Visual Basic™ 32-bit Application Development Standards and Guidelines

Author: Erica Wieland
FemmeTech Inc.
P.O. Box 307
Smithfield, VA 23431-0307
EHWieland@FemmeTech.com

Revised: 1/4/2002

Table of Contents

Visual Basic 32-bit Application Development Standards and Guidelines

1. INTRODUCTION

In my search for guidance on programming standards, I found many "parts" of standards. Naming conventions, best programming practices, commenting guidelines. But I did not find a single, comprehensive source of Visual Basic standards and guidelines that would help me in my day-to-day work. This document is an attempt to pull together the myriad sources of information that I found into a single document that covers every area I found a need for (and a few I wasn't sure I needed). In some cases, the listed standards were lifted almost verbatim from various reference documents (see Acknowledgements). In other cases, bits and pieces from different sources were put together to form a single, comprehensive standard. When sources were in conflict, I chose the one that made the most sense to me. As such, this document is and will remain a "living" document, subject to revisions, additions, and deletions, as I learn more and receive feedback. Because of the disparate sources used in the development of these guidelines, some contradictions may exist. Feedback on any contradictions would be appreciated, so that they can be resolved.

These coding standards and guidelines are intended to provide a framework for development of 32-bit applications for the various Microsoft Windows operating systems using Microsoft's Visual Basic. Specific applications may require additions to these guidelines for application unique requirements. These additions should be documented and maintained during the life cycle of the application.

A comprehensive coding standard encompasses all aspects of code construction and, while developers should exercise prudence in its implementation, it should be closely followed. Completed source code should reflect a harmonized style, as if a single developer wrote the code in one session.

2. ACKNOWLEDGEMENTS

This document owes much to the Application Development Standards and Guidelines developed by IHS Professional Services, and kindly provided on their developer's FTP site at ftp.mindspring.com/users/paquette**.** Other sources of inspiration and guidance are the Sample Development Standards provided by Deborah Kurata of InStep Technologies, Inc. on her website at http://www.insteptech.com/, and Steve McConnell's wonderful book, *Code Complete*, which first convinced me that I needed development standards. Various articles, whitepapers and other documentation available through the Microsoft Developer Network http://www.msdn.microsoft.com were also heavily used in the preparation of these coding standards.

3. REFERENCES

All developers are encouraged to read *Code Complete, A Practical Handbook of Software Construction* by Steve McConnell. The sections on commenting code and code reviews are particularly important.

The *Visual Basic Programmer's Guide to the Win32 API,* by Daniel Appleman, is an excellent resource. Developers should consider this book as part of their Visual Basic documentation.

4. GENERAL CONVENTIONS

4.1. Naming

4.1.1. Purpose

The intent of these naming conventions is to permit any developer to understand the important characteristics of any given object without having to search through code to ascertain that information. Each class of objects uses specific conventions to provide this information.

4.1.2. Visual Basic Intrinsic Naming Rules

4.1.2.1. Names must begin with a letter

4.1.2.2. Names must contain only letters and/or numbers

4.1.2.3. Names may contain the underscore (_) character but not spaces or other punctuation marks

4.1.2.4. Names can be as long as 40 characters

4.1.2.5. Visual Basic reserved words may not be used as names.

4.1.3. General Naming Guidelines

4.1.3.1. Each name will use a predefined prefix to identify the object or variable type. These prefixes are defined in the sections that follow.

4.1.3.2. The first character following the prefix in a name will be an upper case letter.

4.1.3.3. The use of upper and lower case letters, numbers, and underscores in names are encouraged to improve readability.

4.1.3.4. While names may be up to 40 characters in length, developers are encouraged to use the minimum length possible without sacrificing

readability. Name lengths in the range of 9 to 15 characters are considered optimal.

4.1.3.5. Names should be chosen that are clear and unambiguous.

4.1.3.6. Names should reflect the real world nature of the object rather than use computer or data processing terms.

4.1.3.7. Avoid homonyms to prevent confusion during code reviews, such as write and right.

4.1.3.8. Minimize the use of abbreviations. If abbreviations are used, be consistent in their use. An abbreviation should have only one meaning and likewise, each abbreviated word should have only one abbreviation. For example, if using min to abbreviate minimum, do so everywhere and do not later use it to abbreviate minute.

4.1.3.9. Avoid reusing names for different elements, such as a routine called ProcessSales() and a variable called iProcessSales.

4.1.3.10. When naming elements, avoid using commonly misspelled words. Also, be aware of differences that exist between American and British English, such as color/colour and check/cheque.

4.2. Coding

4.2.1. General Coding Considerations

4.2.1.1. Always code for clarity, not efficiency.

4.2.1.2. Choose variable and function names carefully.

4.2.1.3. Write your code for the reader.

4.2.1.4. Size and speed of code, while important, are secondary to readability and, more importantly, maintainability

4.2.2. Required Visual Basic Environment Options

The following options should always be set in the IDE. All other IDE environment options may be set at the developer's discretion:

- Auto Syntax Check Checked
- Require Variable Declaration Checked

- Auto Indent Checked
- Tab Width 4

Save Files as ASCII Text. Save form (.FRM) and module (.BAS) files as ASCII text to facilitate the use of version control systems and minimize the damage that can be caused by disk corruption. To have Visual Basic always save files as ASCII text, from the Environment Options dialog, set the Default Save As Format option to Text. In addition, this allows you to:

- use your own editor
- use automated tools, such as grep
- create code generation or CASE tools for Visual Basic
- perform external analysis of your Visual Basic code

4.2.3. Path Names

Path names will not be hard-coded into any application, unless required by the customer. Provisions must be made that permit the application to ascertain or permit the user to select any required file path information. All path names should use the Universal Naming Convention (UNC) (\\server\share\directory) for shared drives. Mapped drive letters should be avoided unless provided as input from the user. User documents and user output files should be placed in the My Documents directory by default. Placing user output files in a subdirectory under My Documents is acceptable.

4.2.4. Global Modules

While the majority of modules in an application will be class modules, there will be certain standard modules, described below, for each application. The number of global modules will be based on the needs of the application, but should be minimized in favor of class modules.

<EXENAME>.BAS

This module, named as the application EXE, will contain an overview description of the application, enumerating primary data objects, routines, algorithms, dialogs, database and file system dependencies, and so on. It will also contain all documentation for special naming conventions including application specific contexts. In addition, it will contain any application specific public constants.

BEGINEND.BAS

This module will contain the `Sub Main()` and `Sub ExitApp()` procedures along with related startup and shutdown procedures. All applications will have their startup in `Sub Main()`. All applications will have a single exit point in `Sub`

`ExitApp()` containing all necessary housekeeping code required for a normal application shutdown. All application exit points must call this procedure.

