Return to Coding Standards overview

GUI Computing Coding Standards

Author	Jim Karabatsos
Copyright Notice	© 1996 GUI Computing Pty Ltd
Date Created	18 March, 1996
Date of last modification	14 June, 2004
Author of last modification	Mark Trescowthick
Version	1.5.2
Status	Release

Table of Contents

Overview

This document is a working document - it is not designed to meet the requirement that we have "a" coding standard but instead is an acknowledgment that we can make our lives much easier in the long term if we all agree to a common set of conventions when writing code.

Inevitably, there are many places in this document where I have simply had to make a choice between two or more equally valid alternatives. I have tried to actually think about the relative merits of each alternative but inevitably some of my personal preferences have come into play. I hope that this has only happened where it was a line-ball anyway.

This document is not fixed in concrete, but it is not a suggestion, either. The only thing worse than no coding standard is multiple coding standards so these coding standards are mandatory where they apply (see the section "*When Does This Document Apply*").

However, if you think that something could be improved, or even if you think that I've made a wrong call somewhere, then

e-mail me and let's discuss it.

I hope you find that this document is actually readable. I hate standards documents that are so dry as to be about as interesting as reading the white pages. However, do not assume that this document is any less important or is treated any less lightly by GUI management than its drier cousins; believe me, GUI takes these standards very seriously.

When Does This Document Apply?

It is the intention that all code written for or by GUI Computing adhere to this standard. However, there are some cases where it is impractical or impossible to apply these conventions, and others where it would be A Bad Thing.

This document applies to ALL CODE except the following :

- **Code changes made to existing systems not written to this standard.**
  In general, it is a good idea to make your changes conform to the surrounding code style wherever possible. You might choose to adopt this standard for major additions to existing systems or when you are adding code that you think will become part of the GUI Codebank.
- **Code written for customers that require that their standards be adopted.**
  It is not uncommon for GUI to work with customers that have their own coding standards. Most coding standards derive at least some of their content from the Hungarian notation concept and in particular from a Microsoft white paper that documented a set of suggested naming standards. For this reason many coding standards are broadly compatible with each other. This document goes a little further than most in some areas; however it is likely that these extensions will not conflict with most other coding standards. But let's be absolutely clear on this one folks: if there is a conflict, the customer's coding standards are to apply. Always.

Naming Standards

Of all the components that make up a coding standard, naming standards are the most visible and arguably the most important.

Having a consistent standard for naming the various 'thingies' in your program will save you an enormous amount of time both during the development process itself and also during any later maintenance work. I say 'thingies' because there are so many different things you need to name in a VB program, and the word 'object' has a particular meaning.

Variables

Variable names are used very frequently in code; most statements contain the name of at least one variable. By using a consistent naming system for variables, we are able to minimise the amount of time spent hunting down the exact spelling of a variable's name.

Furthermore, by encoding into the variable's name some information about the variable itself, we can make it much easier to decipher the meaning of any statement in which that variable is used and to catch a lot of errors that are otherwise very difficult to find.

The attributes of a variable that are useful to encode into its name are its scope and its data type.

Scope

In Visual Basic there are three scopes at which a variable may be defined. If defined within a procedure, the variable is local to that procedure. If defined in the general declarations area of a form or module then that variable can be referenced from all procedures in that form or module and is said to have module scope. Finally, if it is defined with the Global keyword, then it is (obviously) global to the application.

A somewhat special case exists for arguments to procedures. The names themselves are local in scope (to the procedure itself). However, any change applied to that argument may, in some cases, affect a variable at a totally different scope. Now that may be exactly what you want to happen - it is by no means always an error - but it can also be the cause of subtle and very frustrating errors.

Data Type

VB supports a rich assortment of data types. It is also a very weakly typed language; you can throw almost anything at anything in VB and it will usually stick. In VB4, it gets even worse. Subtle bugs can creep into your code because of an unintended conversion done behind the scenes for you by VB.

By encoding the type of the variable in its name, you can visually check that any assignment to that variable is sensible. This will help you pick up the error very quickly.

Encoding the data type into the variable name has another benefit that is less often cited as an advantage of this technique: reuse of data names. If you need to store the start date in both a string and a double, then you can use the same root name for both variables. The start date is always the StartDate; it just takes a different tag to distinguish the different formats it is stored in.

Option Explicit

First things first. Always use Option Explicit. The reasons are so obvious that I won't discuss it any further. If you don't agree, see me and we'll fight about it --- errr --- we'll discuss it amicably.

Variable Names

Variables are named like this :

```
scope + type + VariableName
```

The scope is encoded as a single character. The possible values for this character are :

g	This denotes that the variable is Global in scope
m	This denotes that the variable is defined at the Module (or Form) level

The *absence* of a scope modifier indicates that the variable is local in scope.

> *I will use text sidelined like this to try and explain why I have made certain choices. I hope it is helpful to you.*
>
> *Some coding standards require the use of another character (often 'l') to indicate a local variable. I really don't like this. I don't think that it adds to the maintainability of the code at all and I do think it makes the code a lot harder to read. Besides, it's ugly.*

The type of the variable is encoded as one or more characters. The more common types are encoded as single characters while the less common types are encoded using three or four characters.

> *I thought long and hard about this one, folks. It is certainly possible to come up with one-character codes for all of the built-in data types. However, it is really hard to remember them all, even if you made them up yourself. On balance, I think that this a better approach.*

The possible values for the type tag are :

i	integer	16-bit signed integer
l	long	32-bit signed integer
s	string	a VB string
n	numeric	integer of no specified size (16 or 32 bits)
c	currency	64-bit integer scaled by 10^{-4}
v	variant	a VB variant
b	boolean	in VB3: an integer used as a boolean, in VB4: a native boolean data type
dbl	double	a double-precision floating point number
sng	single	a single-precision floating point number
flt	float	a floating point number of no particular precision
byte	byte	an eight-bit binary value (VB4 only)
obj	object	a generic object variable (late binding)
ctl	control	a generic control variable

> *We do NOT use VB's type suffix characters. These are redundant given our prefixes and no-one remembers more than a few of them anyway. And I don't like them.*

It is important to note that defining a prefix for the underlying implementation type of the variable, while of some use, is often not the best alternative. Far more useful is defining a prefix based on the underlying aspects of the data itself. As an example, consider a date. You can store a date in either a double or a variant. You don't (usually) care which one it is because only dates can be logically assigned to it.

Consider this code :

```
dblDueDate = Date() + 14
```

We know that dblDueDate is stored in a double precision variable, but we are relying on the name of the variable itself to identify that it is a date. Now suddenly we need to cope with null dates (because we are going to process external data, for example). We need to use a variant to store these dates so we can check whether they are null or not. We need to change our variable name to use the new prefix and find everywhere that it is used and make sure it is changed :

```
vDueDate = Date() + 14
```

But really, DueDate is first and foremost a date. It should therefore be identified as a date using a date prefix :

```
dteDue = Date() + 14
```

This code is immune from changes in the underlying implementation of a date. In some cases, you might need to know whether it is a double or a variant, in which case the appropriate tag can also be used after the date tag :

```
dtevDue = Date() + 14
dtedblDue = Date() + 14
```

The same arguments apply for a lot of other situations. A loop index can be any numeric type. (In VB4, it can even be a string!) You will often see code like this :

```
Dim iCounter As Integer
For iCounter = 1 to 10000
   DoSomething
Next iCounter
```

Now, what if we need to process the loop 100,000 items? We need to change the type of the variable to a long integer, then change all occurrences of its name too.

