A

VBA Programming Standards

PROGRAMMING STANDARDS ARE USED TO HELP PROGRAMMERS create a consistent structure, coding style, and logic of an application. Standards help a programmer create code that's easy to read, unambiguous, and easily maintained by other programmers in the organization. The standards included here are the ones currently used by Paul D. Sheriff and Associates, Inc. for the applications developed for its clients. Feel free to modify these to suit your organization's needs. This document is also included on the companion Web site for this book.

Reasons for Naming Conventions

There are many reasons programming standards are created. Most mainframe shops have had programming standards for years. Unfortunately, most PC programmers have forgotten or have never worked in a formal programming shop, and they often overlook this important step in application development.

Creating a programming standard does not limit your creativity, as most programmers seem to think. Instead, it helps you focus your creativity where it's really needed. You can concentrate on the program itself instead of having to always think about what name to give a variable. Think about the Windows environment (or even the Macintosh)—every program written for Windows has a consistent look and feel. This is why users like using Windows programs, because they don't have to learn everything about how a new program works—they already know how to use most of the new program's features. By using standards in your programming, you can also keep the programmer "look and feel" consistent. This means you spend less time figuring out what the variables are or how many indents a programmer used, and you can focus more on the logic of the program.

The use of standards can lead to reduced maintenance costs due to a consistent look and feel. This means you can move from one project to another very easily—even one someone else wrote—and immediately read and understand the code.

Environment Options

To start, you should make sure every programmer's Visual Basic environment is consistent, or if you are working alone, make sure that you at least set the following defaults. Visual Basic lets you set defaults for the development environment. Table A.1 shows the recommended values for some of those options. To set these values, select Tools, Options from the Visual Basic design menu.

Table A.1 Options to Set in Your Visual Basic Environment

Editor Tab

Option	Value	Description
Tab Stops	`3`	Can be anything you want. I prefer `3`.
Require Variable Declaration	Checked	Requires you to declare all variables prior to usage by adding an Option Explicit statement to the beginning of every module.

General Tab

Option	Value	Description
Grid Units, Height Points	`60`	Can be anything but should be a multiple of `300` because that's the minimum height of a combo box. You should strive to make all controls a consistent size, and `300` is a good default height.
Grid Units, Width Points	`60`	(See preceding description.)

Environment Tab

Option	Value	Description
When a Program Starts…Prompt to Save Changes	Checked	Prompts you to save changes prior to testing the application. This is done to ensure no work is lost due to general protection faults (GPF) or other unexpected errors.

Option Explicit

Declaring all variables in your code saves programming time by reducing the number of bugs caused by typos and undeclared variables. For example, the following variables all look the same: `aUserNameTmp`, `sUserNameTmp`, and `sUserNameTemp`. If you don't set the `Option Explicit` statement at the top of every module, these different variable names could all have potentially different values. At runtime, you check the value of the `sUserNameTmp` variable when you really wanted to be checking the value of the variable `sUserNameTemp`. If you set the `Option Explicit` statement at the top of the module, the compiler will inform you of the typo. If you don't set the `Option Explicit`, the runtime engine will simply create another variable on-the-fly and assign a zero value to that variable. This can lead to a bug that's very hard to track down.

Custom Control Naming Conventions

An important part of any Visual Basic program is the controls placed on a form. All similar custom controls should have a common prefix. For example, text boxes should start with *txt*, and labels should start with *lbl*. Table A.2 shows the recommended control name prefixes for most of the controls in the Visual Basic environment. These prefixes are consistent with those documented in the *Visual Basic Programmers Guide*.

Table A.2 Custom Control Naming Conventions

Prefix	Control Type	Example
ani	Animation button	aniMailBox
cbo	Combo box	cboEnglish
chk	Check box	chkReadOnly
clp	Picture clip	clpToolbar
com	Communications	comFax
dat	Data control	datBiblio
dbc	Data-aware combo box	dbcStates
dbg	Data-aware grid	dbgLineItems
dbl	Data-aware list box	dblCustomers
dir	Directory list box	dirSource
dlg	Common dialog control	dlgFileOpen
drv	Drive list box	drvTarget
fil	File list box	filSource
fra	Frame	fraLanguage
frm	Form	frmEntry
gau	Gauge	gauStatus
gpb	Group pushbutton	gpbChannel
gra	Graph	graRevenue
grd	Grid	grdPrices
hsb	Horizontal scrollbar	hsbVolume
img	Image	imgIcon
iml	ImageList control	imlPictures