4.2.5. Modification of Existing Source Code

When modifying an existing application, all new code should follow these coding standards. Existing code should be changed to be in accordance with these standards during modification if the time required to do the modifications is not excessive, and if the modifications do not have a high likelihood of introducing bugs into the application.

4.3. Commenting

4.3.1. Purpose

Code, when well written, should be self-documenting; however, self-documented code cannot possibly describe the developer's intent or explain an algorithm or section of logic! Therefore, comments are required to ensure that the code can be maintained by communicating this type of information. Comments must communicate information and not what code is coming next. Developers should document their code liberally with in-line comments. Comments should allow a different developer to understand the purpose and function of the code including its relationship to other code modules.

Make it a practice to write comments at the same time that (or earlier than) you write your code. Some developers write the comments for all of their procedures before they write a single line of code. It can be very effective to design procedures using only comments to describe what the code will do. This is a way to sketch out a framework for a procedure, or several related procedures, without getting bogged down in the details of writing the code itself. Later, when you write the code to implement the framework, your original high-level descriptions can be effective comments. Whatever technique you use, always enter or revise your comments as soon as you write the code. Never "save it for later," because there will often never be time to do it later, or if there is, you will not understand the code as well when you come back to it at some other time.

Refer to the section on commenting in *Code Complete, A Practical Handbook of Software Construction* by Steve McConnell. This section gives practical guidance on how to comment code.

4.3.2. General Commenting Guidelines

4.3.2.1. Comment as you code, because most likely there won't be time to do it later. Also, should you get a chance to revisit code you've written, that which is obvious today probably won't be obvious six weeks from now.

4.3.2.2. When modifying code, always keep the commenting around it up to date.

4.3.2.3. Avoid using clutter comments, such as an entire line of asterisks. Instead, use white space to separate comments from code.

4.3.2.4. Avoid surrounding a block comment with a typographical frame. It may look attractive, but it is difficult to maintain.

4.3.2.5. Prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work.

4.3.2.6. If you need comments to explain a complex section of code, examine the code to determine if you should rewrite it. If possible, do not document bad code—rewrite it. Although performance should not typically be sacrificed to make the code simpler for human consumption, a balance must be maintained between performance and maintainability.

4.3.2.7. Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.

4.3.2.8. Avoid the use of superfluous or inappropriate comments, such as humorous sidebar remarks.

4.3.2.9. Use comments to explain the intent of the code. They should not serve as inline translations of the code.

4.3.2.10. Comment anything that is not readily obvious in the code.

4.3.2.11. To prevent recurring problems, always use comments on bug fixes and work-around code, especially in a team environment.

4.3.2.12. Use comments on code that consists of loops and logic branches. These are key areas that will assist the reader when reading source code.

4.3.2.13. Throughout the application, construct comments using a uniform style, with consistent punctuation and structure.

4.3.2.14. Despite the availability of external documentation, source code listings should be able to stand on their own because hard-copy documentation can be misplaced.

5. FORMS, MODULES & CLASSES

5.1. Naming

Class, module, user control, and user document names (the Name property, not the file name) will use a single upper case letter prefix. Forms will use the lower-case three-letter abbreviation frm, unless they are being used as a formal class, in which case they will use the upper-case letter F.

Object Type	Prefix
Form	frm
Form (as Class)	F
Class	C
Interface	I
M	Standard Module

Examples:

CStringUtil String class categorized as "Util"

CError	Error class
IExplorer	Explorer interface class

5.2. Coding

5.2.1. Use of Option Explicit

Option Explicit must be used in every form and module. It must be the first code line following the form/module header, before the declarations sections. Use of Option Explicit will require the developer to explicitly declare all variables thereby eliminating errors introduced by using misspelled variables.

5.2.2. Indentation

Code in all procedures will be indented one (1) tab stop. Code inside programming constructs such as `If...EndIf`, `Select Case...End Select`, `For...Next`, and `Do...Loop` will be indented one (1) additional tab stop. Nested constructs will be indented an additional tab stop for each level of nesting.

5.2.3. White Space

Use white space and individual or double blank lines to indicate logical program sections. Blank lines should separate programming constructs such as `If...EndIf` and `For...Next` from surrounding code. Blank lines should also be used to

improve readability of the code such as in large `Select Case...End Select` constructs.

5.2.4. Multiple Statements

Multiple statements should not appear on a single line. Even a simple If, Then should be put on three lines for clarity.

5.2.5. Use of the Line Continuation Character (_)

The line continuation character should be used to break long statements into multiple lines. In general, a single line should be no longer than what will appear in the code window in the IDE.

5.2.6. Use of Case Statements

All Case statements must include a Case Else to ensure that there is a default case. The Case Else may display an error message or log an error, if appropriate.

5.2.7. Use of "Magic Numbers" or "Magic Literals"

Magic numbers (i.e., hard-coded numbers in code) and magic literals (i.e., hard-coded strings) should be avoided. Named constants and enumerations, where appropriate, should be used instead. Hard-coded 1s or 0s may be used to increment, decrement, and start loops, although a descriptive variable is preferable.

5.2.8. Use of "&" and "+" for Concatenation

Use the concatenation operator ("&") rather than the plus sign ("+") when concatenating strings. Use the plus sign ("+") when working with numeric values.

5.2.9. Use of An Object's Default Property

When referring to an object without referencing a property, the default property of the object is being invoked. Using an object's default property makes the code less maintainable. Therefore, developers are to always use an object's specific property rather than its default property.

5.2.10. Use of GoTo

The use of `GoTo` can make code more difficult to follow and more difficult to maintain. Therefore, `GoTo` should only be used as part of the On Error statement and to enforce a single exit point for each procedure.

5.2.11. Use of IIf

In general, the IIf (immediate if) function should not be used. If it is to be used, it should be thoroughly tested to ensure that the tests are being evaluated as anticipated.

5.2.12. Single Exit Point

All procedures, regardless of their type will have one and only one exit point. This is one of the very few times that `GoTo` should be used. If a procedure has multiple conditions for an exit, each condition should end with a `GoTo` statement. The `GoTo` label for this case should be '`CleanExit`'. The exit label should be positioned just before the section of housekeeping code prior to the exit.

5.2.13. Class Initialization

The Visual Basic form and class modules have a Class object having two events, Initialize and Terminate. These events are fired only once each at the instantiation and termination of the object represented by the class. Unfortunately, the Initialize event does not have any parameters, therefore, it can not be used to initialize the object using passed-in data that is available only at run time. To get around this limitation, any class having the requirement to be initialized using data not available to the object will have a method named InitializeObject. This method may have any parameters necessary to complete initialization of the object. It is to be invoked immediately following the creation of the object.

