The Reddick-Gray Unified Naming Conventions

Version 1.01

In 1992, Stan Leszynski and Greg Reddick and others published a document which became known as the "The Reddick VBA (RVBA) Naming Conventions." As the VBA programming language, for which it was intended, evolved, so did the standard. In reviewing version 6.01 of the standard, published at http://www.xoc.net/downloads/rvbanc.pdf on the Web site of Greg Reddick's company, I am gratified to note that several additions that I suggested in the early days, when I was actively corresponding with Greg, Ken Getz, and others, have been incorporated.

Nevertheless, the standard still has limitations, arising from three sources.

1. **Function and Sub Parameters**. – The most serious weakness is that the convention fails to designate a prefix for function and subroutine parameters that are passed by value, although it makes allowance for those passed by reference. Given that passing by reference is generally, and justifiably, discouraged, I prefer to distinguish parameters *as a class*, retaining the option of incorporating the method by which they are passed in other ways.
2. **Unnecessary Complexity.** – The naming of user defined types and enumerations introduces unnecessary complexity and causes confusion. I propose simplifications for the naming of user defined types, and a different method of naming enumerations.
3. **Naming of Program Labels.** – The convention specified prefixes for labels used within procedures. This labeling method makes cross referencing tools more difficult to use. I use suffixes to address this need.
4. **Other Programming Languages.** – The standard is limited to the VBA programming language. This limits its usefulness as a general tool for all programmers, and in shops, such as mine, in which other languages are employed, including Visual Basic Scripting Edition (VBScript) and JavaScript. With a little effort, the standard can be easily extended to cover VBScript, C, C++, Perl, WinBatch, and other languages.

Before I set forth the complete set of conventions, I think it is important to discuss these four issues, and specify how I address each.

Function and Sub Parameters

Although I liked the Leszynski-Reddick Naming Conventions, as they were initially known, a consultation in 1995 that involved reviewing some code published by of Ken Getz brought to my attention a glaring deficiency, which was that there was no allowance for a scope prefix to cover procedure parameters. This made it difficult to identify the source of a variable that was passed through the parameter lists of three or four levels of functions, and was passed to countless related functions in the same module.

I adopted a rule, and applied it ruthlessly to Ken's code. Applying the rule made the code infinitely easier to follow, and I have used it ever since. The rule is quite simple; all procedure parameter names are preceded by a **p**.

Although the current version of the convention introduces the **r** prefix for parameters that are passed *by reference*, it remains silent with respect to parameters passed *by value*.

Since passing parameters by value is preferred to passing by reference, especially with the increased attention being given to security, this is a serious shortcoming, which *must* be corrected. While it might be important to know that a parameter is passed by reference, I believe it is more important to know that a variable *is a parameter*. If you keep your procedures small, which is also a best practice, it will be easy enough to glance up at the parameter list in the rare case when you need to know whether a parameter is passed by value or by reference.

Nevertheless, there is a way to compactly pack all of this information into the variable name, as shown by **Table 6** on page 13.

Unnecessary Complexity

Reddick goes on at some length about the naming of user defined objects. Having attempted to follow this approach, I concluded that it caused more confusion than communication. I found myself constantly asking myself whether the prefix in question refers to some obscure predefined object type, or whether it was one of my own objects. After years of struggling with this, I concluded that the naming of user defined variables and objects needed to follow a simplified convention involving only a handful of prefixes.

My simplification is based on the idea that there are two classes of user defined variables, and that the prefix should communicate the class to which a variable belongs. With respect to user defined objects, either you are the author of the class, and you know its properties and methods intimately, or you have ready access to its entire source, which is more than can be said about the built-in objects.

Accordingly, I define just two tags, which are listed and defined in **Table 1** below, and repeated in the main list of tags in **Table 4** - Tags for Variable Types, starting on page 6.

Table 1 - Tags for User Defined Types

Tag	Description
uobj	User defined object. At first, I used **obj**, but have reserved this prefix for late bound object variables, and for object variables for which no standard prefix exists.
utp	User Defined Type, defined by Type … End Type block in VBA, and struct in C.

Naming of Program Labels

Conventions for naming of labels within procedures are, to put thing charitably, loose. The RVBA document gives the following examples.

```
ErrorHandler:
ExitProcedure:
```

Elsewhere, I have seen this.

```
MyProcedure:
Error_MyProcedure:
Exit_MyProcedure:
```

I have also seen this.

```
MyProcedure:
Err_MyProcedure:
End_MyProcedure:
```

The first example is ambiguous, because every procedure in a module (not to mention an application composed of possibly hundreds or thousands of modules!) may contain a **ErrorHandler** or **ExitProcedure** label.

While the second and third examples solve the ambiguity problem, they present another problem that causes labels for error handlers and procedure exits to be separated from the procedure name in alphabetical lists of labels, such as are produced by cross reference generators and other code analysis tools.

I propose the use of suffixes to identify error handlers and exit points of procedures, as shown in on page 14.

Other Programming Languages

It might be a stretch to say that the usefulness of the RVBA naming convention is limited to Visual Basic for Applications. That was, after all, Reddick's stated intention. Nevertheless, I feel strongly about building on the work of others. With that in mind, I propose to extend the RVBA Naming Conventions to cover other languages.

The first "foreign" languages to which I adapted the convention were two that are widely used in Web programming, one of which is a close relative of VBA.