Prefix	Control Type	Example
key	Keyboard key state	keyCaps
lbl	Label	lblHelpMessage
lin	Line	linVertical
lst	List box	lstPolicyCodes
lsv	List view	lsvItems
mci	Multimedia control	mciVideo
mdi	MDI child form	mdiNote
mnu	Menu	mnuFileOpen
mpm	MAPI message	mpmSentMessage
mps	MAPI session	mpsSession
msk	Masked edit control	mskPhone
ole	OLE control	oleWorksheet
out	Outline control	outOrgChart
pbr	Progress bar	pbrCompleted
pic	Picture	picVGA
pnl	3D panel	pnlStatus
rpt	Report control	rptQtr1Earnings
rtb	Rich text box	rtbNotes
sbr	Status bar	sbrMDI
shp	Shape controls	shpCircle
sld	Slider	sldScore
spn	Spin control	spnPages
tab	Tab control	tabForm
tbr	Toolbar	tbrTools
tbs	TabStrip control	tbsForm
tmr	Timer	tmrAlarm
trv	Tree view	trvCustomers
txt	Text box	txtLastName
vsb	Vertical scrollbar	vsbRate

If you take care to name your controls with these common prefixes, when you're using a particular control in your source code, you'll be able to identify its type without having to refer back to the form. This can save you time when coding or maintaining your application.

Menu Naming Conventions

Just like custom controls, all the various menus you create should also use appropriate prefixes (see Table A.3). The prefix *mnu* should be used on all menus. This prefix should be followed by the caption of the menu. For each pull-down menu under the main menu item, use the first letter of top-level menu.

Table A.3 Naming Menus

Menu Caption Sequence	Menu Handler Name
`File`	mnuFile
`File.New`	mnuFNew
`File.Open`	mnuFOpen
`File.Send`	mnuFSend
`File.Send.Fax`	mnuFSFax
`File.Send.Email`	mnuFSEmail
`Help`	mnuHelp
`Help.About`	mnuHAbout

When this convention is used, all members of a particular menu group are listed next to each other in the Object drop-down list boxes (in the Code and Property windows). In addition, the menu control names clearly document the menu items to which they're attached.

Naming Conventions for Other Controls

For new controls not listed so far, try to come up with unique three-character prefixes. Note, however, that it's more important to be clear than to stick to a three-character convention.

For derivative controls, such as an enhanced list box, extend the preceding prefixes so that there's no confusion over which control is really being used. A lowercase abbreviation for the manufacturer could be added to the prefix. For example, a Crescent QuickPak Professional text box control could be named ctxtFirstName.

Third-party Controls

Each third-party control used in an application should be listed in the application's overview comment section, providing the prefix used for the control, the full name of the control, and the name of the software vendor (see Table A.4).

Table A.4 Third-party Control Naming Standards

Prefix	Control Type	Vendor
ctxt	Text box	Crescent
sscbo	Sheridan combo box	Sheridan

Variable/Object Naming

Variables are the building blocks of every application you create. As such, naming your variables is very important. When creating names for your variables, you need to make the purpose of the variable clear. The use of one-letter variable names should be avoided at all costs. This only leads to confusion when you look at the code at a later date.

The following list presents some standards you should follow when creating your variable names:

- Avoid one-letter variables.
- Make all variable names mixed case.
- Each word or abbreviation should be capitalized.
- Do not use the underscore character in a variable name.
- Use Hungarian notation (see the following section).
- Preface each variable name with the type of data it contains.
- Abbreviate variables only when absolutely necessary.

Hungarian Notation

Hungarian notation is a naming scheme for variables developed by a programmer who worked at Microsoft. Originally developed for C language programmers, it has now been successfully applied to many other languages. The use of Hungarian notation in a program gives a programmer a lot of information about a variable just by looking at the name. A variable that has been named using the Hungarian notation scheme tells a programmer the variable's scope (global, module, or local), what data type it contains (`Integer`, `Long`, and so on), and, of course, the purpose of that variable.

Hungarian notation is as valuable in Visual Basic as it is in the C language. Hungarian notation is widely used by Windows C language programmers and constantly referenced in Microsoft product documentation and in programming books. With all this attention on Hungarian notation, it seems this is a worthy standard that has stood the test of time, and it will probably be around for many years to come.

Visual Basic supplies type suffixes (`%`, `&`, `$`, and so on) to indicate a variable's data type. However, these suffixes are very cryptic to a new user of Visual Basic and also make source code difficult to read. Additionally, they do not explain what the variable will be used for or how it can be accessed. Table A.5 lists some examples of how you should create your variable names without using the suffixes.

Table A.5 Hungarian Notation Examples

Variable Name	Description
`intSend`	Represents a count of the number of messages sent
`boolSend`	A Boolean flag that defines the success of the last `Send` operation
`hWnd`	A handle to a window

In Table A.5, each of the variable names tell a programmer something very different because of the prefix added to each. This information is lost when the variable name is reduced to `Send&` or `Send%`.