5.2.14. Error Handling

Procedures will use an error trap for expected and unexpected run time errors. Developers may exercise their judgment and decide that a general or event procedure is simple enough that error handling is not necessary; however, all procedures in ActiveX components must use error traps. Handling of those errors is left to the developer's judgement.

Our general error handling philosophy will be to handle errors at the lowest level possible without user intervention. Failing that, all errors will be passed up to the user interface and displayed for the user. As each routine is written, some time should be spent finding the most common errors and writing code to handle them properly in that routine.

Every effort will be made to handle errors within a procedure. When this is not possible, error conditions will be raised back up to the calling procedure using appropriate implementations of the Err.Raise method available in a standard ActiveX component. Refer to the Programmer's Guide in the VB documentation for a discussion on error handling with ActiveX objects.

`GoTo` should be used as a clause in the `On Error` statement. When used, the `On Error` statement should be the first line of code following the variable declarations in a procedure. The `GoTo` label should be '`ErrorHandler`'.

When an error condition is detected within the code (not from a runtime error), a different `GoTo` label made up of the procedure name with '`Err`' appended to the

end may be used if the error handling requirements are different between detected error conditions and runtime errors.

The error numbers should be the VB errors when appropriate. For errors that are not VB errors, such as validation or business rule violations, an error number should be defined. The error number must be generated greater than vbObjectError + 512. The errors that can be generated by a class should be exposed to the other parts of the application using an ENUM statement.

5.3. Commenting

5.3.1. Module, form and class headers

Will be placed as the first entry in the declarations section of the file. `Option Explicit` must immediately follow the header. The declaration sections follow the `Option Explicit` statement. Declarations will be in the order shown in the sample file header. All public declarations should precede file level declarations in each section where such distinctions can be made.

```
‘Created:            03/15/99
‘Author:       Joe Programmer
‘Description:  Description of file; Include sufficient
‘                 info to clearly describe the component
‘Dependencies:    List any other dependent components
‘Issues:          List known issues, problems, etc.
‘
Option Explicit

‘============================================
‘Code Maintenance Log
‘============================================

‘============================================
‘Implements Declaration Section
‘============================================

‘============================================
‘Type Definition Declaration Section
‘============================================

‘============================================
‘Enumeration Declaration Section
‘============================================

‘============================================
‘API/DLL Procedure Declaration Section
‘============================================

‘============================================
‘Constant Declaration Section
```

```
'=========================================

'=========================================
'Variable Declaration Section
'=========================================
```

5.3.2. Code Maintenance Comments

Code maintenance comments will not be used until the 'code complete' milestone. After that point, the following commenting will be used.

A code maintenance section will be appended to the file header under the Code Maintenance Log section to identify and summarize the changes made to the file. Use the following format:

Place this comment block under the Code Maintenance Log section in the file header for each maintenance action. Use AFTER the "code complete" milestone.

```
'
'Developer:       Joe Programmer
'Change Date:     06/05/99
'Reference:       Put in, as appropriate, any problem
'                 report or bug references.  If the
'                 change is an enhancement, indicate it
'                 as such.
'Description:     Put in a brief description of the
'                 change(s) made.
'
```

If the version control tools being used provide the capability to automatically add this information, use the tool's format rather than the one above.

In the procedure where the changes are made, the developer must add an entry to the history section of the procedure header to include the developer's name, the date of the change, and a brief comment describing the changes including any bug numbers. Refer to the example below:

```
'-------------------------------------------------------------
'Date        Developer           Comments
'03/15/99    J. Programmer       Initial creation
'05/24/99    J. Programmer       Bug 1234; Changed column names
'                                to new naming standards
'
```

At the actual location in the code where the changes are made, the developer is to add a comment block showing "**", their initials, the date, and the bug number or other reference in the first comment line. Use subsequent comment lines to describe the actual changes. . Refer to the example below:

```
'JP 05/24/99 – Bug 1234
'Change the query columns in the SQL statement
'to correspond to the new naming standards.
```

6. PROCEDURES AND FUNCTIONS

6.1. Naming

6.1.1. General

Public and private subroutines, functions, and methods will generally follow a <verb><object> syntax, such as OpenFile or WriteReport, using mixed case.

6.1.2. Functions

6.1.2.1. Public and private function procedures will use the <verb><object> naming convention.

6.1.2.2. Functions will include a three-letter lower case prefix to indicate the data type that the function returns. The function data type prefix will use the same letters as the variable data type prefix or context tag. Public Function procedures used as class methods will NOT use the datatype prefixes.

6.1.2.3. When naming functions, include a description of the value being returned, such as `GetCurrentWindowName()`.

6.1.3. Property Procedures and Methods

Class property procedures and public Sub and Function procedures used as class methods will NOT use the datatype prefixes. In addition, parameters of public property or public method procedures will NOT use the datatype prefix. In some cases, a single <verb> is an acceptable public Sub or Function procedure name, such as Move or Add. Names will follow standard Windows conventions for properties and methods. All method names will follow the <verb><object> naming convention.

NOTE: All event procedures are generated by Visual Basic for forms and controls. These event procedures are private to the form containing them and are automatically named based on the form or control name and the event. The above naming conventions do not apply to event procedures. However, the above conventions do apply to events that are created as part of classes by a developer.

6.1.4. Files and Folders

File and folder names, like procedure names, should accurately describe what purpose they serve.

6.2. Coding

6.2.1. Keep It Short

In general, routines should not be longer than one screen page. Routines longer than one screen should be broken down into subroutines.

6.3. Commenting

6.3.1. Procedure Headers

All procedures and functions will have a standard header comment section immediately following the Sub, Function, or Property statement. A variable declaration section will immediately follow the procedure header. Constants will come first in the declarations section followed by variables.

Use for all property, method, general function, and general sub procedures.

```
‘Purpose:   What the routine does (not how)
‘Inputs:    Each non-obvious parameter on a separate line
‘           with in-line comments
‘Assumes:   List of each non-obvious external variable,
‘           control, open file, and so on.
‘Returns:   Explanation of value returned for functions.
‘Effects:   List of each affected external variable, control
‘           file, and so on, and the affect it has (only
‘           if this is not obvious
‘
‘Date        Developer          Comments
‘03/15/99    J. Programmer      Initial creation
‘
‘
‘Local Constant/Variable Declaration Section
‘
```

The header for general procedures will contain a clear description of the procedure and a description of each parameter and any return values. A maintenance line will also be included containing the date of the change activity, the developer’s name, and a brief description of the change activity. During development, only a line for procedure creation will be used. When the application enters the maintenance phase (after “code complete”), the maintenance lines will be included. These headers will be placed in all user defined general procedures.

The header for event procedures will only contain maintenance lines containing the date of the change activity, the developer’s name, and a brief description of the change activity. These headers will be placed in all event procedures that contain code. During development, only a line for procedure creation will be used. When the application enters the maintenance phase (after “code complete”), the maintenance lines will be included. They are NOT to be placed in event procedures that DO NOT contain code.