If we had instead written the routine like this :

```
Dim nCounter As Integer
For nCounter = 1 to 10000
    DoSomething
Next nCounter
```

we could have updated the routine by changing the Dim statement only.

Windows handles are an even better example. A handle is a 16-bit item in Win16 and a 32-bit item in Win32. It makes more sense to tag a handle as a handle than as an integer or a long. Porting such code will be much easier - you change the definition of the variable only while the rest of your code remains untouched.

Here is a list of common data types and their tags. It is not exhaustive and you will quite likely make up several of your own during any large project.

h	handle	16 or 32 bit handle	hWnd
dte	date	stored in either a double or variant	dteDue

The body of the variable's name is comprised of one or more complete words, with each word starting with a capital letter, as in ThingToProcess. There are a few rules to keep in mind when forming the body of the name.

Use multiple words - but use them carefully. It can be hard to remember whether the amount due is called AmountDue or DueAmount. Start with the fundamental and generic *thing* and then add modifiers as necessary. An amount is a more fundamental *thing* (what's a due?), so you should choose AmountDue. Similarly :

Correct	**Incorrect**
DateDue	DueDate
NameFirst	FirstName
ColorRed	RedColour
VolumeHigh	HighVolume
StatusEnabled	EnabledStatus

You will generally find that nouns are more generic than adjectives. Naming variables this way means that related variables tend to sort together, making cross reference listings more useful.

There are a number of qualifiers that are commonly used when dealing with a set of things (as in an array or table).

Consistent use of standard modifiers can significantly aid in code maintenance. Here is a list of common modifiers and their meaning when applied to sets of things:

Count	count	the number of items in a set	SelectedCount
Min	minimum	the minimum value in a set	BalanceMin
Max	maximum	the maximum value in a set	RateHigh
First	first	the first element of a set	CustomerFirst
Last	last	the last element of a set	InvoiceLast
Cur	current	the current element of a set	ReportCur
Next	next	the next element of a set	AuthorNext
Prev	previous	the previous element of a set	DatePrev

At first, placing these modifiers after the body of the name might take a little getting used to; However, the benefits of adopting a consistent approach are very real.

User Defined Types (UDTs)

UDTs are an extremely useful (and often overlooked) facility supported by VB. Look into them - this is not the right document to describe how they work or when to use them. We will focus on the naming rules.

First, remember that to create an instance of a UDT, you must first define the UDT and then Dim an instance of it. This means you need *two* names. To distinguish the type from instances of that type, we use a single letter prefix to the name, as follows:

```
Type TEmployee
   nID      As Long
   sSurname As String
   cSalary  As Currency
End Type

Dim mEmployee As TEmployee
```

We cannot use the C conventions here as they rely on the fact that C is case-sensitive. We need to come up with our own convention. The use of an upper case 'T' may seem at odds with the other conventions so far presented but it is important to visually distinguish the names of UDT definitions from the names of variables. Remember that a UDT definition is not a variable and occupies no space.

Besides which, that's how we do it in Delphi.

Arrays

There is no need to differentiate the names of arrays from scalar variables because you will always be able to recognise an array when you see it, either because it has the subscripts after it in parentheses or because it is wrapped up inside a function like UBound that only makes sense for arrays.

Array names should be plurals. This will be especially helpful in the transition to collections in VB4.

You should always dimension arrays with both an upper and a lower bound. Instead of:

```
Dim mCustomers(10) as TCustomer
```

try:

```
Dim mCustomers(0 To 10) as TCustomer
```

Many times, it makes far more sense to create 1-based arrays. As a general principle, try to create subscript ranges that allow for clean and simple processing in code.

Procedures

Procedures are named according to the following convention:

```
verb.noun
verb.noun.adjective
```

Here are some examples:

Good:

```
FindCustomer
FindCustomerNext
UpdateCustomer
UpdateCustomerCur
```

Bad: `CustomerLookup` should be `LookupCustomer`
`GetNextCustomer` should be `GetCustomerNext`

Scoping rules for procedures are rather inconsistent in VB. Procedures are global in modules unless they are declared Private; they are always local in forms in VB3, or default to Private in VB4 but can be Public if explicitly declared that way. Does the term "dog's breakfast" mean anything?

Because the event procedures cannot be renamed and do not have scope prefixes, user procedures in VB also should NOT include a scope prefix. Common modules should keep all procedures that are not callable from other modules Private.

Function Procedure Data Types

Function procedures can be said to have a data type which is the type of their return value. This is an important attribute of a function because it affects how and where it may be correctly used.

Therefore, function names should be prefixed with a data type tag like variables.

Parameters

Within the body of a procedure, parameter names have a very special status. The names themselves are local to the procedure but the memory that the name relates too may not be. This means that changing the value of a parameter may have an effect at a totally different scope to the procedure itself and this can be the cause of bugs that are particularly difficult to track down later. The best solution is to stop it from happening in the first place.

Ada has a really neat language facility whereby all paramaters to a procedure are tagged as being one of the types In, Out or InOut. The compiler then enforces these restrictions on the procedure body; it is not allowed to assign from an Out

parameter or assign to an In parameter. Unfortunately, no mainstream language supports this facility so we need to kludge it.

Always make sure you are absolutely clear about how each parameter is to be used. Is it used to communicate a value to the procedure or to the caller or both? Where possible, declare input parameters ByVal so that the compiler enforces this attribute. Unfortunately, there are some data types that cannot be passed ByVal, notably arrays and UDTs.

Each parameter name is formed according to the rules for forming variable names. Of course, parameters are always at procedure level (local) scope so there will not be a scope character. At the end of each parameter name, add an underscore followed by one of the words IN, OUT or INOUT as appropriate. Use upper case to really make these stand out in your code. Here is an example:

```
Sub GetCustomerSurname(ByVal nCustomerCode_IN as Long, _
                                  sCustomerSurname_OUT As String)
```

If you see an assignment to a variable that ends in _IN (ie if it is to the left of the '=' in an assignment statement) then you probably have a bug. Ditto if you see a reference to the value of a variable that ends in _OUT. Both these statements are extremely suspect:

```
nCustomerCode_IN = nSomeVariable
nSomeVariable = nCustomerCode_OUT
```

Function Return Values

In VB, you specify the return value of a function by assigning a value to a pseudo-variable with the same name as the function. This is a fairly common construct for many languages and generally works OK.

There are, however, a couple of limitations imposed by this scheme. The first is that it makes it really hard to copy and paste code from one function into another because you have to change references to the original function name. The second is that it is not possible to reference the currently-assigned return value from within the body of the function itself.

Always declare a local variable called Result inside every function procedure. Make sure that you assign a default value to that variable at the very beginning of the function. At the exit point, assign the Result variable to the function name immediately before the exit of the function. Here is a skeleton for a function procedure (minus comment block for brevity):

```
Function DoSomething() As Integer

   Dim Result As Integer
   Result = 42 ' Default value of function

   On Error Goto DoSomething_Error
   ' body of function
DoSomething_Exit:   DoSomething = Result   Exit Function
DoSomething_Error:    ' handle the error here   Resume DoSomething_Exit

End Function
```

In the body of the function, you should always assign the required return value to Result. You are also free to check the current value of Result.