- **Visual Basic Scripting Edition (VBScript)** is the programming language of Active Server Pages (ASP), Internet Explorer, and the WebBrowser control. Although slowly being superseded by ASP.NET (aspx pages), there is still quite a bit of ASP code in production. VBScript is also embedded in Web pages intended for rendering in Microsoft Internet Explorer or the WebBrowser control. Public web sites that return content based on the browser type use VBScript, as do many Intranet sites that run in controlled corporate environments where the browser type is set by corporate edict.
- **JavaScript**, or **ECMAScript**, and its close cousin, **Jscript**, are used primarily for code that is intended to run inside a Web browser. Since most modern browsers can run JavaScript, it is the most widely used language for programming such things as Web menus, image rollovers, and local form validators that run inside users' Web browsers.

I have since adapted the convention to other programming languages, including the following.

- **Perl**, the **P**ractical **E**xtraction and **R**eporting **L**anguage, was created by Larry Wall to expedite creation of reports about the contents of the numerous logs generated by Unix

servers. With the help of many others, Perl has been extended well beyond its initial scope. Today, it is used for numerous other tasks, including CGI programming, and it runs on Windows.

- **WIL**, the **W**indows **I**nterface **L**anguage, also known as **WinBatch**, was created by Morrie Wilson, of Wilson WindowWare, Inc., and is the first batch programming language that targets the Windows operating environment. WinBatch was well established by the late 1990's, when VBScript became a part of the Windows landscape. WinBatch programs usually run faster than comparable VBScript programs, can call the Windows API directly, and can be compiled into executable programs that can be distributed without paying any royalties, and are hard to hack.
- **C and C++** are the traditional languages of "real programmers," and are best suited to systems programming and creation of libraries of reusable code.

Although the Reddick Naming Convention is well suited to these languages, there are three issues that must be addressed. **Table 2** below lists and describes these issues.

Table 2 - Issues to Address in Adapting the Reddick Naming Convention

Issue	Comment
Strict versus loose variable typing	Whereas VBA variables are strongly typed, most scripting languages, including VBScript, JavaScript, Perl, and WIL, are not. Although you may choose to retain the strict naming convention to communicate the type of data that you intend to store in each variable, the language will not enforce your rules. Therefore, you may choose to dispense with most variable type tags, as a reminder to make no assumptions about the type of data in a variable. One noteworthy exception may be object variables, which *must* be treated differently than any other type of variable.
Native variable types	♦ The Reddick Naming Convention wisely calls attention to arrays. Since all of the languages to which I have extended the naming convention support arrays, I retain the **a** prefix, even when I elect to drop the type tags. ♦ In addition to arrays, Perl supports the hash, or associative array. Like ordinary arrays, these require special handling, and I assign the **h** prefix to the names of hash variables. ♦ Like VBA, **C** and **C++** variables are strongly typed. However, there are native types in these languages

Issue	Comment
	that have no equivalent in VBA. I have adopted the well established Hungarian prefixes for these, and incorporated them into the overall naming convention.
Special scope issues	In VBA, the value returned by a function must be assigned to a variable whose name is that of the function. Although I abhor this practice, it is required by the language. Fortunately, except for VBScript, none of the other languages to which I have extended this naming convention has this liability. I set aside the **r** prefix to identify the variable that holds the return value of a function, because I think it is important to know the name of the variable whose value will survive, and pass into the caller's scope, when the function returns. **Note**: This designation conflicts with the meaning assigned in the RVBA conventions. However, my method of tagging function parameters resolves this issue.

A Few Final Comments

The remainder of this document follows closely the work of, and the formatting used by, Greg Reddick. For the benefit of readers who are accustomed to using some version of the RVBA Naming Conventions, or one of its variants, I have highlighted places where my convention deviates. Several of these deviations are to clarify ambiguities involving unlike objects that shared a tag in the RVBA Conventions, and to maintain uniformity between like named Access and Visual Basic objects.

Throughout the remainder of this document, most tables contain an extra column, not present in the work of Greg Reddick, headed **Languages**. This column identifies the languages to which the element in the corresponding row applies. In some cases, additional languages are shown in parentheses. These are languages in which the item is optional, as in the case of loosely typed languages, such as VBScript and Perl.

For the convenience of new programmers who have not seen the work of Greg Reddick, I have incorporated his introduction to Hungarian notation, upon which our work is based.

An Introduction to Hungarian Notation

The RVBA and R-G conventions are based on the Hungarian conventions for constructing object names, named for the native country of the inventor, Charles Simonyi. The objective of Hungarian is to convey information about the object concisely and efficiently. Hungarian takes

some getting used to, but once adopted, it quickly becomes second nature. The format of a Hungarian object name is

```
[prefixes]tag[BaseName[Suffixes]]
```

The square brackets indicate optional parts of the object name, which are detailed in **Table 3**.

Table 3 - Components of a Hungarian Variable Name

Component	Meaning
Prefixes	Modify the tag to indicate additional information. Prefixes are all lowercase. They are usually chosen from a standardized list of prefixes, such as the list given later in this document.
Tag	Short set of characters, usually mnemonic, that indicates the type of the object. The tag is all lowercase. It is usually selected from a standardized list of tags, such as the list given later in this document. Tags are optional in loosely typed languages, such as VBScript, Perl, and WIL.
BaseName	One or more words that indicate what the object represents. Capitalize the first letter of each word in the BaseName. Spaces are forbidden.
Suffixes	Additional information about the meaning of the BaseName. Capitalize the first letter of each word in the Suffix. They are usually picked from a standardized list of suffixes, such as the list given later in this document.

Tags

Use the techniques described in the following sections to construct tags to indicate the data type of an object.

VARIABLE TAGS

Use the tags listed in **Table 4** for VBA data types. You can also use a specific tag instead of **obj** for any data type defined by the host application or one of its objects. (See the section "Host Application and Component Extensions to the Conventions," starting at page 17.)

