumented maximum is 999999999 (9 digits). Line numbers don't have to appear in numeric order. It's perfectly valid to start with 1000 and go down to 0.

LSet, RSet

```
LSet variable = value
RSet variable = value
```

`LSet` and `RSet` are designed to left-align and right-align text in a string variable. `LSet x = y` will copy the contents of `y` to `x` while maintaining the current length of `x`. If `x` is too short, the string is truncated. If `x` is longer than `y`, the rest is padded with spaces on the right, making the string left-aligned. `RSet` does the same, but the padding will appear on the left instead, making the string right-aligned.

These statements are apparently designed for fixed-length strings. Back in the good old DOS days, fixed-length text fields were common on displays and reports, which used fixed-pitch fonts. As user interfaces have become more flexible, fixed-length strings are not so common any longer.

`LSet` can also be used to copy a variable of a user-defined type onto other user-defined types. This could be dangerous and should only be used with care.

Next with multiple variables

```
Next variable, variable, variable…
```

You don't necessarily need a `Next` statement for each `For` statement. A single `Next` statement can end multiple nested `For` loops. `Next k, j, i` is the equivalent of `Next k : Next j : Next i`. This syntax makes sense rarely if ever. PA

Octal numbers

Octal numbers are expressed in digits 0-7. For example, `&O10` is the octal notation for decimal 8.

Octal numbers can be considered obsolete. Since an octal digit represents 3 bits, they were useful on systems with a word size divisible by 3: 6-bit words, 12-bit words and so on. Today's systems don't use such word sizes. Visual Basic runs on computers with 8-bit bytes and 32-bit or 64-bit words. In this environment, hexadecimal and decimal numbers are handier and much more common. Since octal numbers are relatively rare, they can cause problems reading the code. To avoid confusion, express numeric values in either decimal or hexadecimal format, which are more widely understood. PA

On Error GoTo -1

The largely undocumented `On Error GoTo -1` statement is obscure. This statement doesn't set an error handler, nor does it unset it like `On Error GoTo 0` does. The statement clears the error that just occurred in preparation for a new error to occur. It can be used in an error handler that could trigger a new error. If an error occurs within an error handler, execution of the procedure will stop (and flow back to caller or cause a crash). Calling `On Error GoTo -1` allows you to set a "nested error handler" for taking care of errors occurring in an error handler. Example:

```
On Error GoTo mainhandler
Error 5

mainhandler:
MsgBox "Error occurred"
On Error GoTo -1 ' Clear the error
On Error GoTo nestedhandler
Error 6

nestedhandler:
MsgBox "Error occurred in error handler"
```

In this example, `Error 5` will trigger `mainhandler` and `Error 6` will trigger `nestedhandler`. Should you leave out the line with `On Error GoTo -1`, `nestedhandler` would not execute.

On..GoSub, On..GoTo

```
On number GoSub line1, line2, line3…
On number GoTo line1, line2, line3…
```

`On..GoSub` and `On..GoTo` are primitive alternatives to `Select Case`. Both statements provide a way to execute one of multiple branches based on number. If *number* is one, execution continues on *line1*, if it's two, then *line2* and so on.

The `GoSub` and `GoTo` statements alone are not structured programming. The same can be said about `On..GoSub` and `On..GoTo`, which are about as obscure as VB statements can get. The existence of these statements in VB code calls for a rewrite. PA

On Local Error

```
On [Local] Error GoTo|Resume …
```

The `Local` keyword in an `On Error` statement has no effect. In Visual Basic, `On Error` statements are local whether you use the `Local` keyword or not. Leave `Local` out to avoid confusing anyone with the difference of local and non-local error handlers, as the only option is a local error handler anyway. PA

`Local` is a legacy keyword. According to VB 1.0 help, it was originally intended for compatibility with Microsoft QuickBasic and Basic Compiler. The documentation appears incorrect because QuickBasic 4.5 doesn't support it.