6.3.2. In-Line Comments

Comment as you code. Make it a policy to change the comments when the purpose of the code changes. In-line comments for describing particular lines of code, or a group of code lines such as loops, will consist of the comment block with no special comment line before or after it. A blank comment line should precede the comment block, and there should be no blank lines between the comment block and the code to which it refers. It will be constructed as follows:

```
'
'Comment goes here.  Use as many lines as necessary,
'being sure to start a new line so the comment is
'visible in a typically sized code window
```

7. CONTROLS & MENUS

7.1. Naming

7.1.1. Controls

The default control names provided automatically by the development environment WILL NOT be used as the control name. Instead, create a name using a prefix to identify the type of control with a name that adequately describes the function or purpose of the control. The prefix for standard Visual Basic controls will consist of three (3) lower case letters followed by the field name.

The following table contains prefixes for the Visual Basic controls.

Prefix	Control	Example
ani	Animated Button	aniPhoneConnect
bed	Pen BEdit	bedName
cbo	Combo Box	cboTypeCode
chk	Check Box	chkReadOnly
clp	Picture Clip	clpToolBar
cmd	Command Button (3D)	cmdOK (cmd3dOK)
com	Communications	comCommPort1
ctl	Control (when type unk)	ctlUpdate
dat	Data Control	datClient
dbc	DBCombo (data-bound)	dbcState
dbg	DBGrid (data-bound)	dbgRegister
dbl	DBList (data-bound)	dblTitle
dir	Directory List Box	dirSource
dlg	Common Dialog	dlgBrowse

Prefix	Control	Example
drv	Drive List Box	drvTarget
fil	File List Box	filSource
fra	Frame (3D)	fraUserOptions (fra3dUserOptions)
gau	Gauge	gauProgress
gpb	Group Push Button	gpbChannel
gra	Graph	graSalesYTD
grd	Grid	grdToDo
hed	Pen HEdit	hedSignature
hsb	Horizontal Scroll Bar	hsbVolume
ili	ImageList Item	iliToolBarButton
ils	ImageList	ilsToolBarImages
img	Image	imgIcon
ink	Pen Ink	inkMap
key	Key State	keyCapsLock
lbl	Label	lblLastName
lin	Line	linDivider
lst	List Box	lstStates
lvi	List View Item	lviUserName
lvw	List View	lvwNames
mci	Multimedia MCI	mciMMControl
mnu	Menu Option	mnuFileExit
mpm	MAPI Message	mpmMAPIMessage
mps	MAPI Session	mpsMAPISession
msk	Masked Edit	mskWorkPhone
ole	OLE Client	oleWordDoc
opt	Option Button (3D)	optPrefix (opt3dPrefix)
out	Outline	outGroupMembers
pic	Picture Box	picToolBar
pnl	3D Panel	pnlStatusBar
prg	ProgressBar	prgPrinting
rpt	Crystal Reports Control	rptMailingLabels
rtf	RichTextBox	rtfNotes
shp	Shape	shpSquare
sld	Slider	sldVolume
spn	Spin Button	spnCopies
sta	StatusBar	staMessage
tab	Tab	tabSettings
tbs	TabStrip	tbsOptions
tlb	ToolBar	tlbMain

Prefix	Control	Example
tmr	Timer	tmrTrigger
tvn	TreeView Node object	tvnPart
tvw	TreeView	tvwPartList
txt	Text Box	txtLastName
upd	UpDown	updVolume
vsb	Vertical Scroll Bar	vsbMouseSensitivity

7.1.2. Custom and Derived Controls

For new controls not listed above, try to come up with a unique three character prefix. However, it is more important to be clear than to stick to three characters.

For derivative controls, such as an enhanced list box, extend the prefixes above so that there is no confusion over which control is really being used. A lower-case abbreviation for the manufacturer would also typically be added to the prefix. For example, a control instance created from the Visual Basic Professional 3D frame could uses a prefix of fra3d to avoid confusion over which control is really being used. A command button from MicroHelp could use cmdm to differentiate it from the standard command button (cmd).

The prefix for third party custom controls will consist of three (3) lower case letters followed by an additional lower case letter or number (chosen as a unique identifier for the set of custom controls) followed by the field name. If the custom control is a replacement for a standard control, the first three letters in the prefix should be the same as the standard control. If the custom control is a unique control, a three-letter prefix must be chosen and used consistently throughout the application.

All new control prefixes should be documented as part of the project documentation on which they are used, including listing them in the <EXENAME>.BAS file. Using the following format:

```
Prefix    Control Type        Vendor
cmdm      Command Button      MicroHelp
```

The following table lists standard third-party control prefixes:

Control Type	**Control name**	**Prefix**	**Vendor**	**Example**	**VBX Filename**
Alarm	Alarm	almm	MicroHelp	almmAlarm	MHTI200.VBX
Animate	Animate	anim	MicroHelp	animAnimate	MHTI200.VBX
Callback	Callback	calm	MicroHelp	calmCallback	MHAD200.VBX
Combo Box	DB_Combo	cbop	Pioneer	cbopComboBox	QEVBDBF.VBX
Combo Box	SSCombo	cbos	Sheridan	cbosComboBox	SS3D2.VBX
Check Box	DB_Check	chkp	Pioneer	chkpCheckBox	QEVBDBF.VBX