NOTE
Do *not* use Hungarian notation on any properties or classes that will be exposed through an Automation Server to other applications. Keep these properties and class names generic so users can understand them.

Variable Data Type Prefixes

Table A.6 defines the standard variable data types in Visual Basic. These prefixes should be used when naming the variables in your program. Avoid the use of the Visual Basic suffixes (`%`, `&`, `#`, and so on) at all costs.

Table A.6 Variable Type Prefixes

Prefix	Data Type	Storage	Example
bool	`Boolean`	Two bytes	`boolPerform`
byt	`Byte`	One byte	`byteArray`
cur	`Currency`	Eight bytes	`curSales`
int	`Integer`	Two bytes	`intLoop`
dbl	`Double`	Eight bytes	`dblValue`
dt	`Date` and `Time`	Eight bytes	`dtEntered`
lng	`Long`	Four bytes	`lngEnv`
sgl	`Single`	Four bytes	`sglValue`
str	`String`	One byte per char	`strName`
vnt	`Variant`	Sixteen bytes plus one byte for each character in a string type	`vntAnything`
a	`Array`	Depends on size	`astrName`
typ	User-defined type	Depends on size	`typCustomer`
o or obj	`Objects`	Any object	`OConn`

Scope Prefixes

A variables scope refers to the locations within your application that a particular variable may be read or modified. For example, a local variable is one declared within a `Sub...End Sub` or `Function...End Function`. This variable may only be read or modified while code is executing within the `Sub...End Sub` statements. A module-level variable, on the other hand, may be read and modified from any procedure or function within the same module. Global or public variables may be referenced from any procedure or function within the same project. Table A.7 lists the common scope prefixes you should use.

Table A.7 Scope Prefixes

Prefix	Description
g	Global or public
m	Module- or form-level variable/object
st	Static variable

Local variables should not have a prefix. This will distinguish them as being local and not having a scope outside of the current procedure. Here are some examples of declaring variables:

```
Global gintLoop As Integer
Dim mstrName As String
Private pboolOpen As Boolean
```

Handles

When programming Windows using the Windows API, you're frequently dealing with handles to windows. You'll also find handles when dealing with the ODBC API. As such, you need a special prefix to identify handles. Use the prefix *h* to name a handle variable:

```
Dim hWnd As Long       ' Handle to a Window
Dim hEnv As Long       ' Handle to the ODBC environment space
Dim hdbc As Long       ' Handle to an ODBC data connection
```

User-defined Types

Declare user-defined types with the prefix *typ*. Variables that are dimmed of this particular type should also be referenced with a *typ* prefix. This will help distinguish the setting of elements in a user-defined type from the setting of properties in an object. Here's an example:

```
Type typCustomer
   strName As String
   strState As String * 2
   lngID as Long
End Type
```

When declaring an instance variable of a user-defined type, add a prefix to the variable name to reference the type:

```
Dim typCust as typCustomer
```

Class Naming

When creating your own user-defined classes, you need to be aware of who your target user will be for that class. If you'll be exposing that object and its properties through an Automation Server, you should not use Hungarian notation. Nonprogrammer types will not understand this type of notation.

It's recommended that you create `Property Get/Let/Set` procedures for every variable used within the class you want to give the user access to. Use Hungarian notation on all private data but do not use it on the `Property Get/Let/Set` procedures.

When naming your class modules, keep in mind that you need to give each public class a descriptive name that the user can understand. If you're creating an internal class that's only used within your program, use the prefix *cls* or maybe just *C* (see Table A.8).

Table A.8 Prefixes for Class Modules

Prefix	Description
cls or C	Only used for private classes
(none)	For any public classes that will be exposed through an OLE Server

Naming Constants

Constant names should use uppercase letters with an underscore (_) between the words. This will help distinguish your constants from the built-in constants in VBA. Also, be sure to always type your constants. If you do not type your constants, they will be of the type `Variant`. This means you're using up 16 bytes per constant. This is a big waste and should be avoided. Here are some examples of constant declarations:

```
Public Const TAB_ADDRESS As Integer = 0
Public Const TAB_PHONES As Integer = 1
```

Conditional Compile

For conditional compilation constants, use the prefix *cc*:

```
#If ccDEMO Then
    ' Perform some code here
#End If
```

Variant Data Type

If you know that a variable will always store data of a particular type, Visual Basic can handle that data more efficiently if you declare the variable of that type. For example, if you need to loop through an array, create an index of the type `Integer`, not `Variant`. In other words, avoid the use of `Variant` unless it's absolutely necessary.

Function and Procedure Naming

Here are some general guidelines to follow when naming your functions and procedures:

- Use mixed case, where each word in the procedure is capitalized.
- Preface all functions and procedures with a noun.
- Follow this noun with the action that will be performed on that noun.
- Do *not* use underscores in your function names—this makes it hard to determine which procedures are yours and which are Visual Basic event procedures.