> *This might not sound like a big deal if you have never had the ability to use it but once you try it you will not be able to go back to working without it. Trust me on this one, I'm from GUI and here to help you.*

Don't forget that functions have a data type too, so their name should be prefixed in the same way as a variable.

Constants

The formatting rules for constants are going to be very difficult for us all. This is because Microsoft has done an about-face on constant naming conventions. There are two formats used by Microsoft for constants and unfortunately we will need to work with both formats. While it would be possible for someone to work through the file of MS-defined constants and convert them to the new format, that would mean that every article, book and published code fragment would not match our conventions.

A vote failed to resolve the issue - it was pretty much evenly split between the two alternatives - so I have had to make an executive decision which I will try to explain. The decision is to go with the old-style constant formats.

Constants are coded in `ALL_UPPER_CASE` with words separated by underscores. Where possible, use the constant names defined in the Constant.Txt file - not because they are particularly well-formed or consistent but because they are what you will see in the documentation and in books and magazine articles.

The formatting of constants may well change as the world moves to VB4 and later, where constants are exposed by OLE objects through a type library and where the standard format is to use InitialCapitalLetters prefixed with some unique, lower-case tag (as in vbYesNoCancel).

When writing modules that are to be called from various places (especially if it will be used in several programs), define global constants that can be used by the client code instead of magic numbers or characters. Use them internally too, of course. When defining such constants, make up a unique prefix to be added to each of them. For example, if I am developing a Widget control module, I might define constants like this:

```
Global Const WDGT_STATUS_OK = 0
Global Const WDGT_STATUS_BUSY = 1
Global Const WDGT_STATUS_FAIL = 2
Global Const WDGT_STATUS_OFF = 3
Global Const WDGT_ACTION_START = 1
Global Const WDGT_ACTION_STOP = 2
Global Const WDGT_ACTION_RAISE = 3
Global Const WDGT_ACTION_LOWER = 4
```

You get the idea. The logic behind this is to avoid naming clashes with constants defined for other modules.

Constants must indicate scope and data type in one of two ways - they may take the lower-case scope and type prefixes that are used for variables, or they may take upper-case group-specific tags. The widget constants above demonstrate the latter type: the scope is `WDGT` and the data type is `STATUS` or `ACTION`. Constants that are used to hide implementation details from clients of a common module would generally use the latter type, whereas constants used for convenience and maintainability in the main part of a program would generally use the normal variable-like prefixes.

> *I feel I need to explain these decisions a little more, which I guess signifies that I am not totally satisfied with this outcome. While it would be nice to think that we could just go ahead and define our own standard, the fact is that we are part of the wider VB programming community and will need to interface to code written by others (and in particular by Microsoft). I therefore needed to keep in mind what others have done and what they are likely to do in the future.*
>
> *The decision to adopt the old-style comments (ie ALL_UPPER_CASE) was based on the fact that there is a huge body of code (and in particular modules containing constant definitions) that we need to incorporate into our programs. Microsoft's new format (the vbThisIsNew format) is only being implemented in type libraries. The VB4 documentation suggests that we adopt the lower-case prefix "con" for application-specific constants (as in conThisIsMyConstant) but in several on-line discussions*

with developers working in corporations all around the world it appears that this is not being adopted, or at least not for the foreseeable future.

That's basically why I decided on the old-style comments. We are familiar with them, everyone else is using them and they clearly delineate code that is in the program (whether we wrote it or it was included from a vendor's supplied BAS file) from constants published by a type library and referenced in VB4+.

The other dilemma was the issue of scope and type tags on constants. Most people wanted these although they are not generally defined in vendor-supplied constant definitions. I'm not too sure about the type tag myself as I think it is sometimes useful to hide the type to make the client code less dependent on the implementation of some common code.

In the end, a compromise was chosen. For constants where the type is important because they are used to assign values to a program's own data items, the normal variable-style tags are used to denote both scope and type. For 'interface' constants, such as those supplied by vendors to interface to a control, the scope can be considered the control, and the data type is one of the special types supported by a particular property or method of that control. In other words, MB_ICONSTOP has a scope of MB (message box) and a data type of ICON. Of course, I think it should have been MB_ICON_STOP and that MB_YESNOCANCEL should have been MB_BUTTONS_YESNOCANCEL, but you can't ever accuse Microsoft of consistency. I hope this sheds some light on why I decided to go this way.

I suspect this issue will generate further discussion as we all get some experience in applying this standard.

Controls

All controls on a form should be renamed from their default Text*n* names, even for the simplest of forms.

The only exceptions to this are any labels used only as static prompts on the surface of a form. If a label is referenced anywhere in code, even once, then it must be given a meaningful name along with the other controls. If not, then their names are not significant and can be left to their default "Label*n*" names. Of course, if you are really keen, you may give them meaningful names but I really think we are all busy enough that we can safely skip this step. If you don't like to leave those names set to their defaults (and I'll confess to being in that group), you may find the following technique useful. Create a control array out of all the inert labels on a form and call the array lblPrompt( ). The easy way to do this is to create the first label the way you want it (with the appropriate font, alignment and so on) and then copy and paste it as many times as is necessary to create all the labels. Using a control array has an additional benefit because a control array uses up just one name in the form's name table.

Take the time to rename all controls before writing any code for a form. This is because the code is attached to a control by its name. If you write some code in an event procedure and then change the name of the control, you create an orphan event procedure.

Controls have their own set of prefixes. They are used to identify the type of control so that code can be visually checked for correctness. They also assist in making it easy to know the name of a control without continually needing to look it up. (See "Cradle to Grave Naming of Data Items" below.)

Specifying Particular Control Variants - NOT

In general, it is NOT a good idea to use specific identifiers for variations on a theme. For example, whether you are using a

ThreeD button or a standard button generally is invisible to your code - you may have more properties to play with at design time to visually enhance the control, but your code usually traps the Click() event and maybe manipulates the Enabled and Caption properties, which will be common to all button-like controls.

Using generic prefixes means that your code is less dependent on the particular control variant that is used in an application and therefore makes code re-use simpler. Only differentiate between variants of a fundamental control if your code is totally dependent on some unique attribute of that particular control. Otherwise, use the generic prefixes where possible.

Table of Standard Control Prefixes

The following table is a list of the common types of controls you will encounter together with their prefixes:

Prefix	Control
cbo	Combo box
chk	Checkbox
cmd	Command button
dat	Data control
dir	Directory list box
dlg	Common dialog control
drv	Drive list box
ela	Elastic
fil	File list box
fra	Frame
frm	Form
gau	Gauge
gra	Graph
img	Image
lbl	Label
lin	Line
lst	List box
mci	MCI control
mnu	Menu control †
mpm	MAPI Message
mps	MAPI Session
ole	OLE control
opt	Option button
out	Outline control
pic	Picture
pnl	Panel
rpt	Report
sbr	Scroll bar (there is no need to distinguish orientation)
shp	Shape

spn	Spin
ssh	Spreadsheet control
tgd	Truegrid
tmr	Timer ‡
txt	Textbox

† Menu controls are subject to additional rules as defined below.

‡ There is often a single Timer control in a project, and it is used for a number of things. That makes it difficult to come up with a meaningful name. In this situation, it is acceptable to call the control simply "Timer".

Menu Controls

Menu controls should be named using the tag "mnu" followed by the complete path down the menu tree. This has the additional benefit of encouraging shallow menu hierarchies which are generally considered to be A Good Thing in user interface design.