Chart	Chart	chtm	MicroHelp	chtmChart	MHGR200.VBX
Clock	Clock	clkm	MicroHelp	clkmClock	MHTI200.VBX
Button	Command Button	cmdm	MicroHelp	cmdmCommandButton	MHEN200.VBX
Button	DB_Command	cmdp	Pioneer	cmdpCommandButton	QEVBDBF.VBX
Button (Group)	Command Button(multiple)	cmgm	MicroHelp	cmgmBtton	MHGR200.VBX
Button	Command Button(icon)	cmim	MicroHelp	cmimCommandButton	MHEN200.VBX
CardDeck	CardDeck	crdm	MicroHelp	crdmCard	MHGR200.VBX
Dice	Dice	dicm	MicroHelp	dicmDice	MHGR200.VBX
List Box (Dir)	SSDir	dirs	Sheridan	dirsDirList	SS3D2.VBX
List Box (Drv)	SSDrive	drvs	Sheridan	drvsDriveList	SS3D2.VBX
List Box (File)	File List	film	MicroHelp	filmFileList	MHEN200.VBX
List Box (File)	SSFile	fils	Sheridan	filsFileList	SS3D2.VBX
Flip	Flip	flpm	MicroHelp	flpmButton	MHEN200.VBX
Scroll Bar	Form Scroll	fsrm	MicroHelp	fsrmFormScroll	???
Gauge	Gauge	gagm	MicroHelp	gagmGauge	MHGR200.VBX
Graph	Graph	gpho	Other	gphoGraph	XYGRAPH.VBX
Grid	Q_Grid	grdp	Pioneer	grdpGrid	QEVBDBF.VBX
Scroll Bar	Horizontal Scroll Bar	hsbm	MicroHelp	hsbmScroll	MHEN200.VBX
Scroll Bar	DB_HScroll	hsbp	Pioneer	hsbpScroll	QEVBDBF.VBX
Graph	Histo	hstm	MicroHelp	hstmHistograph	MHGR200.VBX
Invisible	Invisible	invm	MicroHelp	invmInvisible	MHGR200.VBX
List Box	Icon Tag	itgm	MicroHelp	itgmListBox	MHAD200.VBX
Key State	Key State	kstm	MicroHelp	kstmKeyState	MHTI200.VBX
Label	Label (3d)	lblm	MicroHelp	lblmLabel	MHEN200.VBX
Line	Line	linm	MicroHelp	linmLine	MHGR200.VBX
List Box	DB_List	lstp	Pioneer	lstpListBox	QEVBDBF.VBX
List Box	SSList	lsts	Sheridan	lstsListBox	SS3D2.VBX
MDI Child	MDI Control	mdcm	MicroHelp	mdcmMDIChild	
Menu	SSMenu	mnus	Sheridan	mnusMenu	SS3D3.VBX
Marque	Marque	mrqm	MicroHelp	mrqmMarque	MHTI200.VB
Picture	OddPic	odpm	MicroHelp	odpmPicture	MHGR200.VBX
Picture	Picture	picm	MicroHelp	picmPicture	MHGR200.VBX
Picture	DB_Picture	picp	Pioneer	picpPicture	QEVBDBF.VBX
Property Vwr	Property Viewer	pvrm	MicroHelp	pvrmPropertyViewer	MHPR200.VBX
Option (Group)	DB_RadioGroup	radp	Pioneer	radqRadioGroup	QEVBDBF.VBX

Slider	Slider	sldm	MicroHelp	sldmSlider	MHGR200.VBX
Button (Spin)	Spinner	spnm	MicroHelp	spnmSpinner	MHEN200.VBX
Spreadsheet	Spreadsheet	sprm	MicroHelp	sprmSpreadsheet	MHAD200.VBX
Picture	Stretcher	strm	MicroHelp	strmStretcher	MHAD200.VBX
Screen Saver	Screen Saver	svrm	MicroHelp	svrmSaver	MHTI200.VBX
Switcher	Switcher	swtm	MicroHelp	swtmSwitcher	???
List Box	Tag	tagm	MicroHelp	tagmListBox	MHEN200.VBX
Timer	Timer	tmrm	MicroHelp	tmrmTimer	MHTI200.VBX
ToolBar	ToolBar	tolm	MicroHelp	tolmToolBar	MHAD200.VBX
List Box	Tree	trem	MicroHelp	tremTree	MHEN200.VBX
Input Box	Input (Text)	txtm	MicroHelp	inpmText	MHEN200.VBX
Input Box	DB_Text	txtp	Pioneer	txtpText	QEVBDBF.VBX
Scroll Bar	Vertical Scroll Bar	vsbm	MicroHelp	vsbmScroll	MHEN200.VBX
Scroll Bar	DB_VScroll	vsbp	Pioneer	vsbpScroll	QEVBDBF.VBX

7.1.3. Menus

Due to their unique hierarchical nature, menus have additional naming requirements. Captions on the menu bar must be one word to conform to Windows standards. Captions of menu options and sub menu options should be no longer than two words. Menus should normally have only one level of sub menu options under any given menu option. Menu names consist of the prefix followed by the menu bar caption and the menu option caption. For example, the Exit menu option under the File menu would have a menu name of mnuFileExit. For menu options with two word captions, remove the space between the two words. For example, the Special Paste menu option under the Edit menu would have a menu name of mnuEditSpecialPaste. To distinguish between menu options and sub menu options, insert an underscore character between their names. For example, the Link sub menu option under the Special Paste menu option would have a menu name of mnuEditSpecialPaste_Link.

7.2. Coding

7.2.1. Use of Control Arrays

When feasible, use control arrays instead of separately named controls. An enumeration should be defined to allow the individual controls to be referenced by name, rather than index number. Unless you anticipate needing to reference labels extensively, an enumeration is not required for label control arrays. Remember that control arrays need not contain sequential index number, nor do the index numbers need to start at 0. The benefits of this technique are:

- Speed. When VB creates the controls, it only has to create one of the controls in the control array. This increased the limit on the number of controls you can have on a form and improves performance.
- You get one event procedure for the control array instead of one for each control. This prevents duplicate code.
- It is easier to work through a set of controls that are in an array. You can cycle through them using For Each/Next to move them or set their properties.

8. VARIABLES

8.1. Naming

8.1.1. General

All variables, including arrays, will be given a three-part name, consisting of a lower-case scope indicator (two characters), a lower-case data type indicator (three characters) and a mixed-case variable-length descriptive name. Single letter variable names (such as i, j, and k) will NOT be used.

8.1.2. Scope and Usage Indicators

The scope and usage indicator will consist of one (1) lower case letter prefix followed by an underscore (_) to indicate the scope of the variable. Local (procedure-level) variables will NOT use a scope indicator.

Scope Indicator	Prefix
Public (global)	g_
Class	c_
Module (includes form-level variables)	m_
Static variable	st_
Local	(none)
Parameter (arguments in procedures and functions)	p_

Parameters in public class property or method procedures will NOT use either the scope or data type prefixes.

8.1.3. Data Type Indicators

The scope indicator will be followed by a three-letter prefix that indicates the data type of the variable. The letters that make up the prefixes are:

Data Type Indicator	Prefix
Boolean	Bln
Byte	Byt
Integer	Int
Long	Lng
Single	Sgl
Double	Dbl
Currency	Cur
String (text)	Str
Date(Time)	Dte
Variant	Var
Form (parameter passed "As Form")	Frm
Control (parameter passed "As Control"; use the control prefix for specific controls such as 'txt' for "As Text")	Ctl
Object	Obj
Collection	Col
User Control	Axc
User Document	Axd

Examples:

Form level integer variable m_intObjectCount
Module level single variable m_sglPercentComplete
Form level currency variable m_curInvoiceTotal
Class level long variable c_lngItemCounter
Local integer variable intCounter
Parameter Boolean variable p_blnShowItem
Parameter control p_ctlSourceControl
Parameter textbox p_txtSourceTextBox

