

Visual Basic Programming Conventions from Microsoft Consulting Services

1. Naming Conventions

1.1 Objectives

- n To help programmers (especially in multiprogrammer projects) standardize and decode the structure and logic of an application
- n To be precise, complete, readable, memorable, and unambiguous
- n To be consistent with other language conventions (most importantly the Visual Basic™ *Programmer's Guide* and standard Microsoft® Windows™ Hungarian C)
- n To be efficient from a string size and labor standpoint, thus allowing a greater opportunity for longer/fuller object names
- n To define the minimum requirements necessary to do the above

Note On pages 32 and 33 of the Visual Basic 2 *Programmer's Guide*, Visual Basic has defined control naming conventions that will likely be adopted by many corporations and Visual Basic ISVs. As a result, it would take a very strong argument to justify deviating from the Visual Basic 2 standard without causing a lot of heartache and confusion. Lacking such a compelling argument, this document therefore is a superset of the published Visual Basic conventions.

1.2 Conventions

1.2.1 Option Explicit

"Option Explicit" must always be used to force proper variable declarations and aid good variable commenting. The time lost trying to track down bugs caused by typos (*aUserNameTmp* vs. *sUserNameTmp* vs. *sUserNameTemp*) far outweighs the time needed to Dim variables.

1.2.2 Control naming

The following table defines our standard Control name prefixes. (These are consistent with those documented in the Visual Basic 2 *Programmers Guide*.)

Table 1. Standard Control Name Prefixes

Pr ef ix	Control Type Description
i	an Animation button

be Pen Bedit
d

cb Combobox and
o dropdown Listbox

ch Checkbox
k

cl Picture Clip
p

c Command Button
m
d

co Communications
m

ctr Control (Used within
procs when the specific
type is unknown)

db ODBC Database

dir Dir List Box

dl Visual Basic Pro
g Common Dialog

dr Drive List Box
v

ds ODBC Dynaset

fil File List Box

fr Form
m

fr Frame
a

ga Gauge
u

gp Group Push Button
b

gr Grid
d

he Pen Hedit
d

hs Horizontal Scroll Bar
b

im Image
g

in Pen Ink
k

ke Keyboard key status
 y
 lbl Label
 lin Line
 lst Listbox
 m MAPI Message
 p
 m
 m MAPI Session
 ps
 m MCI
 ci
 m Menu
 nu
 op Option Button
 t
 ol Ole Client
 e
 pi Picture
 c
 pn 3d Panel
 l
 sh Shape
 p
 sp Spin Control
 n
 txt Text/Edit Box
 tm Timer
 r
 vs Vertical Scroll Bar
 b

1.3 Control Prefix Notes

1.3.1 Menus

Because Menu handlers can be so numerous, Menu names require a little more attention. Menu prefixes therefore continue beyond the initial Mnu label by adding an additional (upper case) character prefix for each level of nesting, with the final menu caption being spelled out at the end of the name string. When there is ambiguity caused by character duplications, such as a menu having both main Format and File menus, use an additional (lower case) character to differentiate the items. Examples:

Menu	Menu Handler
------	--------------

Caption	Name Sequence
	Help.Contents mnuHContents
	File.Open mnuFiOpen
	Format.Character mnuFoCharacter
	File.Send.Fax mnuFSFax
	File.Send.Email mnuFSEmail

This results in all the family members of a particular menu group being listed right next to each other. This multi-tiered format provides a very direct way to find a menu handler, especially when there are a great many of them.

1.3.2 Other controls

For new controls not listed above, try to come up with a unique 3-character prefix. However, it is more important to be clear than to stick to 3 characters. For derivative controls, such as an enhanced list box, extend the prefixes above so that there is no confusion about what control is really being used. For example, a control instance created from the Visual Basic Pro 1.0 3D Frame could use a prefix of fra3d to make sure there is no confusion over which control is really being used.

1.3.3 Variable and routine naming

Variable and function names have the following structure:

```
<prefix><body><qualifier><suffix>
```

The prefix describes the use and the scope of the variable, as in *iGetRecordNext* and *sGetNameFirst*. The qualifier is used to denote standard derivatives of a base variable or function, as in *iGetRecordNext* and *sGetNameFirst*. The suffix is the optional Visual Basic type char (\$, %, #, and so on).

Prefixes

The following table defines variable/function name prefixes that are based on Hungarian C. These must be used universally, even when Visual Basic suffixes (such as %, &, #, and so on) are also used.