Here are some examples of menu control names:

```
mnuFileNew
mnuEditCopy
mnuInsertIndexAndTables
mnuTableCellHeightAndWidth
```

Cradle to Grave Naming of Data Items

As important as all the preceding rules are, the rewards you get for all the extra time thinking about the naming of objects will be small without this final step, something I call cradle to grave naming. The concept is simple but it is amazingly difficult to discipline yourself to do it without fail.

Essentially, the concept is simply an acknowledgment that any given data item has exactly one name. If something is called a CustomerCode, then that is what it is called EVERYWHERE. Not CustCode. Not CustomerID. Not CustID. Not CustomerCde. No name except CustomerCode is acceptable.

Now, let's assume a customer code is a numeric item. If I need a customer code, I would use the name nCustomerCode. If I want to display it in a text box, that text box must be called txtCustomerCode. If I want a combo box to display customer codes, that control would be called cboCustomerCode. If you need to store a customer code in a global variable (I don't know why you'd want to either - I'm just making a point) then it would be called gnCustomerCode. If you want to convert it into a string (say to print it out later) you might use a statement like:

```
sCustomerCode = Format$(gnCustomerCode)
```

I think you get the idea. It's really very simple. It's also incredibly difficult to do EVERY time, and it's only by doing it EVERY time that you get the real payoffs. It is just SO tempting to code the above line like this:

```
sCustCode = Format$(gnCustomerCode)
```

Fields in databases

As a general rule, the data type tag prefixes should be used in naming fields in a database.

This may not always be practical or even possible. If the database already exists (either because the new program is referencing an existing database or because the database structure has been created as part of the database design phase) then it is not practical to apply these tags to every column or field. Even for new tables in existing databases, do not deviate from the conventions (hopefully) already in use in that database.

Some database engines do not support mixed case in data item names. In that case, a name like SCUSTOMERCODE is visually difficult to scan and it might be a better idea to omit the tag. Further, some database formats allow for only very short names (like xBase's limit of 10 characters) so you may not be able to fit the tags in.

In general, however, you should prefix database field/column names with data type tags.

Objects

There are a number object types that are used often in VB. The most common objects (and their type tags) are:

Prefix	Object
db	Database
ws	Workspace
rs	Recordset
ds	Dynaset
ss	Snapshot
tbl	Table
qry	Query
tdf	TableDef
qdf	QueryDef
rpt	Report
idx	Index
fld	Field
xl	Excel object
wrd	Word object

Control Object Variables

Control objects are pointers to actual controls. When you assign a control to a control object variable, you are actually assigning a pointer to the actual control. Any references to the object variable then reference the control to which it points.

Generic control variables are defined as:

```
Dim ctlText As Control
```

This allows a pointer to any control to be assigned to it via the Set statement:

```
Set ctlText = txtCustomerCode
```

The benefit of generic control variables is that they can point to any type of control. You can test what type of control a generic control variable is currently pointing to by using the TypeOf statement:

```
If TypeOf ctlText Is Textbox Then
```

Be careful here; this only looks like a normal If statement. You cannot combine TypeOf with another test, nor can you use it in an Select statement.

The fact that the variable can point to any control type is also its weakness. Because the compiler does not know what the control variable will be pointing to at any point in time, it cannot check what you are doing for reasonableness. It must use run-time code to check every action on that control variable so those actions are less efficient, which just compounds VB's reputation for sluggish performance. More importantly, however, you can end up with the variable pointing to an unexpected control type, causing your program to explode (if you are lucky) or to perform incorrectly.

Specific control variables are also pointers. In this case however, the variable is constrained to point only to controls of a particular type. For example:

```
Dim txtCustomerCode as Textbox
```

The problem here is that I have used the same three-letter tag to denote a control object variable as I would have used to name an actual text control on a form. If I am following these guidelines, then I will have a control called by the same name on a form. How am I supposed to assign it to the variable?

I can use the form's name to qualify the reference to the actual control, but that is confusing at best. For that reason, object control variable names should be distinct from actual controls, so we extend the type code by using a single letter "o" at the front. The previous definition would more correctly be:

```
Dim otxtCustomerCode as Textbox
```

This is another one I had to agonise over. I actually like the name better if the "o" follows the tag, as in `txtoCustomerName`. However, it is just too hard to see that little "o" buried in all those characters. I also considered using `txtobjCustomerName`, but I think that is getting a little bit too long. I'm happy to look at any other alternatives or arguments for or against these ones.

Actually, this is not as big a problem as you might imagine, because in most cases these variables are used as parameters to a generic function call. In that case, we will often use the tag alone as the name of the parameter so the issue does not arise.

A very common use of control object variables is as parameters to a procedure. Here is an example:

```
Sub SelectAll(txt As TextBox)
   txt.SelStart = 0
   txt.SelLength = Len(txt.Text)
End Sub
```

If we place a call to this procedure in the GotFocus event of a textbox, that textbox has all its text highlighted (selected) whenever the user tabs to it.

Notice how we denote the fact that this is a generic reference to any textbox control by omitting the body of the name. This is a common technique used by C/C++ programmers too, so you will see these sorts of examples in the Microsoft documentation.

To show how generic object variables can be very useful, consider the case where we want to use some masked edit controls as well as standard text controls in our application. The previous version of the routine will not work for anything but standard textbox controls. We could change it as follows:

```
Sub SelectAll(ctl As Control)
   ctl.SelStart = 0
   ctl.SelLength = Len(ctl.Text)
End Sub
```

By defining the parameter to be a generic control parameter, we are now allowed to pass any control to this procedure. As long as that control has SelStart, SelLength and Text properties, this code will work fine. If it does not, then it will fail with a run-time error; the fact that it is using late binding means that the compiler cannot protect us. To take this code to production status, either add specific tests for all the known control types you intend to support (and only process those ones) or else add a simple error handler to exit gracefully if the control does not support the required properties:

```
Sub SelectAll(ctl As Control)

   On Error Goto SelectAll_Error
   ctl.SelStart = 0  ctl.SelLength = Len(ctl.Text)
SelectAll_Exit:
   Exit Sub
SelectAll_Error:
   Resume SelectAll_Exit
End Sub
```

API Declarations

If you don't know how to make an API declaration, see any one of several good books available on the subject. There is not much to say about API declarations; you either do it right, or it doesn't work.

However, one issue comes up whenever you try to use a common module that itself uses API calls. Inevitably, your application has made one of those calls too so you need to delete (or comment out) one of the declarations. Try to add several of these and you end up with a real mess; API declarations are all over the place and every addition or removal of a module sparks the great API declaration hunt.

There is a simple technique to get around this, however. All non-sharable code uses the standard API declarations as defined in the Help files. All common modules (those that will or may be shared across projects) is banned from using the standard declarations. Instead, if one of these common modules needs to use an API, it must create an aliased declaration for that API where the name is prefixed with a unique code (the same one used to prefix its global constants).

If the mythical Widgets module needed to use SendMessage, it *must* declare:

```
Declare Function WDGT_SendMessage Lib "User" Alias "SendMessage" _
                  (ByVal hWnd As Integer, ByVal wMsg As Integer, _
                   ByVal wParam As Integer, lParam As Any) As Long
```

This effectively creates a private declaration of SendMessage that can be included with any other project without naming conflicts.