8.1.4. Objects

When object variables are declared, using early binding is preferable. For user objects, such as classes, identify and document the three-letter prefix that will be used to identify that object type. Use the "obj" prefix only when declaring a generic, or late-bound, object variable. Use this prefix even when you are creating a late-bound object variable that represents an Office application. For example:

```
Dim objXLApp As Object
Dim objWDDocument As Object
Dim objOLMailItem As Object
```

Object variable prefixes:

ActiveX Data Objects (ADO)

Prefix	Object Type	Example
cnn	Connection	cnnMain
cmd	Command	cmdSalesBRegion
rst	RecordSet	rstCustomers
fld	Field	fldAddress
prm	Parameter	prmAge
pro	Property	Pro
err	Error	errErrors

Data Access Objects (DAO) and Microsoft Access:

Prefix	Object Type	Example
db	ODBC Database	dbAccounts
ds	ODBC Dynaset object	dsSalesByRegion
fdc	Field collection	fdcCustomer
fd	Field object	fdAddress
ix	Index object	ixAge
ixc	Index collection	ixcNewAge
qd	QueryDef object	qdSalesByRegion
qry (suffix)	Query	SalesByRegionQry
ss	Snapshot object	ssForecast
tb	Table object	tbCustomer
td	TableDef object	tdCustomers

Microsoft Office Automation Applications:

Prefix	Object Type	Example
ac	Access	acReport
xl	Excel	xlApp
fp	FrontPage	fpInstance
mso	Office	Mso
bind	OfficeBinder	Bind
ol	Outlook	olNameSpace
pp	PowerPoint	ppPresentation
wd	Word	wdNewDocument
vb	VBA	Vb

8.1.5. Special Context Tags

When a variable is used in a special context, the tag should reflect that context even if it is one of the above variable types. Typical context tags are shown below.

Prefix	Data Type	Context Tags	Example
h	Integer	Handle	hCursor
hwnd	Integer	Window Handle	hwndForm
hdc	Integer	Device Handle	hdcPrinter
rc	Integer	Return Code	rcStatus

Other context tags may be used but must be documented in the <EXENAME>.BAS module.

It is permissible to use just the context tag without a name (i.e., 'rc' rather than 'rcStatus') when the use of the context tag is completely unambiguous. If there is any chance for confusion, the full tag plus name convention will be used.

8.1.6. Use Descriptive Variable Names

Don't use variables names that are totally unrelated to the variables they represent. Variable names like Sally, Bob and BiteMe (or txtSally, intBob, and varBiteMe) make your code difficult to read and maintain, no matter how clever or cute the names are.

8.1.7. Boolean Variables

Boolean variables should be named to reflect the True condition, i.e. blnIsNew, and should be descriptive of what is being tested. Boolean names should not be negative, i.e. blnNotFound. The use of the word Is immediately following the bln prefix is encouraged. This puts the variable name in the form of a positive statement that is clearly True or False.

8.1.8. Loop-Index Variables

Loop-index variables should be given an appropriate, descriptive name wherever possible. For instance, if the loop is going from 1 to the maximum number of records, name the loop counter intMaxRecords. Be sure to choose a variable type that is large enough to hold the largest number that loop should ever encounter; when in doubt, use a long integer. Non-descriptive loop-index variable names (like i, j, or k) will not be used.

8.2. Coding

8.2.1. Declare All Variables

The use of Option Explicit will require all variables to be declared by the compiler. This option should never be removed, even during development. All variables must be declared.

8.2.2. Place Each Variable Declaration on a Separate Line

Multiple variable declarations placed on a single line, e.g., `Dim y, n, b as String`, declares y and n to be the data type Variant, and only b to be a String. Placing each declaration on a single line avoids this problem. If y and n should be variants, that should be made explicit by declaring them on separate lines as Variant data types.

8.2.3. Declare Variables as a Specific Data Type

All form, class, and module level variables, including arrays, will be explicitly declared as a specific data type in the declarations section of the appropriate form, class, or module. Variables should normally be declared as one of the defined data types rather than the variant data type. Unless a particular requirement dictates the use of a variant data type, variant data type variables should be avoided. Parameters (procedure arguments) should be explicitly declared as specific data types in the parameter list of the procedure declaration. Be sure to use the `ByVal` keyword as appropriate. In addition, provide the return data type for all Function and Property Get procedures.

8.2.4. Use Explicit Scope in Variable Declaration

Variables should be declared with the `Private` keyword rather than the `Dim` keyword. Public variables should never be used! If a variable needs to be accessed from outside a class, module, or form, create a property let/get/set for the variable, as required. For routines, you may also select Friend as the scope. This is useful when creating ActiveX components. All variables local to a Sub procedure or Function procedure must be declared explicitly in the beginning of the procedure using the `Dim` or `Static` keywords as appropriate.

8.2.5. Use Variables for One and Only One Purpose

Variables should not be “reused” for different purposes. One variable should be declared for each purpose. In particular, module-level variables should not be declared in order to reuse them in place of local variables in routines.

8.3. Commenting

Variables should be given names that describe what they are being used for. If a variable is unclear, an in-line comment should be included along with the variable declaration describing the purpose of the variable.

9. ENUMERATIONS

9.1. Naming

Enumerations will be declared with the enumeration type in all capital letters and a lower case "e" prefix to distinguish enumerations from user-defined types. The enumeration components follow normal variable conventions except that the enumeration prefix will be used in place of the scope and datatype prefix. The enumeration prefix will be a 2-5 letter prefix for identifying the enumeration, and it will be placed in an in-line comment on the 'Enum' line. The enumeration's prefix also will be used for the data type portion of a variable prefix, as shown below.

Example:

```
Public Enum eRETURNCODE            'rc
    rcSuccessWithInfo = 1
    rcSuccess = 0
    rcFailure = -1
    rcNoDataFound = -7
    rcNoMoreData = -8
End Enum

Public rcReturn As eRETURNCODE
```

10. USER DEFINED TYPES (STRUCTURES)

10.1. Naming

User defined types will be declared with the data type in all capital letters with their components following normal variable conventions except that no scope prefix will be used. A 2-5 letter prefix for identifying the user-defined type will be placed in an in-line comment on the "Type" line. It will be used for the data type portion of the variable prefix.

Example:

```
Type EMPLOYEE                 'emp
    strName As String
    strAddress As String
    strCityStateZip As String
    dteHireDate As Date
    curSalary As Currency
End Type


Private m_empEmployee As EMPLOYEE
```

11. CONSTANTS

11.1. Naming

All non-variable data used in the application must be defined as named constants. Intrinsic VB and VBA constants will be used wherever applicable. These constants can be found in the VB help file. Of particular interest are constants for special characters:

vbCrLf	[Chr$(13) + Chr$(10)]
vbCr	[Chr$(13)]
vbLf	[Chr$(10)]
vbBack	[Chr$(8)]
vbTab	[Chr$(9)]
vbNullChar	[Chr$(0)]

Use these intrinsic constants rather than the equivalent Chr$ functions. Other predefined constants, such as API and type constants, should be used whenever possible.