Table 4 - Tags for Variable Types

Tag	Object Type	Languages
bool {f, bln}	Boolean	VBA, VB, C, C++ (VBScript, Perl, WIL)
byte {byt}	Byte	VBA, VB, (VBScript, Perl, WIL)

Tag	Object Type	Languages
chr	Character or array of characters (Note 1)	C, C++, (VBScript, Perl, WIL)
cur	Currency (Note 2)	VBA, VB
date {dtm}	Date (Note 3)	VBA, VB, (C, C++ VBScript, Perl, WIL)
dec	Decimal (Note 2)	VBA, VB
dbl	Double	VBA, VB, C, C++ (VBScript, Perl, WIL)
enm	Enumeration	VBA, VB, C, C++
int	Integer (Note 4)	VBA, VB, C, C++ (VBScript, Perl, WIL)
lng	Long (Note 5)	VBA, VB, C, C++ (VBScript, Perl, WIL)
obj	Object (Note 6)	VBA, VB, C, C++, VBScript, Perl, WIL
o	Object (Note 6)	VBA, VB, C, C++, VBScript, Perl, WIL
sng sgl	Single (Note 7)	VBA, VB, C, C++, (VBScript, Perl, WIL)
str	String (Note 8)	VBA, VB, (VBScript, Perl, WIL)
stf	String (fixed length)	VBA, VB (VBScript, Perl, WIL)
sz	Null terminated (ASCIIZ) string) (Note 9)	C, C++ (Perl, WIL)
uint	Unsigned Integer	C, C++
ulng	Unsigned Long	C, C++
uobj	User Defined Object (Note 10)	VBA, VB, C, C++, (VBScript, Perl, WIL)
utp	User Defined Type (Note 11)	VBA, VB, C, C++, (VBScript)
var	Variant	VBA, VB, (VBScript, Perl, WIL)
xml	XML entity (Note 12)	VBA, VB, C, C++, (VBScript, Perl, WIL)

Notes

1 Arrays of characters are commonplace in C programs, and take the place of fixed length strings. I use the array prefix, **a**, to denote such an entity, and reserve the unadorned **chr** prefix for single characters.

2 So far as I know, VB, VBA, and VB.NET are the only common languages that define a Currency or a Decimal data type. You could get the effect of both in COBOL decades ago, but only as elements of records.

3 Although I prefer the older form, **dtm**, because it conveys that a VBA Date variable

may contain a date, a time, or a date *and* a time, out of deference to Greg Reddick, I list them in the order shown.

There is no date type in C and C++. However, the C runtime library and the Windows API define many data types; I use the date tag for variables of these types.

4 Integer means something a tad different and elusive in C and C++. Whereas Visual Basic defines an integer as a variable that holds a whole number between -32,767 and 32,767, C and C++ distinguish between *signed* and *unsigned* integers.

In a C program, an unsigned integer can hold a number between 0 and 65,536. Consequently, I incorporate the **uint** tag to denote such an entity.

To add more confusion, the definition of C specifies that the size of an integer is machine dependent, and corresponds to the capacity of a machine register. This means that, in a Visual C++ 6.0 program, an integer occupies 32 bits (4 bytes) of storage. However, an examination of the storage reveals that the upper 16 bits are ignored.

5 Long means something a tad different in C and C++. Whereas Visual Basic defines a Long as a variable that holds a whole number between -2,147,483,647 and 2,147,483,647, C and C++ distinguish between *signed* and *unsigned* long integers.

In a C program, an unsigned long can hold a number between 0 and 4,294,967,296. Consequently, I incorporate the **ulng** tag to denote such an entity.

6 In the loosely typed languages, such as VBScript, Perl, and WIL, I prefer the single character prefix, which is more consistent with the simplified naming convention that I usually employ.

7 The alternative form, **sgl**, is carried over from the Hungarian notation, as it is usually applied to C and C++ code. I prefer this form, although, in deference to Greg Reddick, I list them in the order shown.

8 This tag is reserved for a Basic String (BSTR), the native format for strings in all versions of Microsoft Basic of which I am aware. Not surprisingly, the BSTR is also the format in which strings are passed to and from COM/OLE objects, and is the format of strings stored in Variant variables.

9 Null terminated strings, also called C strings, are used extensively in C and C++ programs, and are used to pass text, such as the names of files and folders, to Windows API functions. Visual Basic programmers can usually ignore this for parameters, since the runtime system converts strings for them.

In this context, null terminated strings includes both ANSI and Unicode strings, which are terminated by a double null.

10 This is an area in which I deviate significantly from the conventions laid down by Greg Reddick. Please see Unnecessary Complexity on page 2.

11 As explained under the topic of Unnecessary Complexity, on page 2, I prefer this generic tag to the custom tags favored by Greg Reddick.

12 XML was just beginning to take off in 1999, when Greg Reddick last updated his convention. In 2006, XML objects merit their own generic tag, which applies to all XML objects. I have also defined a set of optional tags associated with various types of XML objects. They are covered in **XML Objects**, starting on page 32.

Here are several examples.

```
lngCount
intValue
strInput
```

You should explicitly declare all variables, each on a line by itself. In VB and VBA, avoid the old-type declaration characters, such as %, &, and $. They are extraneous if you use the naming conventions and the **as Type** clause, and there is no character for most of the data types, such as Boolean, Currency, and Variant. You should explicitly declare all variables of type Variant using the As Variant clause, even though it is the default in VBA. For example:

```
Dim intTotal As Integer
Dim varField As Variant
Dim strName As String
```

CONSTRUCTING PROPERTY NAMES

Properties of a class present a particular problem: should they follow a naming convention to indicate the type? To be consistent with the rest of these naming conventions, they should. However, you may have property names without the tags, especially if the class is to be made available to customers who may not be familiar with these naming conventions. For example, COM objects seldom display variable type information in this way. This can sometimes be annoying to an experienced programmer. Nevertheless, it is a widely accepted standard, and I follow it in all my class libraries.