Source Files

Do not start the names of the files with "frm" or "bas"; that's what the file extension is for! As long as we are limited to the 8-character names of Dos/Win16, we need every character we can get our hands on.

Create the file names as follows:

```
SSSCCCCV.EXT
```

where SSS is a three-character system or sub-system name, CCCC is the four-character form name code and V is an (optional) single digit that can be used to denote different versions of the same file. EXT is the standard file extension assigned by VB, either FRM/FRX, BAS or CLS.

The importance of starting with a system or sub-system name is that it is just too easy in VB to save a file in the wrong directory. Using the unique code allows us to find the file using directory search (sweeper) programs and also minimises the chance of clobbering another file that belongs to another program, especially for common form names like MAIN or SEARCH.

Reserving the last character for a version number allows us to take a snapshot of a particular file before making radical changes that we know we might want to back out. While tools like SourceSafe should be used where possible, there will inevitably be times when you will need to work without such tools. To snapshot a file, you just remove it from the project, swap to a Dos box or Explorer (or File Manager or whatever) to make a new copy of the file with the next version number and then add the new one back in. Don't forget to copy the FRX file as well when taking a snapshot of a form file.

Coding Standards

The rest of this document addresses issues relating to coding practices. We all know that there is no set of rules that can always be applied blindly and that will result in good code. Programming is not an art form but neither is it engineering. It is more like a craft: there are accepted norms and standards and a body of knowledge that is slowly being codified and formalised. Programmers become better by learning from their previous experience and by looking at good and bad code written by others. Especially by maintaining bad code.

Rather than creating a set of rules that must be slavishly followed, I have tried to create a set of principles and guidelines that will identify the issues you need to think about and where possible indicate the good, the bad and the ugly alternatives.

Ultimately, I want each programmer at GUI to take responsibility for creating good code, not simply code that adheres to some rigid standard. That is a far nobler goal - one that is both more fulfilling to the programmer and more useful to the organisation.

The underlying principle is to keep it simple.
Eschew obfuscation.
I've always wanted to use that term.

Procedure Length

There has been an urban myth in programming academia that short procedures of no more than "a page" (whatever that is) are better. Actual research has shown that this is simply not true. There have been several studies that suggest the exact opposite. For a review of these studies (and pointers to the studies themselves) see the book Code Complete by Steve McConnell (Microsoft Press, ISBN 1-55615-484-4) which is a book well worth reading. Three times.

To summarise, hard empirical data suggests that error rates and development costs for routines decreases as you move from small (<32 lines) routines to larger routines (up to about 200 lines). Comprehensibility of code (as measured on computer-science students) was no better for code super-modularised to routines about 10 lines long than one with no routines at all. In contrast, on the same code modularised to routines of around 25 lines, students scored 65% better.

What this means is that there is no sin in writing long routines. Let the requirements of the process determine the length of

the routine. If you feel that this routine should be 200 lines long, just do it. Be careful how far you go, of course. There *is* an upper limit beyond which it is almost impossible to comprehend a routine. Studies on really BIG software, like IBM's OS/360 operating system, showed that the most error prone routines were those over 500 lines, with the rate being roughly proportional to length above this figure.

Of course, a procedure should do ONE thing. If you see an "And" or an "Or" in a procedure name, you are probably doing something wrong. Make sure that each procedure has high cohesion an low coupling, the standard aims of good structured design.

IF

Write the nominal path through the code first, then write the exceptions. Write your code so that the normal path through the code is clear. Make sure that the exceptions don't obscure the normal path of execution. This is important for both maintainability and efficiency.

Make sure that you branch correctly on equality. A very common mistake is to use > instead of >= or vice versa.

Put the normal case after the If rather than after the Else. Contrary to popular thought, programmers really do not have a problem with negated conditions. Create the condition so that the Then clause corresponds to normal processing.

Follow the If with a meaningful statement. This is somewhat related to the previous point. Don't code null Then clauses just for the sake of avoiding a Not. Which one is easier to read:

```
If EOF(nFile) Then
   ' do nothing
Else
   ProcessRecord
End IF
```

```
If Not EOF(nFile) Then
   ProcessRecord
End If
```

Always at least consider using the Else clause. A study of code by GM showed that only 17% of If statements had an Else clause. Later research showed that 50 to 80% should have had one. Admittedly, this was PL/1 code and 1976, but the message is ominous. Are you *absolutely* sure you don't need that Else clause?

Simplify complicated conditions with boolean function calls. Rather than test twelve things in an If statement, create a function that returns True or False. If you give it a meaningful name, it can make the If statement very readable and significantly improve the program.

Don't use chains of If statements if a Select Case statement will do. The Select Case statement is often a better choice than a whole series of If statements. The one exception is when using the TypeOf condition which does not work with Select Case statements.

SELECT CASE

Put the normal case first. This is both more readable and more efficient.

Order cases by frequency. Cases are evaluated in the order that they appear in the code, so if one case is going to be selected 99% of the time, put it first.

Keep the actions of each case simple. Code no more than a few lines for each case. If necessary, create procedures

called from each case.

Use the Case Else only for legitimate defaults. Don't ever use Case Else simply to avoid coding a specific test.

Use Case Else to detect errors. Unless you really do have a default, trap the Case Else condition and display or log an error message.

> *When writing any construct, the rules may be broken if the code becomes more readable and maintainable. The rule about putting the normal case first is a good example. While it is good advice in general, there are some cases (if you'll pardon the pun) where it would detract from the quality of the code. For example, if you were grading scores, so that less than 50 was a fail, 50 to 60 was an E and so on, then the 'normal' and more frequent case would be in the 60-80 bracket, then alternating like this:*
>
> ```
> Select Case iScore
>   Case 70 To 79:   sGrade = "C"
>   Case 80 To 89:   sGrade = "B"
>   Case 60 To 69:   sGrade = "D"
>   Case Is < 50:    sGrade = "F"
>   Case 90 To 100:  sGrade = "A"
>   Case 50 To 59:   sGrade = "E"
>   Case Else:     ' they cheated
> End Select
> ```
>
> *However, the natural way to code this would be do follow the natural order of the scores:*
>
> ```
> Select Case iScore
>   Case Is < 50:   sGrade = "F"
>   Case Is < 60:   sGrade = "E"
>   Case Is < 70:   sGrade = "D"
>   Case Is < 80:   sGrade = "C"
>   Case Is < 90:   sGrade = "B"
>   Case Is <= 100: sGrade = "A"
>   Case Else:     ' they cheated
> End Select
> ```
>
> *Not only is this easier to understand, it has the added advantage of being more robust - if the scores are later changed from integers to allow for fractional points, then the first version would allow 89.1 as an A which is probably not what was expected.*
>
> *On the other hand, if this statement was identified as being the bottleneck in a program that was not performing quickly enough, then it would be quite appropriate to order the cases by their statistical probability of occurring, in which case you would document why you did it that way in comments.*
>
> *This discussion was included to reinforce the fact that we are not seeking to blindly obey rules - we are*

> *trying to write good code. The rules must be followed unless they result in bad code, in which case they must not be followed.*

DO

Keep the body of a loop visible on the screen at once. If it is too long to see, chances are it is too long to understand buried inside that loop and should be taken out as a procedure.

Limit nesting to three levels. Studies have shown that the ability of programmers to understand a loop deteriorates significantly beyond three levels of nesting.

FOR

See DO above.

Never omit the loop variable from the Next statement. It is very hard to unravel loops if their end points do not identify themselves properly.