Operator + on strings

```
string1 + string2
```

The plus (`+`) operator will usually concatenate two strings. For any two strings, `a + b = a & b`. But, if either argument is a number, VB will attempt to sum them up:

```
"2" + "3" = "23"
"2" +  3  =  5
```

Confusing! An error occurs when adding text to a number:

```
"a" + 3 ⇒ error
```

It's best to avoid `+` with strings. VB should warn about it.

Option Base

```
Option Basic 0|1
```

`Option Base` is a legacy statement that defines the lower bound of arrays. The possible lower bounds are 0 and 1, the default being 0.

The statement is unnecessary in Visual Basic. VB has always let programmers declare the lower bound of arrays in `Dim` statements.

Option Private Module

This statement is completely useless in VB6. It does have a use in VBA, though. In VBA, `Option Private Module` prevents the contents of a module from being used outside its own project.

Rem

```
Rem remark
```

`Rem` is an old way to add remarks to a program. Visual Basic has supported the handier apostrophe (`'`) syntax since version 1.0. There has never been any reason to use `Rem`. It's there for backwards compatibility only. PA

Reset, Close

The `Reset` statement closes all open files and supposedly writes all file buffers to disk. The `Close` statement, if not followed by any file numbers, also closes all files, but it doesn't flush buffers. According to VB 1.0 help, `Reset` was supposed to be called after `Close`. The documentation of VB 6.0 doesn't refer to this use. It's unclear if there is any actual difference. Another issue is, why on earth would you want to keep your files open and close them all at once? Wouldn't you want to close each file as soon as it's not required, not leaving any used files open unnecessarily? It's better to use `Close` *`filenum`* to explicitly close each file right after use and not attract bugs by leaving files open and closing them with a single statement.

Resume 0

In an error handler, `Resume 0` will restart execution from the statement that failed. `Resume 0` is exactly the same as `Resume` without the zero. The zero is both unnecessary and confusing. Anyone reading the code could easily think execution will redo from line 0 and wonder where line 0 might be. It's best not to have the zero in the statement.

Static Sub, Static Function, Static Property

The `Static` attribute in a procedure header forces all local variables to be `Static` in that procedure. This is a way to avoid declaring each variable with the `Static` *`varname`* statement. This programming style possibly dates back to the era when local variables were not declared at all.

`Static` variables are preserved between calls to the procedure. That's all right. Having all variables `Static` is another thing. It could confuse a person who's reading or modifying the code. There are uses for `Static` variables, but it's more explicit to declare each variable with the `Static` statement rather than relying on the `Static` keyword in the procedure declaration. Thus, stay away from `Static Sub`, `Static Function` and `Static Property`, and use `Static` *`varname`* instead.

Type declaration characters (!#$%&@^)

Type declaration characters are from an era before Visual Basic. VB has always allowed you to declare datatype with the `As Datatype` clause. You can declare ! `As Single`, # `As Double`, $ `As String`, % `As Integer`, & `As Long`, @ `As Currency` and ^ `As LongLong`. The last option is available in VBA 7. PA

While..Wend

```
While condition : statements : Wend
```

The `While..Wend` loop syntax is very old. `Do..Loop`, available already before VB was invented, has replaced it a long time ago. Today you should use `Do..Loop` as it's more versatile. For most uses, `While..Wend` is exactly the same as `Do While..Loop`. `While..Wend` could confuse developers who don't recognize the obsolete syntax. PA

A special case requiring the use of `While..Wend` does exist. This is when you need to nest two or more loops and you want to exit several loops with one command.

```
Do
  While condition
    …
    If quit Then Exit Do
    …
  Wend
Loop
```

In this example, `Exit Do` will exit both the `Do` and `While` loops simultaneously. Clever! You couldn't do this with two nested `Do` loops.

Width

```
Width #filenum, value
```

The `Width` statement is intended for writing text files. It forces a fixed line length on an output file. The `Width` statement affects successive `Print #` statements but not `Write #` statements. If a line would become longer than desired, text will wrap to the next line in the output file. While this could be useful in some cases, forced line wrapping in the middle of words isn't a very modern thing.

> PA This sign means that Project Analyzer detects the syntax mentioned so that you can rewrite it. Project Analyzer is a Visual Basic and VBA code analyzer that runs code review to help you improve your projects.