COLLECTION TAGS

You treat a collection object with a special tag. You construct the tag using the data type of the collection followed by the letter **s**. For example, if you had a collection of Longs, the tag is **lngs**. If it was a collection of forms, the tag for the collection is **frms**. Although, in theory, a collection can hold objects of different data types, in practice, each of the data types in the collection is the same. If you must use different data types in a collection, use the **objs** tag. For example:

```
intsEntries
frmsCustomerData
objsMisc
```

CONSTANTS

Constants always have a data type in VBA. Although VBA chooses this data type for you if you don't specify it, you should always specify the data type for a constant. Constants declared in the General Declarations section of a module should always have a scope keyword of Private or

Public, and be prefixed by the scope prefixes **m** or **g**, respectively. A constant is indicated by appending the letter **c** to the end of the data type tag for the constant. For example:

```
Const intcGray As Integer = 3
Private Const mdblcPi As Double = 3.14159265358979
```

Although this technique is the recommended method of naming constants, if you are more concerned about specifying that you are dealing with constants rather than their data type, you can alternatively use the generic tag **con** or **cnst** instead. For example:

```
Const conPi As Double = 3.14159265358979
```

Finally, if you are accustomed to following the conventions used in C programming, especially the Microsoft Windows API, you may prefer to simply name your constants in upper case, like this.

```
Const CUSTOMER_FLAG  = 536870912
```

This is my preference, regardless of programming language.

MENU ITEMS

The names of menu items should reflect their position in the menu hierarchy. All menu items should use the tag **mnu**, but the BaseName should indicate where in the hierarchy the menu item falls. Use **Sep** in the BaseName to indicate a menu separator bar, followed by an ordinal. For example:

```
mnuFile (on menu bar)
mnuFileNew (on File popup menu)
mnuFileNewForm (on File New flyout menu)
mnuFileNewReport (on File New flyout menu)
mnuFileSep1 (first separator bar on file popup menu)
mnuFileSaveAs (on File popup menu)
mnuFileSep2 (second separator bar on file popup menu)
mnuFileExit (on File popup menu)
mnuEdit (on menu bar)
```

CREATING DATA TYPES

VBA gives you three ways to create new data types: enumerated types, classes, and user-defined types. In each case, you may need to invent a new tag that represents the data type that you create.

Enumerated types

Related groups of constants of the long data type should be made an enumerated type. Invent a tag for the type, append a "c," and then define the enumerated constants using that tag, or use the generic **enm** tag,. Because the name used in the **Enum** line is seen in the object

browser, you can add a BaseName to the tag to spell out the abbreviation indicated by the tag. For example:

```
Public Enum ervcErrorValue
      ervcInvalidType = 205
      ervcValueOutOfBounds
End Enum
```

The BaseName should be singular, so that the enumerated type should be **ervcErrorValue**, not **ervcErrorValues**. The tag that you invent for enumerated types can then be used for variables that can contain values of that type. For example:

```
Dim erv As ervcErrorValue
Private Sub Example(ByVal ervCur As ervcErrorValue)
```

While VBA, C, and C++ only provides enumerated types of groups of the long type, you can still create groups of constants of other types. Just create a set of constant definitions using an invented tag. For example:

```
Public Const estcError205 As String = "Invalid type"
Public Const estcError206 As String = "Value out of bounds"
```

Unfortunately, because this technique doesn't actually create a new type, you don't get the benefit of the compiler performing type checking for you. You create variables that will hold constants using a similar syntax to variables meant to hold instances of enumerated types. For example:

```
Dim estError As String
```

TAGS FOR CLASSES AND USER-DEFINED TYPES

This is an area in which I deviate significantly from the conventions laid down by Greg Reddick. Please see Unnecessary Complexity on page 2. For consistency, I retained this heading, however.

I use the simple prefixes listed in **Table 5** for user defined objects and classes.

Table 5 - Prefixes for User Defined Objects and Classes

Prefix	Description
uobj	User defined class name. I use this in VBA and VB.
o	For consistency, I use the abbreviated prefix in Perl, VBScript, and other loosely typed languages. Of the languages in which I have used the abbreviated tag, Perl is the only one that lends itself to defining your own classes. Although VBScript has the Class keyword, its support of user defined classes seems pretty weak. Although I am sure that they exist, I have never seen a Visual Basic script that defined its own classes.

Prefix	Description
uo	This alternative form can be used in loosely typed languages to distinguish user defined objects from predefined or otherwise packaged objects.

POLYMORPHISM

In VBA, you use the Implements statement to derive classes from a base class. The derived class should use the same tag as the base class. The derived classes, though, should use a different BaseName from the base class. For example:

```
anmAnimal (base class)
anmZebra (derived class of anmAnimal)
anmElephant (derived class of anmAnimal)
```

This logic of naming derived classes is used with forms, which are all derived from the pre-defined Form base class and use the frm tag. If a variable is defined to be of the type of the base class, then use the tag, as usual. For example:

```
Dim anmArbitrary As anmAnimal
Dim frmNew As Form
```

On the other hand, if you define a variable as an instance of a derived class, include the complete derived class name in the variable name. For example:

```
Dim anmZebraInstance As anmZebra
Dim anmElephantExample As anmElephant
Dim frmCustomerData As frmCustomer
```

CONSTRUCTING PROCEDURES

VBA procedures require you to name various items: procedure names, parameters, and labels. These objects are described in the following sections. I deviate significantly from the RVBA conventions.