Try not to use *i*, *j* and *k* as the loop variables. Surely you can come up with a more meaningful name. Even if it's something generic like iLoopCounter it is better than *i*.

> *This is a suggestion only. I know how convenient single-character loop variable names are when they are used for a number of things inside the loop, but do think about what you are doing and the poor programmer who has to figure out what i and j actually equate to.*

Assignment Statements (Let)

Do not code the optional Let key-word for assignment statements. (This means you, Peter.)

GOTO

Do not use Goto statements unless they make the code simpler. The general consensus of opinion is that Goto statements tend to make code difficult to follow but that in some cases the exact opposite is true.

In VB, you have to use Goto statements as an integral part of error handling, so there is really no choice about that.

You may also want to use Goto to exit from a very complex nested control structure. Be careful here; if you really feel that a Goto is warranted, perhaps the control structure is just too complex and you should decompose the code into smaller routines.

That is not to say that there are no cases where the best approach is to use a Goto. If you really feel that it is necessary then go ahead and use one. Just make sure that you have thought it through and are convinced that it really is a good thing and not a hack.

EXIT SUB and EXIT FUNCTION

Related to the use of a Goto is an Exit Sub (or Exit Function) statement. There are basically three ways to make some trailing part of the code not execute:

- Make it part of a conditional (If) statement:

```
Sub DoSomething()

  If CanProceed() Then

     . . .

     . . .

  End If

End Sub
```

- Jump over it with a Goto

```
Sub DoSomething()

  If Not CanProceed() Then

     Goto DoSomething_Exit

  End If

  . . .

  . . .

DoSomething_Exit:

End Sub
```

- Exit prematurely with an Exit Sub/Function

```
Sub DoSomething()

  If Not CanProceed() Then

     Exit Sub

  End If

  . . .

  . . .
```

```
End Sub
```

The one that seems to be the clearest in these simple, skeletal examples is the first one and it is in general a good approach to coding simple procedures. This structure becomes unwieldy when the main body of the procedure (denoted by the '…' above) is nested deep inside a series of control structures required to determine whether that body should be executed. It is not at all difficult to end up with the main body indented half way across the screen. If that main body is itself complex, the code looks quite messy, not to mention the fact that you then need to unwind all those nested control structures after the main body of the code.

When you find this happening in your code, adopt a different approach: determine whether you should proceed and, if not, exit prematurely. Both of the other techniques shown work. Although I think that the Exit statement is more 'elegant', I am forced to mandate the use of the Goto ExitLabel as the GUI standard. The reason that this was chosen is that sometimes you need to clean up before exiting a procedure. Using the Goto ExitLabel construct means that you can code that cleanup code just once (after the label) instead of many times (before each Exit statement).

If you need to prematurely exit a procedure, prefer the Goto ExitLabel construct to the Exit Sub or Exit Function statements unless there is no chance that any cleanup will be needed before those statements.

One case where the Exit statement is very much OK is in conjunction with gatekeeper variables to avoid unwanted recursion. You know:

```
Sub txtSomething_Change()
   Dim bBusy As Integer
   If bBusy Then Exit Sub
      bBusy = True
   . . . ' some code that may re-trigger the Change() event
   bBusy = False
Exit Sub
```

EXIT DO/FOR

These statements prematurely bail out of the enclosing Do or For loop. Do use these when appropriate but be careful because they can make it difficult to understand the flow of execution in the program.

On the other hand, use these statements to avoid unnecessary processing. We always code for correctness and maintainability rather than efficiency, but there is no point doing totally unnecessary processing. In particular, do NOT do this:

```
For nIndex = LBound(sItems) to UBound(sItems)
    If sItems(nIndex) = sSearchValue Then
        bFound = True
        nFoundIndex = nIndex
    End If
Next nIndex

If bFound Then . . .
```

This will always loop through all elements of the array, even if the item is found in the first element. Placing an Exit For statement inside the If block would improve performance with no loss of clarity in the code.

Do avoid the need to use these statements in deeply nested loops. (Indeed, avoid deeply nested loops in the first place.) Sometimes there really is no option, so this is not a hard and fast rule, but in general it is difficult to determine where an Exit For or Exit Do will branch to in a deeply nested loop.

GOSUB

The Gosub statement is not often used in VB and with good reason. In most cases it does not offer any advantages over the use of standard Sub procedures. Indeed, the fact that the routine that is executed by a Gosub is actually in the same scope as the calling code is generally a dangerous situation and should be avoided.

However, this very same attribute is sometimes very useful. If you are writing a complex Sub or Function procedure you would generally try to identify discrete units of processing that can be implemented as separate procedures that this one can call as appropriate. In some particular cases, however, you will find that the amount of data that needs to be passed around between these related procedures becomes quite absurd. In those cases, implementing those subordinate routines as subroutines inside the main procedure can significantly simplify the logic and improve the clarity and maintainability of the code.

The particular situation where I have found this technique useful is in creating multi-level reports. You end up with procedures like ProcessDetail, ProcessGroup, PrintTotals, PrintSubTotals, PrintHeadings and so forth, all of which need to access and modify a common pool of data items like the current page and line number, the current subtotals and grand totals, the current and previous key values (and a whole lot more I can't think of right now).

For situations like this, the GUI standard does permit the use of Gosub statements when appropriate. However, be careful when you decide to use this approach. Treat the enclosing Sub or Function as if it was a single COBOL program; that is essentially what it is from a control and scope point of view. In particular, ensure that there is only one Return statement at the end of every subroutine. Also be sure that there is an Exit Sub or Exit Function statement before all the subroutine code to avoid the processing falling through into that code (which is highly embarrassing).

Procedure Calls

Since the days of QuickBASIC for Dos, there has been a raging debate about whether you should use the 'Call' keyword for non-function procedure calls. In the final analysis, it is one of those issues for which there is no definitive 'right' way to do this.

The GUI standard is to NOT use the Call keyword.

> *My reasoning for this is that the code reads more naturally. If we have well-named procedures (using the verb.noun.adjective format) then the resulting VB code is almost pseudocode. Further, I think there is no need to code a keyword if it is obvious what is going on, just like omitting the Let keyword in assignment statements.*

Procedure Parameters

One of the problems with using procedures is remembering the order of parameters. You want to try to avoid the need to constantly look at the procedure definition to determine what order the arguments need to be supplied to match the parameters.

Here are some guidelines to try to address this issue.

First, code all input parameters before all output parameters. Use your judgement about InOut parameters. Usually, you will not have all three types in a single procedure; if you do I would probably code all the input parameters, followed by the InOut and ending with the output. Again, use your judgement on this.