Table 2. Prefixes for Variable and Function Names

Pr Variable Use
efi Description
x (precedes Control

prefix and body)

- b* Boolean (vb type = %)
- c* Currency - 64 bits (vb type = @)
- d* Double - 64 bit signed quantity (vb type = #)
- db* Database
- ds* Dynaset
- dt* Date+Time (vb type = variant)
- f* Float/Single - 32 bit signed floating point (vb type = !)
- h* Handle (vb type = %)
- i* Index (vb type = %)
- l* Long - 32 bit signed quantity (vb type = &)
- n* Integer (sizeless, counter) (vb type = %)
- s* String (vb type = \$)
- u* Unsigned - 16 bit unsigned quantity (must use &)
- ul* Unsigned Long - 32 bit unsigned quantity (must use #)
- vn* Variant (big and ugly to discourage use and make sure it gets the reader's attention)
- t*
- w* Word - 16 bit signed quantity (vb type = %)
- a* Array
- User defined type

**Pr Scope or Use
efi (precedes Use prefix
x above)**

- g* Global
- m* Local to module or form
- st* Static variable

- v* Variable passed by value (local to a routine)
- r* Variable passed by reference (local to a routine)

Hungarian is as valuable in Visual Basic as it is in C because the Visual Basic type suffixes alone do not provide standard (and valuable) information about what a variable/function is used for or where it is accessible. For example, *iSend* (which might be a count of the number of messages sent), *bSend* (which might be a flag/Boolean defining the success of the last Send operation), and *hSend* (which might be a handle to the Comm interface) all *succinctly* tell a programmer something very different. This information is fundamentally lost when the name is reduced down to *Send%*. Scope prefixes such as *g* and *m* also help reduce the problem of name contention, especially in multideveloper projects. Hungarian is also well known to Windows programmers and constantly referenced in Microsoft and industry programming books. Additionally, the bond between C programmers and Visual Basic programmers can be expected to become much stronger as Visual C++ begins to live up to its potential. This transition will result in many Visual Basic programmers moving to C for the first time and many programmers moving fluidly back and forth between each environment.

Body

The body of variable and routine names should use mixed case and should be as long as needed to describe their purpose. Function names should also begin with a verb, such as **InitNameArray** or **CloseDialog**.

For frequently used or long terms, abbreviations (such as Init, Num, Tbl, Cnt, and Grp for Initialization, Number, Table, Count, and Group) are suggested to help keep name lengths reasonable. Names greater than 32 characters generally begin to inhibit readability on VGA displays. When abbreviations are used, they must be used consistently throughout the application. Randomly switching between "Cnt" and "Count" within a project will greatly frustrate developers.

Qualifiers

Often related variables and routines are used to manage and manipulate a common object. In these cases it can be very helpful to use standard qualifiers to label the derivative variables and routines. Although putting the qualifier after the body of the name might seem a little awkward (as in *sGetNameFirst*, *sGetNameLast* instead of *sGetFirstName*, and so on), this practice will help order these names together in the Visual Basic editor routine lists, making the logic and structure of the application easier to understand.

The following table defines common qualifiers and their standard meaning.

Table 3. Common Qualifiers

Q ua lifi er	Description (follows Body)
Fir st	First element of a set.

La Last element of a set.
st

Ne Next element in a set.
xt

Pr Previous element in a
ev set.

Cu Current element in a
r set.

Mi Minimum value in a set.
n

M Maximum value in a
ax set.

Sa Used to preserve
ve another variable which
must be reset later.

T A "scratch" variable
m whose scope is highly
p localized within the
code. The value of a
Tmp variable is usually
only valid across a set
of contiguous
statements.

Sr Source. Frequently
c used in comparison and
transfer routines.

Ds Destination. Often used
t in conjunction with
Source.

1.3.4 Constant naming

- n The body of constant names are described in UPPER_CASE with underscores ("_") between words.
- n Although standard Visual Basic constants do not include Hungarian use information, prefixes such as i, s, g, and m can be very useful in understanding the value and scope of a constant, so constant names follow the same rules as variables. Examples:

mnU 'Max entry limit
SER for User list
_LIS (integer value,
T_M local to module)
AX

gsNE 'New Line
W_LI character string
NE (global to entire
application)

1.3.5 Variant data types

With the single exception listed below, variants should NOT be used. When a type conversion is needed, variant use would probably provide a slight performance win over the explicit basic type conversion routines (`val()`, `str$()`, and the like), but this gain is not sufficient to overcome the ambiguity and general sloppiness they allow in code statements.