Constructing Procedure Names

VBA, VB. NET, and other languages name event procedures, and you cannot change them. You should use the capitalization defined by the system. For user-defined procedure names, capitalize the first letter of each word in the name. For example:

```
cmdOK_Click
GetTitleBarString
PerformInitialization
```

Procedures should always have a scope keyword, Public or Private, when they are declared. For example:

```
Public Function GetTitleBarString() As String
Private Sub PerformInitialization
```

Unless the functions will be exposed to users who may not be accustomed to these conventions, their names should be tagged to indicate the type of variable that they return. For example:

```
Public Function dblAreaOfCircle(pdblRadius as Double) as Double
```

Note the use of the `as Double` clause to specify the type of data that the function returns. Just as variables should always have an explicit type, so should functions. Likewise, functions that return a Variant should say so, even though this is the default for VB and VBA. Sub procedures, which return nothing, should have no type prefix. Absence of a prefix reinforces their status as Sub procedures.

You should prefix all parameters in a procedure definition with ByVal or ByRef, even though ByRef is optional and redundant. Procedure parameters are named the same as simple variables of the same type, except that the name is prefixed as shown in Table 6 - Prefixes for Parameter Name.

Table 6 - Prefixes for Parameter Name

Prefix	Description
p	Parameter, conveying no information about whether passed by value or reference
pr	Parameter passed by reference, allowing changes to flow back into the caller's address space
pv	Parameter passed by value, confining changes to the current procedure

For example:

```
Public Sub TestValue(ByVal pvintInput As Integer, _
      ByRef prlngOutput As Long)
Private Function GetReturnValue(ByVal pvstrKey As String, _
      ByRef prgph As Glyph) As Boolean
```

Procedure Labels

Procedure Labels are named using upper and lower case, capitalizing the first letter of each word. For example:

```
MyProcedure:
MyProcedure_Err:
MyProcedure_End:
```

While Greg Reddic specifies prefixes for procedure names, I specify suffixes. Please see Naming of Program Labels on page 2 for an explanation of my reason for deviating.

Table 7 - Suffixes for Procedure Labels

Suffix	Description
_Err	User defined object. I used to use **obj**, but have reserved this prefix for late bound object variables.
_nnn	Internal label, used for short local jumps. The value of **nnn** is a number, starting with **010**, and incrementing by **10**. See the text below for more about this.
_End	Variant, which can hold anything, even an object reference, and can hold different types of data throughout its lifetime I apply the **a** prefix to this type to denote an array of variants.

Although few procedures should need them, internal labels should be assigned in such a way that additional labels can be inserted between existing labels, without the need to renumber existing labels.

- By numbering the labels sequentially, they appear in sorted cross references in their order of appearance in the code. Also, if you see a label with a suffix of **020**, you know that there is at least one other label above it.
- This numbering convention is inspired by the naming conventions that we used at least as long ago as the late 1970's to name labels in COBOL programs.

PREFIXES

Prefixes modify an object tag to indicate more information about an object.

Arrays of Objects Prefix

Arrays of an object type use the prefix **a**. For example:

```
aintFontSizes
astrNames
```

Index Prefix

You indicate an index into an array by the prefix **i**, and for consistency the data type should always be a long. You may also use the index prefix to index into other enumerated objects, such as a collection of user-defined classes. For example:

```
iaintFontSizes
iastrNames
igphsGlyphCollection
```

Prefixes for Scope and Lifetime

Three levels of scope exist for each variable in VBA: Public, Private, and Local. A variable also has a lifetime of the current procedure, the lifetime of the object in which it is defined, or the lifetime of the procedure that called the procedure that returns it. Use the prefixes in .**Table 8**.

Table 8 - Prefixes for Variable Scope, Lifetime, and How Passed

Tag	Object Type	Languages
(None)	Local variable, procedure-level lifetime, declared with "Dim"	All
s	Local variable, object lifetime, declared with "Static"	VBA, VB, C. C++ (VBScript, Perl, WIL)
m	Private (module) variable, object lifetime, declared with "Private"	VBA, VB, C. C++ (VBScript, Perl, WIL)
g	Public (global) variable, object lifetime, declared with "Public"	VBA, VB, C. C++ (VBScript, Perl, WIL)
r	Value returned by function	C, C++, Perl, WIL

You also use the "m" and "g" constants with other objects, such as constants, to indicate their scope. For example:

```
intLocalVariable
mintPrivateVariable
gintPublicVariable
mdblcPi
```

You may follow the prefix with an underscore, so that the examples above would look like this.

```
m_intPrivateVariable
g_intPublicVariable
m_dblcPi
```

VBA allows several type declaration words for backward compatibility. The older keyword "Global" should always be replaced by "Public," and the "Dim" keyword in the General Declarations section should be replaced by "Private." These conventions carry forward into the Microsoft .NET languages, suggesting that the older keywords are deprecated.

Other Prefixes

Table 9 - Other commonly-used prefixes

Prefix	Object Type	Languages
c cnt	Count of some object type	All
h	Handle to a Windows object (Note 1)	All *except* Perl
r	Value returned by a function (Note 2)	All except VB, VBA, and VBScript

Notes

1 The only type of handle known to Perl is a **filehandle**, and these are specified in upper case. So far as I am aware, this is a requirement of the language. In any case, it is a globally accepted coding standard among Perl programmers.

Further, in Perl, the **h** prefix is reserved for hashes, or associative arrays.

2 All dialects of the Basic programming language, being descendants of Fortran, require that the return value be assigned to a variable with the sane name as the function.

Although I have been programming in Basic since 1978, and learned Fortran before I learned Basic, this linguistic quirk still bothers me, as the return value is implicitly defined. (However, from a theoretical perspective, this syntax rule is quite logical.) Indeed, it is a syntax error to define a variable of the sane name as the function anywhere within the procedure itself, or within any procedure or module to which the function is visible.