Don't write procedures that need scores of parameters. If you do need to pass a lot of information to and from a procedure, then create a user defined type that contains the required parameters and pass that instead. This allows the calling program to specify the values in any convenient order by "setting" them in the UDT. Here is a rather trivial example that demonstrates the technique:

```
Type TUserMessage
   sText          As String
   sTitle         As String
   nButtons       As Integer
   nIcon          As Integer
   nDefaultButton As Integer
   nResponse      As Integer
End Type

Sub Message (UserMessage As TUserMessage)
'  << comment block omitted for brevity >>
UserMessage.Response = MsgBox(UserMessage.sText,_
                              UserMessage.nButtons Or _
                              UserMessage.nIcon Or _
                              UserMessage.nDefaultButton, _
                              UserMessage.sTitle)
End Sub
```

Now here is the calling code:

```
Dim MessageParms As TUserMessage

  MessageParms.sText = "Severe error in some module."
  MessageParms.sTitle = "Error"

  MessageParms.nButtons = MB_YESNOCANCEL
  MessageParms.nIcon = MB_ICONSTOP
  MessageParms.sDefaultButton = MB_DEFBUTTON1

  Message MessageParms

  Select Case MassageParms.nResponse
     Case ID_YES:
     Case IS_NO:
     Case ID_CANCEL:
  End Select
```

Error Handling

The ideal situation is to have an error handler around every bit of code and that should be the starting point. This is not always achievable in practice for all sorts of reasons and it is not *always* necessary. The rule is, however, that unless you are absolutely sure that a routine does not need an error handler, you should at the very least create a handler around the entire routine, something like this:

> *Note that the next section discusses the GUI Default Error Handler which requires an expanded form of this skeleton. Please refer to that section for a more complete skeleton for a procedure with an error handler. It has not been included here because we are looking at other components of the standard in this section.*

```
Sub DoSomething()
  On Error Goto DoSomething_Error
```

```
  . . .
DoSomething_Exit:
  Exit Sub
DoSomething_Error:
  Resume DoSomething_Exit
End Sub
```

Always create the two labels by appending "`_Exit`" and "`_Error`" to the name of the procedure. I would have much preferred to use the same labels in all procedures but unfortunately line labels have (at least) module level scope in VB so this is not possible.

Within the body of the procedure, you may create regions where different error handling is required. To put it bluntly, VB stinks in this area. The technique I prefer is to temporarily disable error trapping and to test for errors using in-line code, something like this:

```
DoThingOne
DoThingTwo

On Error Resume Next

   DoSomethingSpecial

   If Err Then
      HandleTheLocalError
   End If

   On Error Goto ThisProcedureName_Error
```

Another technique is to set flag variables to indicate where you are up to in the body of the procedure and then to write logic in the error handler to take different action based on where the error occurred. I do not generally like this approach but confess that I have used it on occasion because I just could not figure out a way I liked better.

If you get the feeling that I am not too impressed with VB's error handling mechanisms, you are absolutely correct. For that reason, I am not going to mandate any particular style of error handling other than to say that you should implement sufficient error handling in your application to ensure that it works correctly and reliably. If you implement an error handler in a procedure, use the naming convention as shown previously for the line labels.

Standard Error Handler

In the fullness of time GUI will develop a standard error handler that will be able to handle the vast majority of errors in a generic way. The idea is to have this as a sort of generic error handler that individual error-handling routines can call to handle many of the common errors. In the meantime, a standard error handling mechanism framework has been created so that we can start wiring our programs now to use the enhanced version later.

There are two modules than need to be included in every program. They are `ERRORGEN.BAS` and `ERRORAPP.BAS`. The `ERRORGEN` module is not to be modified for any application; it is the module that will be enhanced over time and it is important that we can make changes to this module centrally without breaking any progams. The `ERRORAPP` module contains application-specific constants and can be modified to customise the error-handling routines for individual applications.

These two modules can be found in the Projects share (in the Code Library\Error Handler directory).

Using the standard error handler involves the following steps:

- add `ERRORGEN.BAS` and `ERRORAPP.BAS` to a project
- call DefErrorHandler from your error handling routines

This is the standard way to do this:

```
Sub DoSomething()

   On Error Goto DoSomething_Error
   . . .
DoSomething_Exit:
   Exit Sub

DoSomething_Error:
   Select Case Err
      Case xxx:
      Case yyy:
      Case xxx:
      Case Else: DefErrorHandler Err, "DoSomething",_
                                 "", ERR_ASK_ABORT_
                                 Or ERR_DISPLAY_MESSAGE
   End Select

   Resume DoSomething_Exit

End Sub
```

Note the Select Case statement used to trap errors that will be handled locally. All expected errors that need to be handled locally will have a corresponding Case statement. The individual cases may choose to use a different form of the Resume statement to control the flow of the program. Any handled errors that do not do a Resume will fall through to the final statement that will cause the procedure to exit gracefully.

Any errors that have not been handled will be passed on to DefErrorHandler. Individual cases may also call upon DefErrorHandler to log the message and display a user alert.

The first parameter to DefErrorHandler is the error number that has been trapped. The second parameter is a string to denote where the error occurred. This string is used in message boxes and in the log file. The third parameter is a message that is to be written to the log file and displayed to the user. The fourth parameter is a set of flags indicating what the procedure should do. See the constants defined in `ERRORGEN` for more information.

`ERRORGEN` also provides exits in which you can further customise the way the generic error handler works without modifying the code. These take the form of procedures defined in `ERRORAPP` that are called at convenient places in the processing cycle. By customising how these procedures operate you can change the appearance and function of the error handling process.

These are the constants and procedures you may want to modify:

`gsERROR_LogFileName`

: a constant that defines the root file name for the log file

`gsERROR_LogFilePath`

: a constant that defines the path for the log file can be left empty to use App.Path or can be changed to a variable and set during program initialisation to a valid path

`gsERROR_MSG_???`

: a series of constants that define some strings, buttons and icons for default message boxes.

`gsERROR_LOGFORMAT_???`

: a series of constants that define the delimiters in and the format of the lines in the log file

`sERROR_GetLogLine`

: a function that can totally redefine the format of the log line

`sERROR_GetLogFileName`

a function that returns the complete path to the log file; this function can be customised to allow non-fixed locations for the log file to be handled (see the comments in the function)

`ERROR_BeforeAbort`

a sub that is called just before the program is terminated with an End statement; this can be used to do any last-minute clean-up in an error situation

`bERROR_FormatMessageBox`

a function that can do custom formatting of the error message box or can replace the message box altogether with custom processing (eg. using a form to gather additional information from the user before continuing or ending)

It is recommended that you peruse the source code to see how all this works. The `ERRORAPP` module is commented specifically to help identify where modifications can be made. `ERRORGEN` must not be modified, but it is worth looking at to see how it has been architected. Feedback is strongly encouraged! The more we improve this module, the less work error handling will be in future projects.

Formatting Standards

The physical layout of code is very important in determining how readable and maintainable it is. We need to come up with a common set of conventions for code layout to ensure that programs incorporating code from various sources is both maintainable and aesthetically pleasing.

These guidelines are not hard and fast rules, less so than the variable naming conventions already covered. As always, use your own judgement and remember that you are trying to create the best possible code, not slavishly follow rules.

That said, you'd better have a good reason before deviating from the standard.

White Space and Indentation

Indent three spaces at a time. I know that the default editor tab width is four characters, but I prefer three for a couple of reasons. I don't want to indent too much, and I think two spaces is actually more conservative of screen real estate in the editor. The main reason that I chose three is that the most common reasons to indent (the If and Do statements) look good when their bodies line up with the first character of the word following the keyword:

```
If nValueCur = nValueMax Then
   MsgBox . . .
End If
Do While nValueCur <= nValueMax
   Print nValueCur
   nValueCur = nValueCur + 1
Loop
```

Unfortunately, the other two prime reasons to indent (the For and Select statements) do not fit neatly into this scheme.

When writing a For statement, the natural inclination is to indent four characters:

```
For nValueCur = nValueMin To nValueMax
    MsgBox . . .
Next nValueCur
```

I'm not too fussed about this as long as the For is not too long and does not encase further indentation. If it is long and if it does contain more indentation, it is hard to keep the closing statements aligned. In that case, I would strongly suggest that you stick to indenting by the standard three spaces.