Example:

```
vnt1 = "10.01" : vnt2 = 11 : vnt3 = "11" : vnt4 = "x4"
vntResult = vnt1 + vnt2      ' Does vntResult = 21.01 or 10.0111?
vntResult = vnt2 + vnt1      ' Does vntResult = 21.01 or 1110.01?
vntResult = vnt1 + vnt3      ' Does vntResult = 21.01 or 10.0111?
vntResult = vnt3 + vnt1      ' Does vntResult = 21.01 or 1110.01?
vntResult = vnt2 + vnt4      ' Does vntResult = 11x4 or ERROR?
vntResult = vnt3 + vnt4      ' Does vntResult = 11x4 or ERROR?
```

Additionally, the type conversion routines assist in documenting implementation details, which make reading, debugging, and maintaining code more straightforward.

Example:

```
(iVar1 = 5 + val(sVar2)      'use this
vntVar1 = 5 + vntVar2        'not this!
```

Exception

While working with databases, messages, DDE, or OLE, a generic service routine can receive data that it does not need to know the type of in order to process or pass on.

Example:

```
Sub ConvertNulls(rvntOrg As Variant, rvntSub As Variant)
  'If rvntOrg = Null, replace the Null with rvntSub
  If IsNull(rvntOrg) Then rvntOrg = rvntSub
End Sub
```

2. Commenting

- n All procedures and functions must begin with a brief comment describing the *functional* characteristics of the routine (what it does). This description should *not* describe the implementation details (how it does it) because these often change over time, resulting in unnecessary comment maintenance work or, worse, erroneous comments. The code itself and any necessary inline or local comments will describe the implementation. Parameters passed to a routine should be described (a) if they are not obvious and (b) when specific ranges are assumed by the routine. Function return values and global variables that are changed by the routine (especially through reference parameters) must also be described at the beginning of each routine.
- n Every nontrivial variable declaration should include an inline comment describing the use of the variable being declared.
- n Variables, controls, and routines should be named clearly enough that inline commenting is only needed for complex or nonobvious implementation details.

- n An overview description of the application enumerating primary data objects, routines, algorithms, user interface dialogs, database and file system dependencies, and so on, should be included at the start of the .BAS module that contains the project's Visual Basic generic constant declarations.

Note The Project window inherently describes the list of files in a project, so this overview section only needs to provide information on the most important files and modules or files that the Project window doesn't know about, such as .INI or database files.

3. Code Formatting

Because many programmers still use VGA displays, screen real estate must be conserved as much as possible while still allowing code formatting to reflect logic structure and nesting. For this reason:

- n Standard (tab-based) block nesting indentations should be from two to four spaces. More than four spaces is unnecessary and causes unnecessary statement hiding through truncation. Fewer than two is not effective in reflecting logic nesting.
- n The functional overview comment of a routine should be indented one space. The highest level statements that follow the overview comment should be indented one tab, with each nested block indented an additional tab. Example:

```
Function iFindUser (rasUserList() as String, rsTargetUser as String) as Integer
    'Search UserList and if found, return index of first occurrence of TargetUser,
    ' else return -1
    Dim i as Integer                'loop counter
    Dim bFound as Integer           'target found flag
    iFindUser = -1
    i = 0
    While i <= Ubound(rasUserList) and Not bFound
        If rasUserList(i) = rsTargetUser Then
            bFound = True
            iFindUser = i
        End If
    Wend
End Function
```

- n Variables and nongeneric constants should be grouped by function rather than being split off into isolated areas or special files. (Visual Basic generic constants such as "HOURGLASS" should be grouped in a generic section of a main global file so that they do not complicate the reading of the application-specific declarations.)

4. Operators

- n Always use "&" when concatenating strings and "+" when working with numerical values. Using only "+" can cause problems when operating on two variants. For example:

```
vntVar1 = "10.01"
vntVar2 = 11
vntResult = vntVar1 + vntVar2                'vntResult = 21.01
vntResult = vntVar1 & vntVar2                'vntResult = 10.0111
```

5. Scope

- n Always define variables with the smallest scope possible. Global variables can create enormously complex state machines and make understanding the logic of an application extremely difficult. They also make the reuse and maintenance of your code much more difficult. If you have to use globals, keep their declarations grouped by functionality and comment them well.
- n With the exception of globals that should not be passed, procedures and functions should only operate on objects that are passed to them. Global variables that are used in routines should be identified in the general comment area at the beginning of the routine.
- n Likewise, try to put as much logic and as many user interface objects in Dialog Boxes as possible. This will help segment your application's complexity and minimize its run-time overhead.