SUFFIXES

Suffixes modify the base name of an object, indicating additional information about a variable. You will likely create your own suffixes that are specific to your development work. **Table 10** lists some generic VBA suffixes.

Table 10 - Generic VBA Suffices

Suffix	Description
Min	The absolute first element in an array or other kind of list
First	The first element to be used in an array or list during the current operation
Last	The last element to be used in an array or list during the current operation
Max	The absolutely last element in an array or other kind of list

Suffix	Description
Cnt	Used with database elements to indicate that the item is a Counter. Counter fields are incremented by the system and are numbers of either type Long or type Replication Id.
Idx	The item is an index. This provides an alternative to the **i** prefix for array and collection indices, discussed above.

Here are some examples:

```
iastrNamesMin
iastrNamesMax
iaintFontSizesFirst
igphsGlyphCollectionLast
lngCustomerIdCnt
varOrderIdCnt
```

FILE NAMES

When naming items stored on the disk, no tag is needed because the extension already gives the object type. For example:

```
Test.Frm (frmTest form)
Globals.Bas (globals module)
Glyph.Cls (gphGlyph class module)
```

Host Application and Component Extensions to the Conventions

Each host application for VBA, as well as each component that can be installed, has a set of objects it can use. This section defines tags for the objects in the most widely used host applications and components. These extensions are appropriate for use in any language that hosts an object of the types discussed below.

MICROSOFT ACCESS 2000, VERSION 9.0 (AND LATER) OBJECTS

Table 11 lists Microsoft Access object variable tags. Besides being used in code to refer to these object types, these same tags are used to name these kinds of objects in the designers.

Table 11 - Microsoft Access object variable tags.

Tag	Description
aob	AccessObject
aops	AccessObjectProperties
aop	AccessObjectProperty
app	Application

Tag	Description
bfr	BoundObjectFrame
cbk	CheckBox
cbo	ComboBox
cmd	CommandButton
ctl	Control
ctls	Controls (collection)
ocx	CustomControl
dap	DataAccessPage
dcm	DoCmd
frm	Form
fcd	FormatCondition
fcds	FormatConditions (collection)
frms	Forms (collection)
grl	GroupLevel
hyp	Hyperlink
img	Image
lbl	Label
lin {lne}	Line
lst {lbo]	ListBox
bas {mod}	Module
ole	ObjectFrame
opt	OptionButton
fra	OptionGroup (frame)
brk	PageBreak
pal	PaletteButton
prps	Properties

Tag	Description
shp	Rectangle
ref	Reference
refs	References
rpt	Report
rpts	Reports
scr	Screen
sec	Section
sfr	Subform
srp	SubReport
tab	TabControl
txt	TextBox
tgl	ToggleButton

Some examples:

```
txtName
lblInput
```

For ActiveX custom controls, you can use the tag **ocx** as specified in **Table 11** above or more specific object tags that are listed later in this document in **Tables 14 and 15**. For an ActiveX control that does not appear in Tables 14 or 15, you can either use **ocx** or invent a new tag.

MICROSOFT DAO 3.6 OBJECTS

Microsoft DAO is the programmatic interface to the Jet database engine shared by Access, Visual Basic, and Visual C++. The tags for DAO 3.6 objects are shown in **Table 12**.

Table 12 – Microsoft Data Access Objects (DAO) object tags

Tag	Description
cnt	Container
cnts	Containers
db	Database
dbs	Databases (collection)
dbe	DBEngine
doc	Document

Tag	Description
docs	Documents
err	Error
errs	Errors
fld	Field
flds	Fields
grp	Group
grps	Groups
idx	Index
idxs	Indexes
prm	Parameter
perm	Parameters
pdbe	PrivDBEngine
prp	Property
prps	Properties
qry	QueryDef
qrys	QueryDefs
rst	Recordset
rsts	Recordsets
rel	Relation
rels	Relations
tbl	TableDef
tbls	TableDefs
usr	User
usrs	Users
wrk	Workspace
wrks	Workspaces

Here are some examples:

```
rstCustomers
idxPrimaryKey
```

Table 13 lists the tags used to identify types of objects in a Microsoft Access database.

Table 13 – Microsoft Access Database Object Tags

Tag	Description
tbl	Table
qry	Query
frm	Form
rpt	Report
mcr	Macro
bas {mod}	Module
dap	DataAccessPage

If you wish, you can use tags that are more exact or suffixes to identify the purpose and type of a database object. If you use the suffix, use the tag given from **Table 13** to indicate the type. Use either the tag or the suffix along with the more general tag, but not both. The tags and suffixes are shown in **Table 14**

Table 14 - Specific Object Tags and Suffixes for Microsoft Access Database Objects

Tag	Suffix	Object Type
tlkp	Lookup	Table (lookup)
qsel	(none)	Query (select)
qapp	Append	Query (append)
qztb	XTAB	Query (crosstab)
qddl	DDL	Query (DDL)
qdel	Delete	Query (delete)
qflt	Filter	Query (filter)
qlkp	Lookup	Query (lookup)
qmak	MakeTable	Query (make table)
qspt	PassThru	Query (SQL pass-through)
qtot	Totals	Query (totals)
quni	Union	Query (union)
fdlg	Dlg	Form (dialog)

Tag	Suffix	Object Type
fmnu	Mnu	Form (menu)
fmsg	Msg	Form (message)
fsfr	SubForm	Form (subform)
rsrp	SubReport	Form ((subreport)
mmnu	Mnu	Macro (menu)

Here are some examples:

```
tblValidNamesLookup
tlkpValidNames
fmsgError
mmnuFileMnu
```

When naming objects in a database, do not use spaces. Instead, capitalize the first letter of each word. For example, instead of Quarterly Sales Values Table, use tblQuarterlySalesValues.