Application specific constants will be named as appropriate using all uppercase letters, numbers, and an underscore (_) between words. If a group of constants is related, they should have a common prefix of up to five (5) upper case letters and/or numbers followed by an underscore (_) character.

Examples:

APP_NAME
CLR_BLACK
FAILURE

11.2. Coding

Named constants should be used to the maximum extent possible. The use of named constants will greatly improve the readability and maintainability of the source code. More importantly, if future modifications are necessary, the constants can be changed in one location.

The exception to using named constants is to use Enumerations wherever a group of related constants that represent long integer data types are required.

12. MESSAGE BOXES

12.1. Coding

Message boxes will be used throughout applications to provide feedback to the user and to solicit a response from the user for specific purposes. Three types of message boxes, described below, will be used. A fourth type, the query (question) message box, should not be used. It currently exists in Visual Basic for backward compatibility. All message boxes will be invoked in code with a message, the appropriate message box parameters, and the

full application name. The message box parameters, using standard Visual Basic constants, will be concatenated to obtain the required results. The `MsgBox` statement will be used whenever the user just needs to acknowledge the message by clicking on the OK button. When the user is presented with multiple options such as with Yes, No, Cancel buttons, the `MsgBox` function will be used in order to trap for the return value corresponding to the button the user selected.

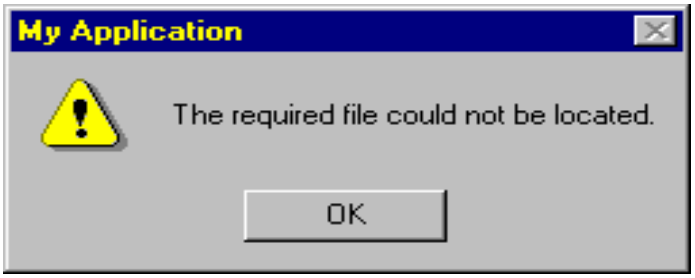
12.1.1. Information Message Box

Information messages should be used to report the results of an action or to provide feedback to the user that cannot otherwise be provided. An example of what an information message box should look like, and the code that invoked it, is shown below:


```
strMsg = “No records were found to satisfy the query.”
MsgBox strMsg, vbOKOnly + vbInformation, strAppName
```

12.1.2. Warning Message Box

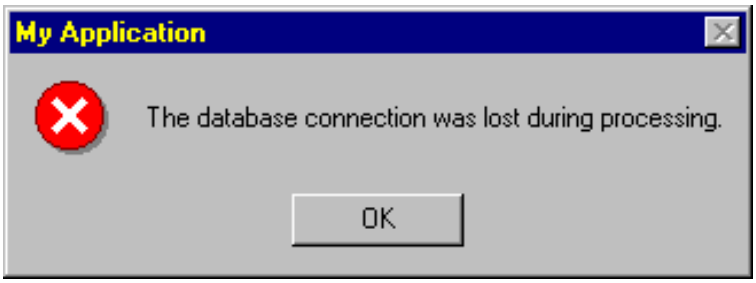
Warning messages should be used to inform the user that a non-critical action cannot be performed or an abnormal result or condition occurred. An example of what a warning message box should look like, and the code that invoked it, is shown below:



```
strMsg = "The required file could not be located."
MsgBox strMsg, vbOKOnly + vbExclamation, strAppName
```

12.1.3. Critical Message Box

Critical messages should be used to inform the user that a critical action cannot be performed or an abnormal result or condition occurred that may cause the application to abort. An example of what a critical message box should look like, and the code that invoked it, is shown below:




```
strMsg = "The database connection was lost during processing."
MsgBox strMsg, vbOKOnly + vbCritical, strAppName
```

13. DATABASES AND STORED PROCEDURES

13.1. Naming

13.1.1. Tables

13.1.1.1. Table names will be in mixed case, beginning with an upper case letter.

13.1.1.2. Table names will be descriptive.

13.1.1.3. Table names will be the singular form of the object they describe, e.g., Part or Supplier.

13.1.2. Fields

13.1.2.1. Field names will be in mixed case, beginning with an upper case letter.

13.1.2.2. Field names will be unique (within any data dictionary in which it appears).

13.1.2.3. Field names will be stated in the singular.

13.1.2.4. Field names will be stated as a descriptive word or phrase.

13.1.2.5. Field names will contain only commonly understood abbreviations.

13.1.2.6. When naming fields, do not repeat the table name; for example, avoid having a field called EmployeeLastName in a table called Employee.

13.1.2.7. Field names will use a capital letter to identify each word of the name-phrase, for example LastName.

13.1.2.8. Do not incorporate the data type in the name of a column. This will reduce the amount of work needed should it become necessary to change the data type later.

13.1.3. SQL Server Stored Procedures

13.1.3.1. Do not prefix stored procedures with sp_, because this prefix is reserved for identifying system-stored procedures.

13.1.3.2. In Transact-SQL, do not prefix variables with @@, which should be reserved for truly global variables such as @@IDENTITY.

13.1.3.3. Variables, functions and stored procedure names should follow the Visual Basic naming conventions outlined in this document to the greatest extent possible.

13.2. Coding

13.2.1. Tables and Fields

13.2.1.1. Tables will contain a primary key of type long integer that is unique and meaningless.

13.2.1.2. When other tables use this key as a foreign key, the field name will be identical to the primary key field name. For example, PartID is the primary key in the Part table. In the SupplierPart table, the foreign key will be PartID.

13.2.1.3. All tables will contain the User ID of the user who last saved the record. The field name for this field will always be UserID.

13.2.1.4. All tables will contain the timestamp of the last update. The field name for this field will always be TimeStamp.

13.2.1.5. When writing SQL statements, use all uppercase for keywords and mixed case for database elements, such as tables, columns, and views.

13.2.1.6. Put each major SQL clause on a separate line so statements are easier to read and edit, for example:

```
SELECT FirstName, LastName
FROM Customers
WHERE State = 'WA'
```

13.2.1.7. Use RETURN statements in stored procedures to help the calling program know whether the procedure worked properly.

13.2.1.8. Never use SELECT *. Always be explicit in which columns to retrieve and retrieve only the columns that are required.

13.2.1.9. Refer to fields implicitly; do not reference fields by their ordinal placement in a Recordset.

13.2.1.10. Use stored procedures in lieu of SQL statements in source code to leverage the performance gains they provide.

13.2.1.11. Use a stored procedure with output parameters instead of single-record SELECT statements when retrieving one row of data.