Don't forget that you can set up the VB editor so that the Tab key indents by three spaces if you prefer to use the Tab key. Personally, I use the space bar.

For the Select Case statement, there are two commonly used techniques, both of which are valid.

In the first technique, the Case statements are not indented at all but the code that is controlled by each statement is indented by the standard amount of three spaces, as in:

```
Select Case nCurrentMonth

Case 1,3,5,7,8,10,12
   nDaysInMonth = 31

Case 4,6,9,11
   nDaysInMonth = 30

Case 2
   If IsLeapYear(nCurrentYear) Then
      nDaysInMonth = 29
   Else
      nDaysInMonth = 28
   End If

Case Else
   DisplayError "Invalid Month"
End Select
```

In the second technique, the Case statements themselves are indented by a small amount (two or three spaces) and the statements they control are super-indented, essentially suspending the rules if indentation:

```
Select Case nCurrentMonth
  Case 1,3,5,7,8,10,12:  nDaysInMonth = 31
  Case 4,6,9,11:         nDaysInMonth = 30
  Case 2                 If IsLeapYear(nCurrentYear) Then
                            nDaysInMonth = 29
                         Else
                            nDaysInMonth = 28
                         End If
  Case Else:             DisplayError "Invalid Month"

End Select
```

Notice how the colon is used to allow the statements to appear right beside the conditions. Notice also how you cannot do this for If statements.

Both techniques are valid and acceptable. In some sections of a program, one or the other will be clearer and more maintainable, so use common sense when deciding.

Let's not get too hung up on indentation, folks. Most of us understand what is acceptable and what is not when it comes to indenting code.

Commenting Code

This one will be really controversial. Don't comment more than you need to. Now, here's a list of where you definitely need to; there may be others too.

Comment Header Block

Each major routine should have a header that identifies:

- who wrote it
- what it is supposed to do
- what the parameters mean (both input and output)
- what it returns (if it's a function)
- what assumptions it makes about the state of the program or environment
- any known limitations
- an amendment history

Here is an example of a function's header:

```
Function ParseToken(sStream_INOUT As String, _
                    sDelimiter_IN As String) As String
'=================================================================
'ParseToken
'-----------------------------------------------------------------
'Purpose : Removes the first token from sStream and returns it
'          as the function result. A token is delimited by the
'          first occurrence of sDelimiter in sStream.
'
' Author : Jim Karabatsos, March 1996
'
' Notes   : sStream is modified so that repeated calls to this
'           function will break apart the stream into tokens
'-----------------------------------------------------------------
' Parameters
'-----------
' sStream_INOUT : The stream of characters to be scanned for a
'                 token.  The token is removed from the string.
'
' sDelimiter_IN : The character string that delimits tokens
'-----------------------------------------------------------------
' Returns : either a token (if one is found) or an empty string
'           if the stream is empty
'-----------------------------------------------------------------
'Revision History
'-----------------------------------------------------------------
' 11Mar96 JK  : Enhanced to allow multi-character delimiters
' 08Mar96 JK  : Initial Version
'=================================================================
```

Looks like a lot, doesn't it? The truth of the matter is you will not write one of these for every procedure. For many procedures, a few lines of comments at the top is all you need to convey all the information required. For event procedures you generally do not need such a comment block at all unless it contains significant amounts of code.

For significant procedures, however, until you can write one of these you haven't thought the processing through enough to start coding. If you don't know what's in that header, you can't start maintaining that module. Nor can you hope to reuse it. So you may as well type it into the editor before you write the code and pass on your knowledge rather than require someone else to read through the code to decipher what is going on.

Notice that the header does not describe how the function does its job. That's what the source code is for. You don't want to have to maintain the comments when you later change the implementation of a routine. Notice also that there are no closing characters at the end of each comment line. DO NOT DO THIS:

```
'*************************************************
'* MY COMMENT BLOCK                              *
'*                                               *
'* This is an example of a comment block that is *
'* almost impossible to maintain. Don't do it !!! *
'*************************************************
```

This might look 'nice' (although that really is debatable) but it is very hard to maintain such formatting. If you've got time to do this sort of thing, you obviously aren't busy enough. You can't be a GUI employee.

In general terms, naming variables, controls, forms and procedures sensibly, combined with a well-thought-out comment block, will be sufficient commenting for most procedures. Remember, you are not trying to explain your code to a non-programmer; assume that the person looking at the code knows VB pretty well.

In the body of the procedure, there are two types of comments that you will use as required: In-line comments and End of Line Comments.

In-line Comments

In-line comments are comments that appear by themselves on a line, whether they are indented or not.

In-line comments are the Post-It notes of programming. This is where you make annotations to help yourself or another programmer who needs to work with the code later. Use In-line comments to make notes in your code about

- what you are doing
- where you are up to
- why you have chosen a particular option
- any external factors that need to be known

Here are some examples of appropriate uses of In-line comments:

What we are doing:

```
' Now update the control totals file to keep everything in sync
```

Where we are up to:

```
' At this point, everything has been validated.
' If anything was invalid, we would have exited the procedure.
```

Why we chose a particular option:

```
' Use a sequential search for now because it's simple to code
' We can replace with a binary search later if it's not fast
' enough
' We are using a file-based approach rather than doing it all
' in memory because testing showed that the latter approach
' used too many resources under Win16.  That's why the code
' is here rather than in ABCFILE.BAS where it belongs.
```

External factors that need to be kept in mind:

```
' This assumes that the INI file settings have been checked by
' the calling routine
```

Notice that we are not documenting what is self-evident from the code. Here are some examples of inappropriate In-line comments:

```
 ' Declare local variables
Dim nEmployeeCur As Integer
 ' Increment the array index
```

```
nEmployeeCur = nEmployeeCur + 1
' Call the update routine
UpdateRecord
```

Comment what is not readily discernible from the code. Do not re-write the code in English, otherwise you will almost certainly not keep the code and the comments synchronised and *that is very dangerous.* The reverse side of the same coin is that when you are looking at someone else's code *you should totally ignore any comments that relate directly to the code statements.* In fact, do everyone a favour and remove them.

End of Line Comments

End of Line (EOL) comments are small annotations tacked on the end of a line of code. The perceptual difference between EOL comments and In-Line comments is that EOL comments are very much focused on one or a very few lines of code whereas In-Line comments refer to larger sections of code (sometimes the whole procedure).

Think of EOL comments like margin notes in a document. Their purpose is to explain why something needs to be done or why it needs to be done now. They may also be used to document a change to the code. Here are some appropriate EOL comments:

```
mnEmployeeCur = mnEmployeeCur + 1     ' Keep the module level
                                      ' pointer synchronised
                                      ' for OLE clients

mnEmployeeCur = nEmployeeCur          ' Do this first so that the
UpdateProgress                        ' meter ends at 100%

If nEmployeeCur < mnEmployeeCur Then ' BUG FIX 3/3/96 JK
```

Notice that EOL comments may be continued onto additional lines if necessary as shown in the first example.

Here are some examples of inappropriate EOL comments:

```
nEmployeeCur = nEmployeeCur + 1 ' Add 1 to loop counter
UpdateRecord  ' Call the update routine
```

Do you really want to write every program twice?

[Return to Table Of Contents]