There is strong debate over whether fields in a tables should have tags. Whether you use them is up to you. However, if you do use them, use the tags from **Table 15**.

Table 15 – Microsoft Access Table Field Name Tags (if you decide to use them)

Tag	Description
autoi {lng}	Autoincrementing (either sequential or random) Long (used with the suffix Cnt) (Note 1)
bin	Binary
byt {byte }	Byte (Note 2)
cur	Currency
dtm {date}	Date/Time (Note 3)
dbl	Double
guid	Globally unique identified (GUID) used for replication AutoIncrement fields
int	Integer
lng	Long
mem	Memo
ole	OLE

Tag	Description
sng sgl	Single (Note 4)
str	Text (string)
bool	Yes/No

Notes

1 The Reddick convention specifies **lng**, which I find a bit confusing, as it suggests that you can set the value in code. Although I leave his **lng** prefix, I prefer the less ambiguous **autoi** prefix.

2 The Reddick convention specifies **byte**, which, although quite clear, is longer than the majority of the prefixes. Instead, I prefer the more consistent, equally communicative, **byt**.

3 For the same reasons that I favor **dtm** for program variables that contain dates, I prefer it for table column (field) names.

4 The alternative form, **sgl**, is carried over from the Hungarian notation, as it is usually applied to C and C++ code. I prefer this form, although, in deference to Greg Reddick, I list them in the order shown.

Whatever you do, be consistent.

MICROSOFT VISUAL BASIC 6.0 OBJECTS

Table 16 shows the tags for Visual Basic 6.0 objects. Since most of these objects also exist in Microsoft Access, and can be treated alike, I did my best to use identical prefixes. This lead to one deviation from the RVBA convention, which is noted.

Table 16 - Microsoft Visual Basic 6.0 Object Tags

Tag	Description
app	App
chk	CheckBox
clp	Clipboard
cbo	ComboBox
cmd	CommandButton
ctl	Control
dat	Data
dir	DirListBox

Tag	Description
drv	DriveListBox
fil	FileListBox
frm	Form
fra	Frame
glb	Global
hsb	HscrollBar
img	Image
lbl	Label
lics	Licenses
lin	Line
lst {lbo]	Listbox (Note 1)
mdi	MDIForm
mnu	Menu
ole	OLE
opt	OptionButton
pic	PictureBox
prt	Printer
prp	PropertyPage
scr	Screen
shp	Shape
txt	Textbox
uctl	UserControl
udoc	UserDocument
vsb	VscrollBar

Notes

1 Since there is no difference between this entity and the one covered by **Table 11**, on page 17, I assigned the same prefixes. As I did above, I included the lbo prefix, marking it as a "grandfathred" prefix. Incidentially, since it is the first prefix that I

learned, and the one that makes the most sense to me, I still use it.

MICROSOFT ACTIVEX DATA OBJECTS 2.1 (AND LATER) TAGS

Office 2000 provides version 2.1 of the ActiveX Data Objects library. **Table 17** lists the recommended tags for this version of ADO.

Note: Many of the ADO, ADOX, and JRO tags overlap with existing DAO tags. Make sure you include the object library name in all references in your code, so there's never any possibility of confusion. For example, use

```
Dim rst As ADODB.Recordset
```

or

```
Dim cat As ADOX.Catalog
```

rather than using the object types without the library name. This will not only make your code more explicit and avoid confusion about the source of the object, but will also make your code run a bit faster.

Table 17 - Microsoft ADO 2.1 Object Tags

Tag	Description
cmn {cmd}	Command
cnn {cnx}	Connection
err	Error
errs	Errors
fld	Field
flds	Fields
prm	Parameter
prms	Parameters
prps	Properties
prp	Property
rst	Recordset

MICROSOFT ADO EXT. 2.1 (AND LATER) FOR DDL AND SECURITY (ADOX) TAGS

In order to support DDL and security objects within Jet database, Microsoft provides ADOX, an additional ADO library of objects. **Table 18**lists tags for the ADOX objects.

Table 18 - Microsoft ADOX Object Tags

Tag	Description
cat	Catalog
clm	Column
clms	Columns
cmd	Command
grp	Group
grps	Groups
idx	Index
idxs	Indexes
key	Key
keys	Keys
prc	Procedure
prcs	Procedures
prp	Property
prps	Properties
tbl	Table
tbls	Tables
usr	User
usrs	Users
vw	View
vws	Views

MICROSOFT JET AND REPLICATION OBJECTS 2.1 (AND LATER)

In order to support Jet's replication features, ADO provides another library (JRO). **Table 19** lists suggested tags for the JRO objects.

Table 19 - Microsoft JRO Object Tags

Tag	Description
flt	Filter
flts	Filters
jet	JetEngine
rpl	Replica

MICROSOFT SQL SERVER AND MICROSOFT DATA ENGINE (MSDE) OBJECTS

Table 20 lists tags for Microsoft SQL Server and the Microsoft Data Engine (a limited-connection version of SQL Server 7) objects.

Table 20 - SQL Server/MSDE Object Tags

Tag	Description
tbl	Table
proc sp_ sp	Stored Procedure (Note 1)
trg	Trigger
qry	View
dgm	Database Diagram
pk	Primary Key
fk	Foreign Key
idx	Other (non-key) index
rul	Check Constraint
def	Default

Notes

1 The sp_ and sp prefixes are commonly used. Since both are as logical as is proc, I include them here.

MICROSOFT SQL SERVER TABLE COLUMN (FIELD) NAMES

Table 21, on page 28 lists prefixes for the Microsoft SQL Server data types defined at http://msdn2.microsoft.com/en-us/library/aa258271(SQL.80).aspx. Fields are listed in the order shown at that page, as of 20 February 2007. Where applicable, these prefixes follow the pattern established for tagging of similar objects in Microsoft Access tables and program variables. In a few cases, alternative prefixes are shown in square brackets.