Here are some examples of procedure names that should not be used because the verb comes first:

- `DisplayForm`
- `InitializeForm`
- `LoadStates`

Here are some examples of better procedure names:

- `FormShow`
- `FormInit`
- `cboStateLoad`

The difference is the noun, or *object*, is placed first. There are fewer ways to describe an object than there are for a verb to describe what will be done. For example, the verb *display* can be expressed a couple other ways, such as *show*, *draw*, or even *print*. However, a *form* is always a *form*, there's no other descriptor for it.

In object-oriented languages and even in VB, the object name is placed first and then the method or action you want to perform on that object. Objects are always nouns, and actions are verbs. You should follow this coding style when creating names for your procedures.

An additional benefit of this coding style is that when you use a cross-referencing tool or are looking in the Proc combo box, you'll see all functions that operate on a particular object grouped in one place.

Error Labels

Labels in a procedure should only be used for error handling. Otherwise, too many programmers would be tempted to use the `GoTo` statement. A `GoTo` statement can lead to many problems. Name your labels with the name of the procedure followed by the suffix *_EH*. Here's an example:

```
Private Sub cmdSave_Click()
   On Error GoTo cmdSave_Click_EH
   ...
   ...
cmdSave_Click_EH:
   ' Error handling here
End Sub
```

Commenting Your Code

Each procedure and function should 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 yet, erroneous comments. The code itself and any necessary inline or local comments should describe the implementation.

Parameters passed to a routine should be described when their function is not obvious and when the routine expects the parameters to be in a specific range. 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.

Each nontrivial variable declaration should include an inline comment describing the use of the variable being declared.

Variables, controls, and routines should be named clearly enough that inline commenting is only needed for complex or nonintuitive implementation details.

An overview description of the application, enumerating primary data objects, routines, algorithms, dialog boxes, 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 the files the Project window doesn't list, such as initialization (INI) or database files.

Formatting Your Code

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. Standard tab-based, block-nesting indentations should be three spaces. More than four spaces is unnecessary and can cause statements to be hidden or accidentally truncated. Less than two spaces does not sufficiently show logic nesting. The highest-level statements that follow the overview comment should be indented one tab, with each nested block indented an additional tab. Here's an example:

```
'*******************************************************
'* Procedure: FormCenter() - Center Form on the Screen
'* Author    : Paul D. Sheriff
'* Date      : Oct. 22, 1998
'* Revised   : Oct. 22, 1998
'*
'* Description:
'* Center Form on the Screen or MDI Main Form
'*
'* Syntax: Call FormCenter(frmFormName)
'*
'* Parameters:
'* <frm> frmFormName    => Form To Center
'*******************************************************
Sub FormCenter (frmFormName As Form)
   frmFormName.Left = (Screen.Width - frmFormName.Width) / 2
   frmFormName.Top = (Screen.Height - frmFormName.Height) / 2
End Sub
```

Operators

Always use the & operator when concatenating strings and the + operator when working with numerical values. Using + may cause problems when operating on two variants. Here's an example:

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

As you can see from this source code, the + and & operators create two different results when using the `Variant` data type. This is a good reason to always use the appropriate operator and avoid using a `Variant` data type.

Scope

Variables should always be defined with the smallest scope possible. Global variables can create enormously complex state machines and make the logic of an application extremely difficult to understand. Global variables also make the reuse and maintenance of your code much more difficult. Variables in Visual Basic can have the following scope, as detailed in Table A.9.

Table A.9 Scope of Variables

Scope	Variable Declared In:	Visibility
Procedure level	Event procedure, sub, function, or method	Visible in the procedure in which it's declared
Form level, Module level	Declarations section of a form or module (FRM, BAS, or CLS)	Visible in every procedure in the form or code module
Global/public level	Declarations section of a code module (BAS using `Global` keyword)	Always visible

In a Visual Basic application, only use global variables when there's no other convenient way to share data between forms. You should use a property on a form instead of using global variables.

Global Variables

If you must use global variables, it's good practice to declare all of them in a single module and group them by function. Give the module a meaningful name that indicates its purpose, such as GLOBAL.BAS.

With the exception of global variables, 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. In addition, you should pass arguments to subs and functions using `ByVal`, unless you explicitly want to change the value of the passed argument.

Write modular code whenever possible. For example, if your application displays a dialog box, put all the controls and code required to perform the dialog box's task in a single form. This helps keep the application's code organized into useful components and minimizes its runtime overhead.

Summary

Programming standards can make the job of the maintenance programmer much easier. Because 90 percent of the time you'll be the maintenance programmer, you should try to follow some programming standards when designing your applications. Programming standards are absolutely essential in a multiprogrammer shop.