13.2.1.12. Verify the row count when performing DELETE operations.

13.2.1.13. Perform data validation at the client during data entry. Doing so avoids unnecessary round trips to the database with invalid data.

13.2.1.14. Avoid using functions in WHERE clauses.

13.2.1.15. If possible, specify the primary key in the WHERE clause when updating a single row.

13.2.1.16. When using LIKE, do not begin the string with a wildcard character because SQL Server will not be able to use indexes to search for matching values.

13.2.1.17. Use WITH RECOMPILE in CREATE PROC when a wide variety of arguments are passed, because the plan stored for the procedure might not be optimal for a given set of parameters.

13.2.1.18. Stored procedure execution is faster when you pass parameters by position (the order in which the parameters are declared in the stored procedure) rather than by name.

13.2.1.19. Use triggers only for data integrity enforcement and business rule processing and not to return information.

13.2.1.20. After each data modification statement inside a transaction, check for an error by testing the global variable @@ERROR.

13.2.1.21. Use forward-only/read-only recordsets. To update data, use SQL INSERT and UPDATE statements.

13.2.1.22. Never hold locks pending user input.

13.2.1.23. Use uncorrelated subqueries instead of correlated subqueries. Uncorrelated subqueries are those where the inner SELECT statement does not rely on the outer SELECT statement for information. In uncorrelated subqueries, the inner query is run once instead of being run for each row returned by the outer query.

13.3. Commenting

13.3.1. Tables and Fields

13.3.1.1. Descriptions of the purpose of each table and field will be maintained. Table descriptions should include the source of the data, if applicable, and what the data represents.

13.3.1.2. Field information should include any restrictions on the size, format or valid values in the field.

13.3.2. Stored Procedures

13.3.2.1. Stored procedures should follow the same commenting standards as Visual Basic code, with modifications made only as required by the development environment.

14. ACTIVE SERVER PAGES

14.1. Naming

14.1.1. General

Scripting languages are “loosely typed” and, consequently, all variables used in script have a **Variant** data type. In addition, script is written directly into the HTML code behind a Web page and there are no modules used to contain code as there are in VBA and other strongly typed languages. Finally, scripting languages do not require that you expressly declare variables before you use them.

Given these unique characteristics, does it still make sense to talk about a naming convention in the context of writing script? Absolutely!

The naming conventions and other coding guidelines discussed here apply just as well to script in an HTML page as they do to VBA code in an Office application, or to VB code. The benefits associated with writing reusable, understandable, and maintainable code can be realized whether you are writing script or VB code. In fact, there is probably more work to be done persuading script developers to pay attention to issues of code reuse and maintainability. There are just as many benefits to writing solid script as there are to writing solid code.

14.1.2. Use Visual Basic Naming Standards

Because ASP technology relies on scripting engines to do its work, and because of the loosely typed nature of script, naming conventions have been somewhat fuzzy. In strongly typed languages, variables are declared as their actual type. When using ASP technology, it's common practice to declare your variables in ASP code the way they should be treated, rather than their actual data type. For example, when working with Visual Basic® Scripting Edition (VBScript), you would declare your flag for success as blnIsSuccessful (bln for Boolean) rather than varIsSuccessful (var for Variant), even though all VBScript variables are Variants.

14.1.3. Declare All Variables

In order to prevent errors, all variables will be declared before being used.

14.1.4. Scope and Usage Prefixes for VBScript

Even though script is written directly into the HTML code of a Web page, questions of visibility and lifetime are still important. Variables and constants declared within a procedure are local to that procedure and have a lifetime that lasts only so long as the script within the procedure is executing.

Variables and constants declared in script outside a procedure are visible to any script contained in the current HTML page. These variables have the equivalent of the module-level scope described earlier. Variables and constants declared in VBScript by using the **Public** keyword are visible to all script in the current HTML page and to all script in all other currently loaded pages. For example, if you had an HTML page that contained multiple frames designated by a <FRAMESET> tag pair, a variable or constant declared with the **Public** keyword would be visible to all pages loaded within all the frames specified by the <FRAMESET> tag. To clarify the scope of a given variable, all variables will be declared with a scope prefix, where appropriate:

Prefix	Description
g_	Created in the Global.asa.

m_	Local to the ASP page or in an Include file.
(no prefix)	Non-static variable, prefix local to procedure

14.2. Coding

14.2.1. Microsoft Standards Adopted

Microsoft's ASP Conventions, developed by the IIS Resource Kit Team, Posted March 17, 1998, are adopted by, and considered part of these standards. These conventions are available on the Microsoft Developer Network at http://msdn.microsoft.com/workshop/server/asp/aspconv.asp. These conventions were adapted from Appendix B, "ASP Standards" of Internet Information Server Resource Kit published by Microsoft Press, 1998.

Microsoft's ASP Guidelines, dated December 27, 1999, written by J.D. Meier, which originally appeared in the MSDN Online Voices "Servin' It Up" column are also incorporated as part of this document. The naming conventions outlined in section 14.1 are derived from this document.

14.2.2. Tag and Attribute Formats

When writing HTML, establish a standard format for tags and attributes, such as using all uppercase for tags and all lowercase for attributes. As an alternative, adhere to the XHTML specification to ensure all HTML documents are valid. Although there are file size trade-offs to consider when creating Web pages, use quoted attribute values and closing tags to ease maintainability.

14.2.3. Keep Blocks of Script Together

In ASP, use script delimiters around blocks of script rather than around each line of script or interspersing small HTML fragments with server-side scripting. Using script delimiters around each line or interspersing HTML fragments with server-side scripting increases the frequency of context switching on the server side; this hampers performance and degrades code readability.

14.3. Commenting

You add comments to an HTML page by wrapping them in comment tags. The HTML element for a comment is the <!-- and --> tag pair. At a minimum, add comments to document the HTML where necessary. Use an introductory (header) comment to document each subroutine and function in the HTML page. In VBScript, comments are indicated by an apostrophe (') character.

Comments serve an additional purpose when they are used in script in an HTML file. Browsers will ignore any unrecognized HTML tag. However, if the script tags are ignored, the browser will attempt to render the script itself as plain text. This is rarely the behavior

you want. The correct way to format script so that older browsers will ignore both the script tags and the script itself is to wrap your script (but not the script tags) in the <!-- and --> comment tags. If you are using VBScript, you will need to use the apostrophe character to add comments to script that is nested within the <!-- and --> comment tags. The following example uses both forms of comment tags:

```
<SCRIPT LANGUAGE="VBSCRIPT">
<!--
   Option Explicit

   Sub UpdateMessage()
      ' This procedure calls code in a scriptlet to get
      ' values for the current day, month, and year, and then
      ' uses the innerHTML property of a <DIV> tag to
dynamically
      ' display those values on the page.

      .
      .
      .
-->
</SCRIPT>
```