Table 21 - Microsoft SQL Server Table Column (Field) Name Prefixes

Tag	Description
bint	BigInt
int	Int
sint	SmallInt
tint	TinyInt
Bit [bln] [bool]	Bit
dec	Decimal
num	Numeric
mny	Money
smny	SmallMoney
flt [fl]	Float
real [rl]	Real
dtm	DateTime
sdt	SmallDateTime
chr [char]	Char
vchr	VarChar
txt	Text
nchr	Nchar
nvchr	NVarChar

Tag	Description
ntxt	Ntext
bin	Binary
vbin	VarBinary
img	Image
csr	Cursor
svar	SQL_Variant
rslt	Table (stored result set)
tms	TimeStamp
guid	UniqueIdentifier

MICROSOFT COMMON CONTROL OBJECTS

Windows 95 and Windows NT have a set of common controls that are accessible from VBA, C++, and many other programming languages. **Table 22** lists the tags for objects created using these controls.

Table 22 - Microsoft Common Control Object Tags

Tag	Description
ani	Animation
btn	Button (Toolbar)
bmn	ButtonMenu (Toolbar)
bmns	ButtonMenus (Toolbar)
bnd	Band (Coolbar)
bnds	Bands (Coolbar)
bnp	BandsPage (CoolBar)
btns	Buttons (Toolbar)
cbr	Coolbar
cbp	CoolBarPage (CoolBar)
hdr	ColumnHeader (ListView)
hdrs	ColumnHeaders (ListView)
cbi	ComboItems (ImageCombo)
ctls	Controls (Note 1)

Tag	Description
dto	DataObject
dtf	DataObjectFiles
dtp	DTPicker
fsb	FlatScrollBar
imc	ImageCombo
iml	ImageList
lim	ListImage
lims	ListImages
lit	ListItem (ListView)
lits	ListItems (ListView)
lsi	ListSubItem (ListView)
lsis	ListSubItems (ListView)
lvw	ListView
mvw	MonthView
nod	Node (TreeView)
nods	Nodes (TreeView)
pnl	Panel (Status Bar)
pnls	Panels (Status Bar)
prb	ProgressBar
sld	Slider
sbr	StatusBar
tab	Tab (Tab Strip)
tabs	Tabs (Tab Strip)
tbs	TabStrip
tbr	Toolbar
tvw	TreeView
udn	UpDown

Notes

1 To maintain consistency with the naming of Microsoft Access and Visual Basic 6.0 controls, I changed this from **ctl** to **ctls**.

OTHER CUSTOM CONTROLS AND OBJECTS

Finally, **Table 23** lists the tags for other commonly used custom controls and objects.

Table 23 - Tags for Commonly-used Custom Controls

Tag	Description
cdl	CommonDialog (Common Dialog)
dbc	DBCombo (Data Bound Combo Box)
dbg	DBGrid (Data Bound Grid)
dls	DBList (Data Bound List Box)
gau	Gauge (Gauge)
gph	Graph (Graph)
grd	Grid (grid)
msg	MAPIMessages (Messaging API Message Control)
ses	MAPISession (Messaging API Session Control)
msk	MaskEdBox (Masked Edit Textbox)
key mhs	MhState (Key State) (Note 1)
mmc	MMControl (Multimedia Control)
com	MSComm (Communication Port)
out	Outline (Outline Control)
pcl	PictureClip (Picture Clip Control)
rtf	RichTextBox (Rich Textbox)
spn	SpinButton (Spin Button)

Notes

1 Although Reddick assigns the prefix **key**, I prefer the unambiguous **mhs**.

XML OBJECTS

Because they have come into such wide use, the objects exposed by the `MSXMLx.DLL` libraries merit a place in any modern naming convention. **Table 24** lists the prefixes.

Table 24 - XML Object Prefixes

Tag	Description
xmlcd	XML CDATA element
xmld	XML DOMDocument
xmlds	XML Data Source
xmle	XML Element tag
xmlh	XML HTTP Object
xmln	Generic XML Node
xmlpi	XML Processing Instruction
xmls	XML Schema object
xmlsx	XML SAX object
xmlt	XML Template

Summary

Using a naming convention requires a considerable initial effort on your part. The payoff comes when either you or another programmer has to revisit your code later. Using the conventions given here will make your code more readable and maintainable.

Greg Reddick is the President of **Xoc Software**, a software development company developing programs in Visual Basic, Microsoft Access, C/C++, and for the web. He leads training seminars in Visual Basic for Application Developers Training Company (AppDev). In a previous life, he worked for four years on the Access development team at Microsoft. Greg can be reached at grr@xoc.net or from the Xoc Software web site, http://www.xoc.net.

David Gray is President and Chief Wizard of **WizardWrx**, a software development and consulting company that applies numerous programming languages and techniques to solve difficult business problems. David is proficient in VB, VBA, Perl, WIL (WinBatch), and other languages, and uses C and C++ to create advanced libraries for encryption and tight integration with Microsoft Windows. David can be reached at dgray@wizardwrx.com or through the WizardWrx Web site at http://www.wizardwrx.com/.

REVISION HISTORY

Version	Date	Comment
1.00	2006/09/11	First published version
1.01	2007/02/20	♦ Add prefixes for Microsoft SQL Server table column (field) names. ♦ Add missing table captions. ♦ Add missing headings on tables that span multiple pages. ♦ General cosmetic cleanup.